Free Chapter!

DEEP LEARNING: From Basics to Practice

Andrew Glassner

www.glassner.com

@AndrewGlassner

Deep Learning:
From Basics to Practice

www.glassner.com / @AndrewGlassner

First published February 20, 2018

Version 1.0.1 March 3, 2018

Version 1.1 March 22, 2018

Published by The Imaginary Institute, Seattle, WA.

http://www.imaginary-institute.com

Contact: andrew@imaginary-institute.com

Chapter 18

Backpropagation

This chapter is from my book, “Deep Learning: From Principles to Practice,” by Andrew Glassner. I’m making it freely available! Feel free to share this and other bonus chapters with friends and colleagues.

The book is in 2 volumes, available here:
http://amzn.to/2F4nz7k
http://amzn.to/2EQtPR2

You can download all the figures in the entire book, and all the Python notebooks, for free from my GitHub site:
https://github.com/blueberrymusic

To get a free Kindle reader for your device, visit
https://www.amazon.com/kindle-dbs/fd/kcp

Contents

18.1 Why This Chapter Is Here

This chapter is about training a neural network. The very basic idea is appealingly simple. Suppose we're training a categorizer, which will tell us which of several given labels should be assigned to a given input. It might tell us what animal is featured in a photo, or whether a bone in an image is broken or not, or what song a particular bit of audio belongs to.

Training this neural network involves handing it a sample, and asking it to **predict** that sample's label. If the prediction matches the label that we previously determined for it, we move on to the next sample. If the prediction is wrong, we change the network to help it do better next time.

Easily said, but not so easily done. This chapter is about how we "change the network" so that it **learns**, or improves its ability to make correct predictions. This approach works beautifully not just for classifiers, but for almost any kind of neural network.

Contrast a feed-forward network of neurons to the dedicated classifiers we saw in Chapter 13. Each of those dedicated algorithms had a customized, built-in learning method that measured the incoming data to provide the information that classifier needed to know.

But a neural network is just a giant collection of neurons, each doing its own little calculation and then passing on its results to other neurons. Even when we organize them into layers, there's no inherent learning algorithm.

How can we train such a thing to produce the results we want? And how can we do it efficiently?

The answer is called **backpropagation**, or simply **backprop**. Without backprop, we wouldn't have today's widespread use of deep learning, because we wouldn't be able to train our models in reasonable amounts of time. With backprop, deep learning algorithms are practical and plentiful.

Backprop is a low-level algorithm. When we use libraries to build and train deep learning systems, their finely-tuned routines give us both speed and accuracy. Except as an educational exercise, or to implement some new idea, we're likely to never write our own code to perform backprop.

So why is this chapter here? Why should we bother knowing about this low-level algorithm at all? There are at least four good reasons to have a general knowledge of backpropagation.

First, it's important to understand backprop because knowledge of one's tools is part of becoming a master in any field. Sailors at sea, and pilots in the air, need to understand how their autopilots work in order to use them properly. A photographer with an auto-focus camera needs to know how that feature works, what its limits are, and how to control it, so that she can work with the automated system to capture the images she wants. A basic knowledge of the core techniques of any field is part of the process of gaining proficiency and developing mastery. In this case, knowing something about backprop lets us read the literature, talk to other people about deep learning ideas, and better understand the algorithms and libraries we use.

Second, and more practically, knowing about backprop can help us design networks that learn. When a network learns slowly, or not at all, it can be because something is preventing backprop from running properly. Backprop is a versatile and robust algorithm, but it's not bulletproof. We can easily build networks where backprop won't produce useful changes, resulting in a network that stubbornly refuses to learn. For those times when something's going wrong with backprop, understanding the algorithm helps us fix things [Karpathy16].

Third, many important advances in neural networks rely on backprop intimately. To learn these new ideas, and understand why they work the way they do, it's important to know the algorithms they're building on.

Finally, backprop is an elegant algorithm. It efficiently solves a problem that would otherwise require a prohibitive amount of time and computer resources. It's one of the conceptual treasures of the field. As curious, thoughtful people it's well worth our time to understand this beautiful algorithm.

For these reasons and others, this chapter provides an introduction to backprop. Generally speaking, introductions to backprop are presented mathematically, as a collection of equations with associated discussion [Fullér10]. As usual, we'll skip the mathematics and focus instead on the concepts. The mechanics are common-sense at their core, and don't require any tools beyond basic arithmetic and the ideas of a derivative and gradient, which we discussed in Chapter 5.

18.1.1 A Word On Subtlety

The backpropagation algorithm is not complicated. In fact, it's remarkably simple, which is why it can be implemented so efficiently.

But simple does not always mean easy.

The backprop algorithm is subtle. In the discussion below, the algorithm will take shape through a process of observations and reasoning, and these steps may take some thought. We'll try to be clear about every step, but making the leap from reading to understanding may require some work.

It's worth the effort.

18.2 A Very Slow Way to Learn

Let's begin with a very slow way to train a neural network. This will give us a good starting point, which we'll then improve.

Suppose we've been given a brand-new neural network consisting of hundreds or even tens of thousands of interconnected neurons. The network was designed to classify each input into one of 5 categories. So it has 5 outputs, which we'll number 1 to 5, and whichever one has the largest output is the network's prediction for an input's category. Figure 18.1 shows the idea.

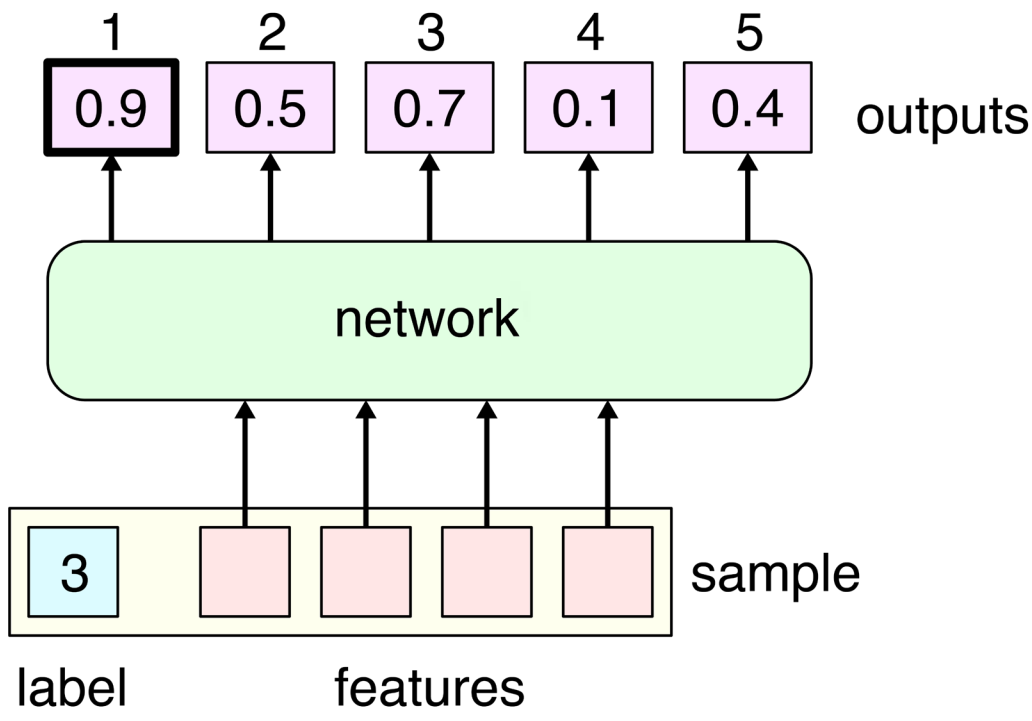

Figure 18.1: A neural network predicting the class of an input sample.

Starting at the bottom of Figure 18.1, we have a sample with four features and a label. The label tells us that the sample belongs to category 3. The features go into a neural network which has been designed to provide 5 outputs, one for each class. In this example, the network has incorrectly decided that the input belongs to class 1, because the largest output, 0.9, is from output number 1.

Consider the state of our brand-new network, before it has seen any inputs. As we know from Chapter 16, each input to each neuron has an associated weight. There could easily be hundreds of thousands, or many millions, of weights in our network. Typically, all of these weights will have been initialized with small random numbers.

Let's now run one piece of labeled training data through the net, as in Figure 18.1. The sample's features go into the first layer of neurons, and the outputs of those neurons go into more neurons, and so on, until they finally arrive at the output neurons, when they become the output of the network. The index of the output neuron with the largest value is the predicted class for this sample.

Since we're starting with random numbers for our weights, we're likely to get essentially random outputs. So there's a 1 in 5 chance the network will happen to predict the right label for this sample. But there's a 4 in 5 chance it'll get it wrong, so let's assume that the network predicts the wrong category.

When the prediction doesn't match the label, we can measure the error numerically, coming up with a single number to tell us just how wrong this answer is. We call this number the **error score**, or **error**, or sometimes the **loss** (if the word "loss" seems like a strange synonym for "error," it may help to think to think of it as describing how much information is "lost" if we categorize a sample using the output of the classifier, rather than the label.).

The error (or loss) is a floating-point number that can take on any value, though often we set things up so that it's always positive. The larger the error, the more "wrong" our network's prediction is for the label of this input.

An error of 0 means that the network predicted this sample's label correctly. In a perfect world, we'd get the error down to 0 for every sample in the training set. In practice, we usually settle for getting as close as we can.

Let's briefly recap some terminology from previous chapters. When we speak of "the network's error" with respect to a training set, we usually mean some kind of overall average that tells us how the network is doing when taking all the training samples into consideration. We call this the **training error**, since it's the overall error we get from predicting results from the training set. Similarly, the error from the test or validation data is called the **test error** or **validation error**. When the system is deployed, a measure of the mistakes it makes on new data is called the **generalization error**, because it represents how well (or poorly) the system manages to "generalize" from its training data to new, real-world data.

A nice way to think about the whole training process is to anthropomorphize the network. We can say that it "wants" to get its error down to zero, and the whole point of the learning process is to help it achieve that goal.

One advantage of this way of thinking is that we can make the network do anything we want, just by setting up the error to "punish" any quality or behavior that we don't want. Since the algorithms we'll see in this chapter are designed to minimize the error, we know that anything about the network's behavior that contributes to the error will get minimized.

The most natural thing to punish is getting the wrong answer, so the error almost always includes a term that measures how far the output is from the correct label. The worse the match between the prediction and the label, the bigger this term will be. Since the network wants to minimize the error, it will naturally minimize such mistakes.

This approach of "punishing" the network through the error score means we can choose to include terms in the error for anything we can measure and want to suppress. For example, another popular measure to add into the error is a **regularization term**, where we look at the magnitude of all the weights in the network. As we'll see later in this chapter, we usually want those weights to be "small," which often means between −1 and 1. As the weights move beyond this range, we

add a larger number to the error. Since the network "wants" the smallest error possible, it will try to keep the weights small so that this term remains small.

All of this raises the natural question of how on earth the network is able to accomplish this goal of minimizing the error. That's the point of this chapter.

Let's start with a basic error measure that only punishes a mismatch between the network's prediction and the label.

Our first algorithm for teaching the network will be just a thought experiment, since it would be absurdly slow on today's computers. But the motivation is right, and this slow algorithm will form the conceptual basis for the more efficient techniques we'll see later in this chapter.

18.2.1 A Slow Way to Learn

Let's stick with our running example of a classifier. We'll give the network a sample and compare the system's prediction with the sample's label.

If the network got it right and predicted the correct label, we won't change anything and we'll move on to the next sample. As the wise man said, "If it ain't broke, don't fix it" [Seung05].

But if the result for a particular sample is incorrect (that is, the category with the highest value does not match our label), we will try to improve things. That is, we'll learn from our mistakes.

How do we learn from this mistake? Let's stick with this sample for a while and try to help the network do a better job with it. First, we'll pick a small random number (which might be positive or negative). Now we'll pick one weight at random from the thousands or millions of weights in the network, and we'll add our small random value to that weight.

Now we'll evaluate our sample again. Everything up to that change will be the same as before. But there will be a chain reaction of changes in the outputs of the neurons starting at the weight we modified. The new weight will produce a new input for the neuron that uses that input, which will change that neuron's output value, which will change the output of every neuron that uses that output, which will change the output of every neuron that uses any of *those* outputs, and so on. Figure 18.2 shows this idea graphically.

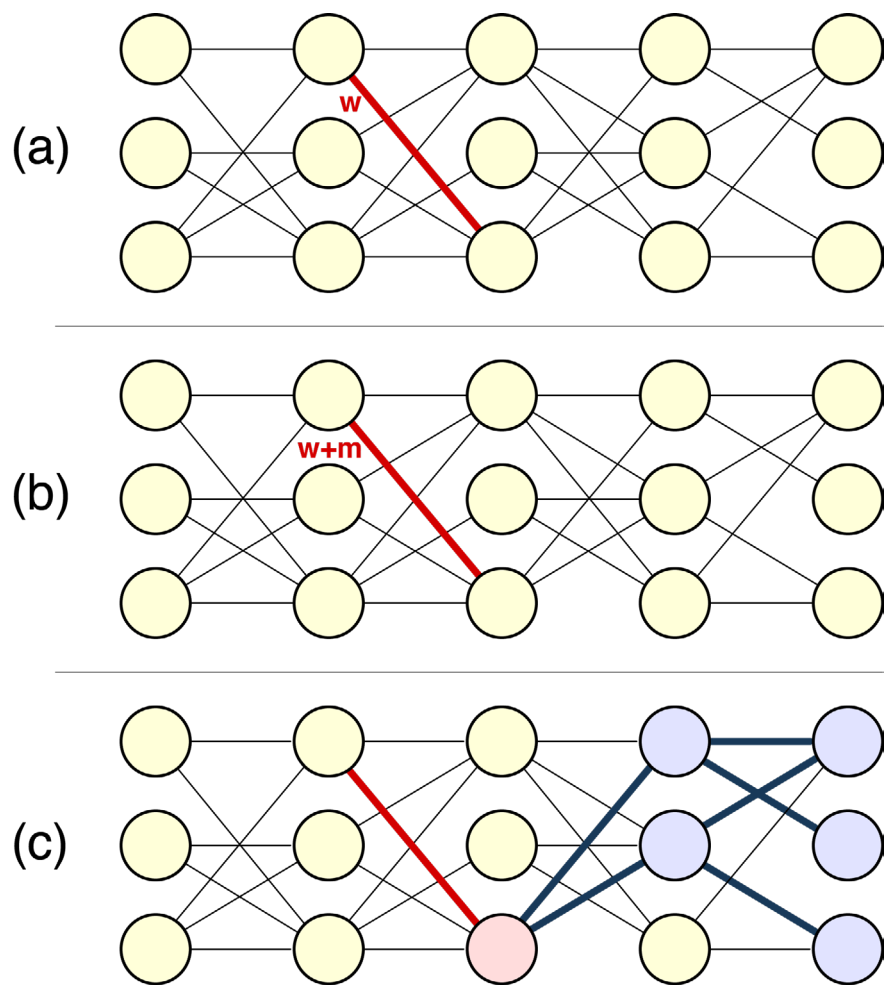

Figure 18.2: Updating a single weight causes a chain reaction that ultimately can change the network's outputs.

Figure 18.2 shows a network of 5 layers with 3 neurons each. Data flows from the inputs at the left to the outputs at the right. For simplicity, not every neuron uses the output of every neuron on the previous layer. In part (a) we select one weight at random, here shown in red and marked w. In part (b) we modify the weight by adding a value m to it, so the weight is now $w+m$. When we run the sample through the network again, as shown in part (c), the new weight causes a change

in the output of the neuron it feeds into (in red). The output of that neuron changes as a result, which causes the neurons it feeds into to change their outputs, and the changes cascade all the way to the output layer.

Now that we have a new output, we can compare it to the label and measure the new error. If the new error is less than the previous error, then we've made things better! We'll keep this change, and move on to the next sample.

But if the results didn't get better then we'll undo this change, restoring the weight back to its previous value. We'll then pick a new random weight, change it by a newly-selected small random amount, and evaluate the network again.

We can continue this process of picking and nudging weights until the results improve, or we decide we've tried enough times, or for any other reason we decide to stop. Then we just move on to the next sample.

When we've used all the samples in our training set, we'll just go through them all again (maybe in a different order), over and over. The idea is that we'll improve a little bit from every mistake.

We can continue this process until the network classifies every input correctly, or we've come close enough, or our patience is exhausted.

With this technique, we would expect the network to slowly improve, though there may be setbacks along the way. For example, adjusting a weight to improve one sample's prediction might ruin the prediction for one or more other samples. If so, when those samples come along they will cause their own changes to improve their performance.

This thought algorithm isn't perfect, because things could get stuck. For example, there might be times when we need to adjust more than one weight simultaneously. To fix that, we can imagine extending our algorithm to assign multiple random changes to multiple random weights. But let's stick with the simpler version for now.

Given enough time and resources, the network would eventually find a value for every weight that either predicts the right answer for every sample, or it comes as close as that network possibly can.

The important word in that last sentence is *eventually*. As in, "The water will boil, eventually," or "The Andromeda galaxy will collide with our Milky Way galaxy, eventually" [NASA12].

This technique, while a valid way to teach a network, is definitely not practical. Modern networks can have millions of weights. Trying to find the best values for all those weights with this algorithm is just not realistic.

But this *is* the core idea. To train our network, we'll watch its output, and when it makes mistakes, we'll adjust the weights to make those mistakes less likely. Our goal in this chapter will be to take this rough idea and re-structure it into a vastly more practical algorithm.

Before we move on, it's worth noting that we've been talking about weights, but not the bias term belonging to every neuron. We know that every neuron's bias gets added in along with the neuron's weighted inputs, so changing the bias would also change the output. Doesn't that mean that we want to adjust the bias values as well? We sure do. But thanks to the **bias trick** we saw in Chapter 10, we don't have to think about the bias explicitly. That little bit of relabeling sets up the bias to look like an input with its own weight, just like all the other inputs. The beauty of this arrangement is that it means that as far as our training algorithm is concerned, the bias is just another weight to adjust. In other words, all we need to think about is adjusting weights, and the bias weights will automatically get adjusted along the way with all the other weights.

Let's now consider how we might improve our incredibly slow weight-changing algorithm.

18.2.2 A Faster Way to Learn

The algorithm of the last section would improve our network, but at a glacial pace.

One big source of inefficiency is that half of our adjustments to the weights are in the wrong direction: we add a value when we should instead have subtracted it, and vice-versa. That's why we had to undo our changes when the error went up. Another problem is that we tuned each weight one by one, requiring us to evaluate an immense number of samples. Let's solve these problems.

We could avoid making mistakes if we knew beforehand whether we wanted to nudge each weight along the number line to the right (that is, make it more positive) or to the left (and make it more negative).

We can get exactly that information from the **gradient** of the error with respect to that weight. Recall that we met the gradient in Chapter 5, where it told us how the height of a surface changes as each of its parameters changes. Let's narrow that down for the present case. In 1D (where the gradient is also called the **derivative**), the gradient is the slope of a curve above a specific point. Our curve describes the network's error, and our point is the value of a weight. If the slope of the error (the gradient) above the weight is positive (that is, the line goes up as we move to the right), then moving the point to the right will cause the error to go up. More useful to us is that moving the point to the left will cause the error to go down. If the slope of the error is negative, the situations are reversed.

Figure 18.3 shows two examples.

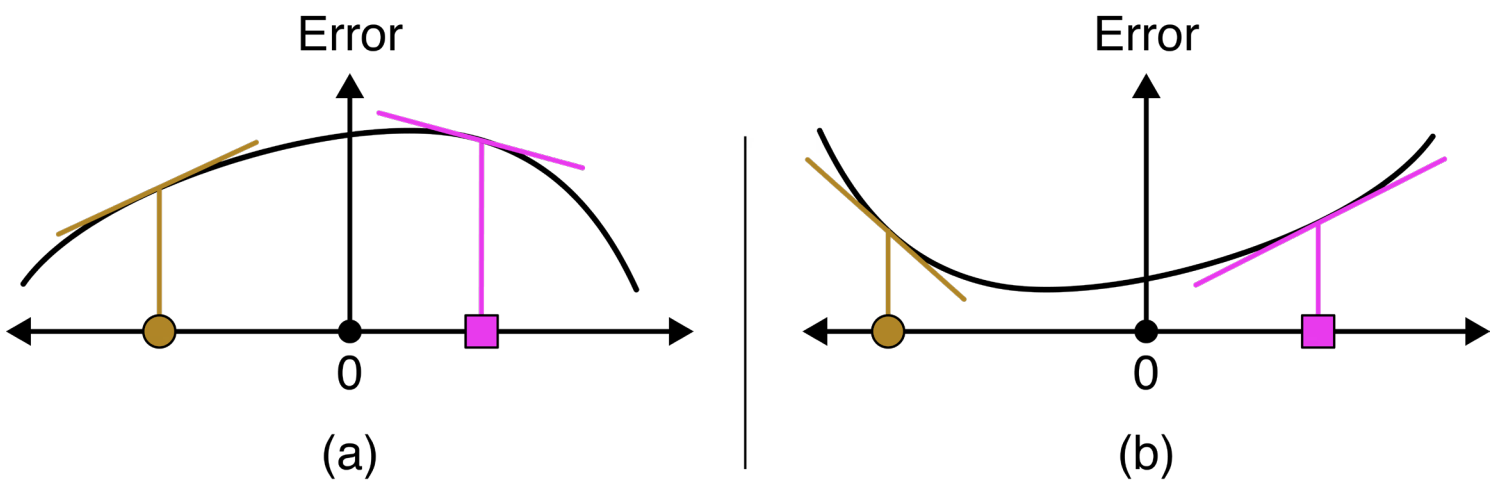

Figure 18.3: The gradient tells us what will happen to the error (the black curves) if we move a weight to the right. The gradient is given by the slope of the curve directly above the point we're interested in. Lines that go up as we move right have a positive slope, otherwise they are negative.

In Figure 18.3(a), we see that if we move the round weight to the right, the error will increase, because the slope of the error is positive. To reduce the error, we need to move the round point left. The square point's gradient is negative, so we reduce the error by moving that point right. Part (b) shows the gradient for the round point is negative, so moving to the right will reduce the error. The square point's gradient is positive, so we reduce the error by moving that point to the left.

If we had the gradient for a weight, we could always adjust it exactly as needed to make the error go down.

Using the gradients wouldn't be much of an advantage if they were time-consuming to compute, so as our second improvement let's suppose that we can calculate the gradients for the weights very efficiently. In fact, let's suppose that we could quickly calculate the gradient for *every* weight in the whole network. Then we could update *all of the weights simultaneously* by adding a small value (positive or negative) to each weight in the direction given by its own individual gradient. That would be an immense time-saver.

Putting these together gives us a plan where we'll run a sample through the network, measure the output, compute the gradient for every weight, and then use the gradient at each weight to move that weight to the right or the left. This is exactly what we're going to do.

This plan makes knowing the gradient an important issue. Finding the gradient efficiently is the main goal of this chapter.

Before we continue, it's worth noticing that this algorithm makes the assumption that tweaking all the weights independently and simultaneously will lead to a reduction in the error. This is a bold assumption, because we've already seen how changing one weight can cause ripple effects through the rest of the network. Those effects could change the values of other neurons, which in turn would change their gradients. We won't get into the details now, but we'll see later that if we make the changes to the weights small enough, that assumption will generally hold true, and the error will indeed go down.

18.3 No Activation Functions for Now

For the next few sections in this chapter, we're going to simplify the discussion by pretending that our neurons don't have activation functions.

As we saw in Chapter 17, activation functions are essential to keep our whole network from becoming nothing more than the equivalent of a single neuron. So we need to use them.

But if we include them in our initial discussion of backprop, things will get complicated, fast. If we leave activation functions out for just a moment, the logic is much easier to follow. We'll put them back in again at the end.

Since we're temporarily pretending that there are no activation functions in our neurons, neurons in the following discussions just sum up their weighted inputs, and present that sum as their output, as in Figure 18.4. As before, each weight is named with a two-letter composite of the neuron it's coming from and the neuron it's going into.

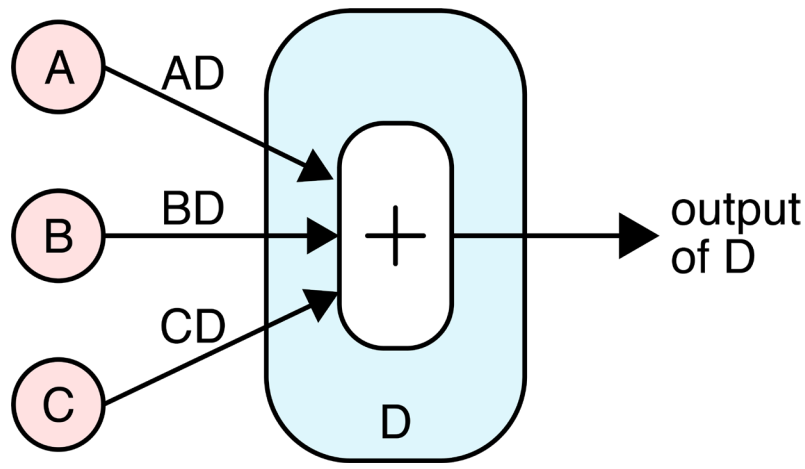

Figure 18.4: Neuron D simply sums up its incoming values, and presents that sum as its output. Here we've explicit named the weights on each connection into neuron D.

Until we put explicitly put activation functions back in, our neurons will emit nothing more than the sum of their weighted inputs.

18.4 Neuron Outputs and Network Error

Our goal is to reduce the overall error for a sample, by adjusting the network's weights.

We'll do this in two steps. In the first step, we calculate and store a number called the "delta" for every neuron. This number is related to the network's error, as we'll see below. This step is performed by the **backpropagation** algorithm.

The second step uses those delta values at the neurons to update the weights. This step is called the **update** step. It's not typically considered part of backpropagation, but sometimes people casually roll the two steps together and call the whole thing "backpropagation."

The overall plan now is to run a sample through the network, get the prediction, and compare that prediction to the label to get an error. If their error is greater than 0, we use it to compute and store a number we'll call "delta" at every neuron. We use these delta values and the neuron outputs to calculate an update value for each weight. The final step is to apply every weight's individual update so that it takes on a new value.

Then we move on to the next sample, and repeat the process, over and over again until the predictions are all perfect or we decide to stop.

Let's now look at this mysterious "delta" value that we store at each neuron.

18.4.1 Errors Change Proportionally

There are two key observations that will make sense of everything to follow. These are both based on how the network behaves when we ignore the activation functions, which we're doing for the moment. As promised above, we'll put them back in later in this chapter.

The first observation is this: *When any neuron output in our network changes, the output error changes by a proportional amount.*

Let's unpack that statement.

Since we're ignoring activation functions, there are really only two types of values we care about in the system: weights (which we can set and change as we please), and neuron outputs (which are computed automatically, and which are beyond our direct control). Except for the very first layer, a neuron's input values are each the output of a

previous neuron times the weight of the connection that output travels on. Each neuron's output is just the sum of all of these weighted inputs. Figure 18.5 recaps this idea graphically.

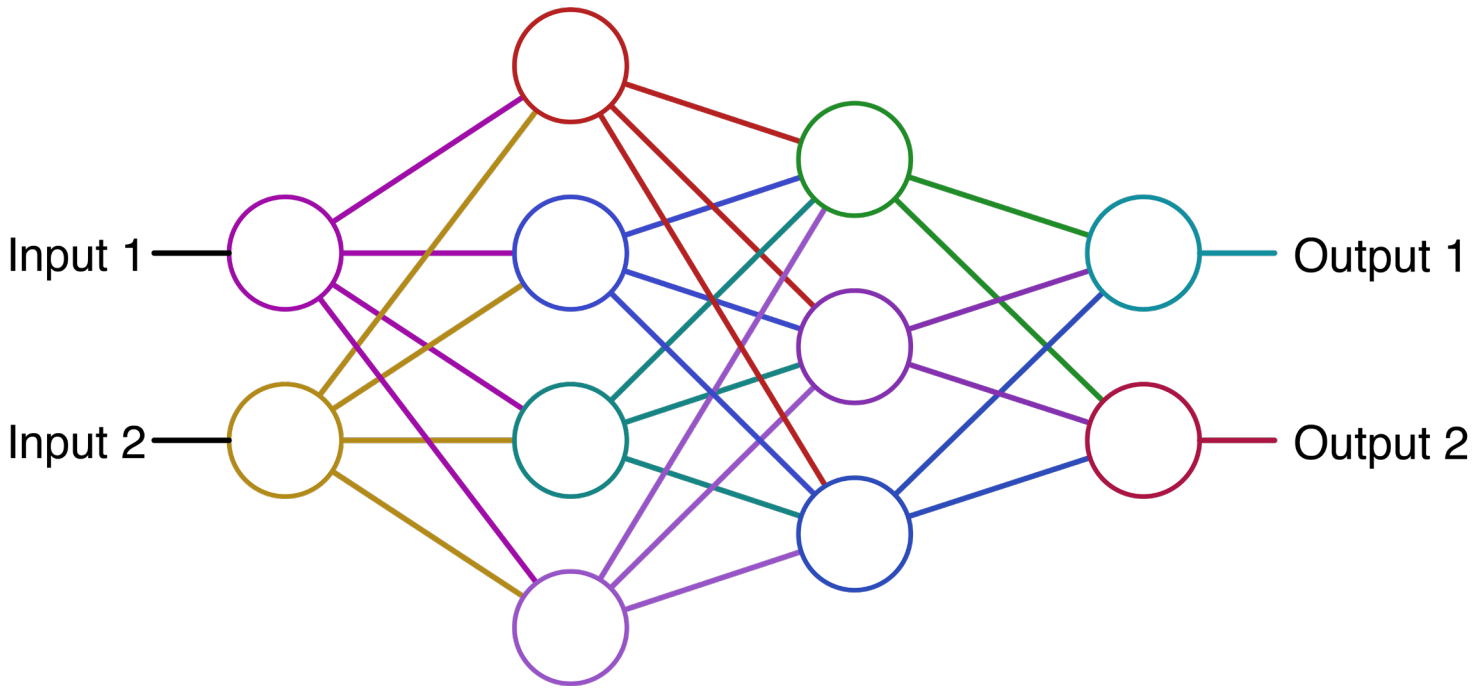

Figure 18.5: A small neural network with 11 neurons organized in 4 layers. Data flows from the inputs at the left to the outputs at the right. Each neuron's inputs come from the outputs of the neurons on the previous layer. This type of diagram, though common, easily becomes dense and confusing, even with color-coding. We will avoid it when possible.

We know that we'll be changing weights to improve our network. But sometimes it's easier to think about looking at the change in a neuron's output. As long as we keep using the same input, the only reason a neuron's output can change is because one of its weights has changed. So in the rest of this chapter, any time we speak of the result of a change in a neuron's output, that came about because we changed one of the weights that neuron depended on.

Let's take this point of view now, and imagine we're looking at a neuron whose output has just changed. What happens to the network's error as a result? Because the only operations that are being carried out in our network are multiplication and addition, if we work through the numbers we'll see that the result of this change is that the change in the error is **proportional** to the change in the neuron's output.

In other words, to find the change in the error, we find the change in the neuron's output and multiply that by some particular value. If we double the amount of change in the neuron's output, we'll double the amount of change in the error. If we cut the neuron's output change by one-third, we'll cut the change in the output by one-third.

The connection between any *change* in the neuron's output and the resulting *change* in the final error is just the neuron's change times some number. This number goes by various names, but the most popular is probably the lower-case Greek letter δ (delta), though sometimes the upper-case version, Δ, is used. Mathematicians often use the delta character to mean "change" of some sort, so this was a natural (if terse) choice of name.

So every neuron has a "delta," or δ, associated with it. This is a real number that can be big or small, positive or negative. If the neuron's output changes by a particular amount (that is, it goes up or down), we multiply that change by that neuron's delta, and that tells us how the entire network's output will change.

Let's draw a couple of pictures to show the "before" and "after" conditions of a neuron whose output changes. We'll change the output of the neuron using brute force: we'll add some arbitrary number to the summed inputs just before that value emerges as the neuron's output. As in Figure 18.2, we'll use the letter m (for "modification") for this extra value.

Figure 18.6 shows the idea graphically.

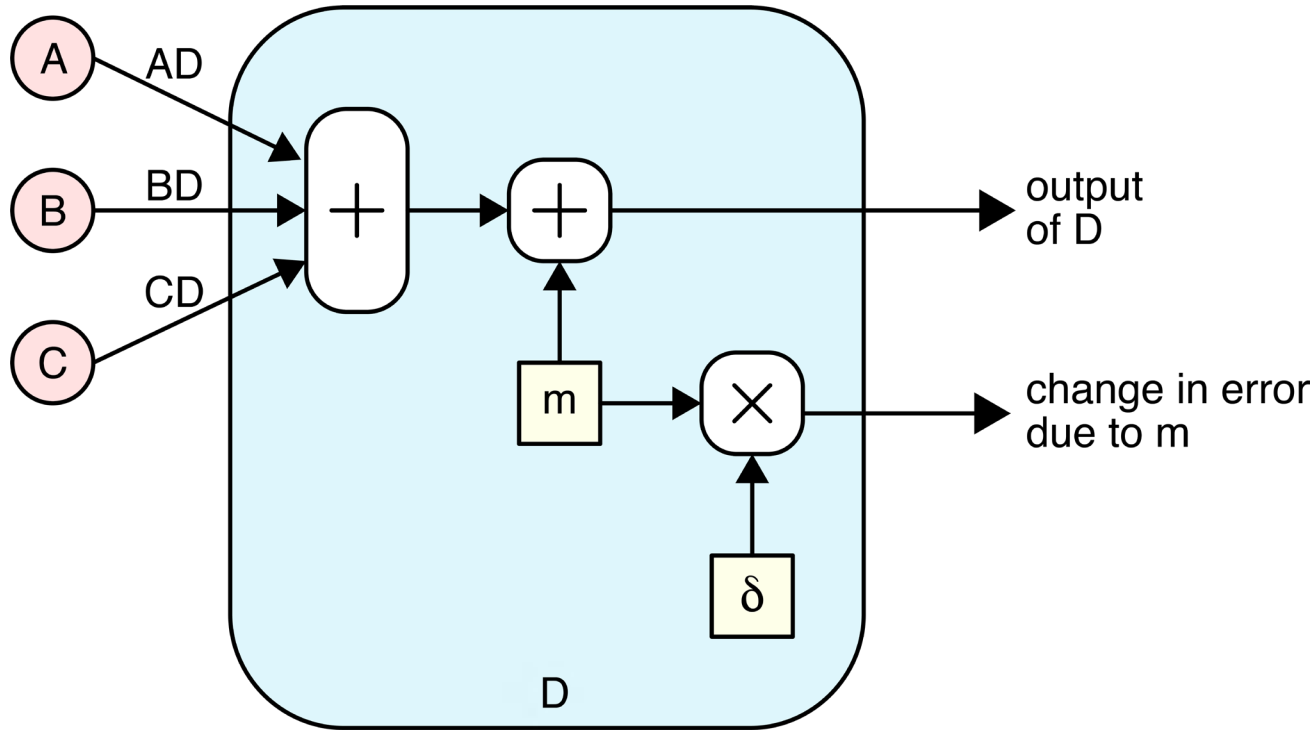

Figure 18.6: Computing the change in the error due to a change in a neuron's output. Here we're forcing a change in the neuron's output by adding an arbitrary amount m to the sum of the inputs. Because the output will change by m, we know the change in the error is this difference m times the value of δ belonging to this neuron.

In Figure 18.6 we placed the value m inside the neuron. But we can also change the output by changing one of the inputs. Let's change the value that's coming in from neuron B. We know that the output of B will get multiplied by the weight BD before it's used by neuron D. So let's add our value m right after that weight has been applied. This will have the same result as before, since we're just adding m to the overall sum that emerges from D. Figure 18.7 shows the idea. We can find the change in the output like before, multiplying this change m in the output by δ.

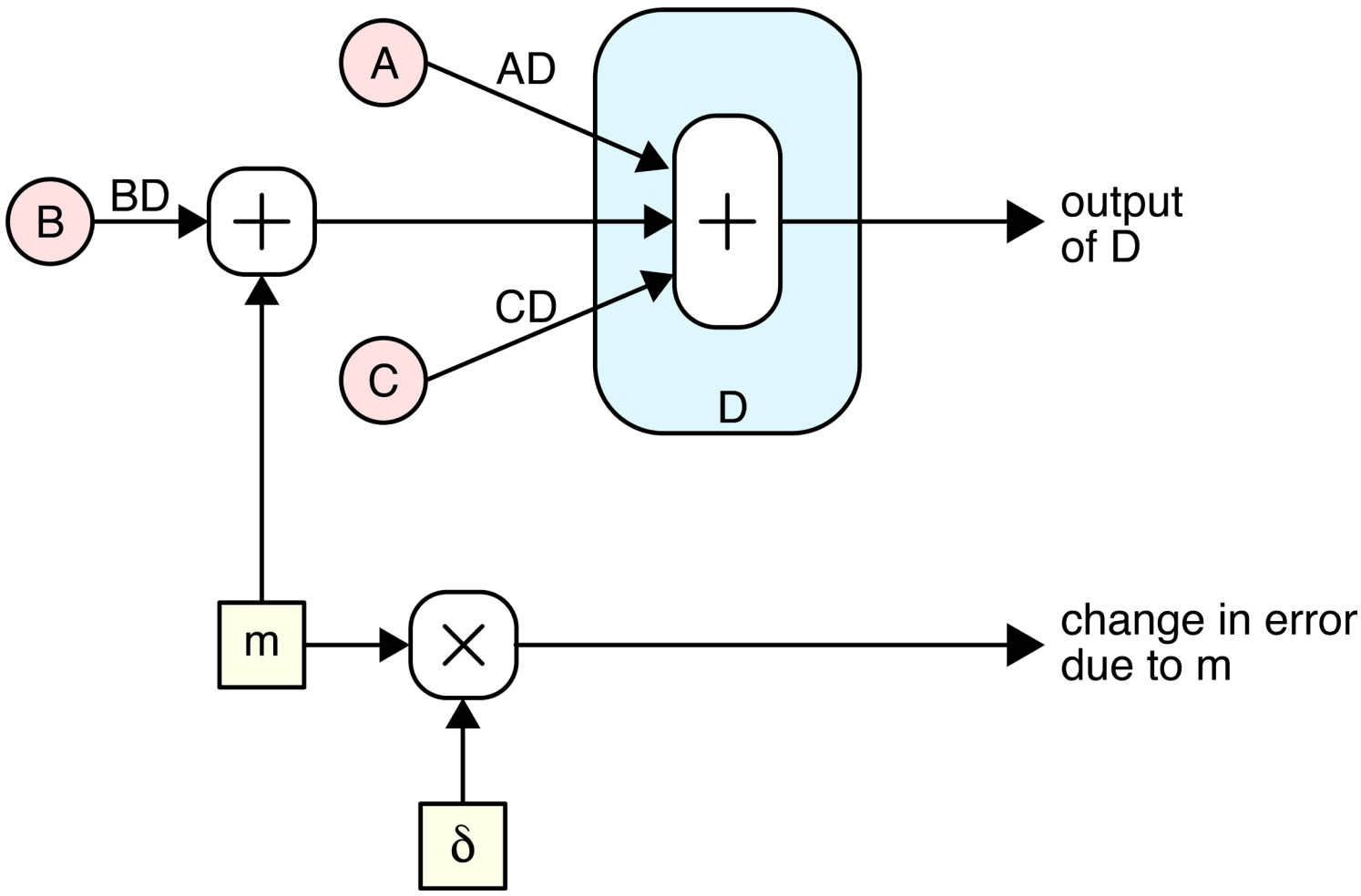

Figure 18.7: A variation of Figure 18.6, where we add *m* to the output of B (after it has been multiplied by the weight BD). The output of D is again changed by *m*, and the change in the error is again *m* times this neuron's value of δ.

To recap, if we know the change in a neuron's output, and we know the value of delta for that neuron, then we can predict the change in the error by multiplying that change in the output by that neuron's delta.

This is a remarkable observation, because it shows us explicitly how the error changes based on the change in output of each neuron. The value of delta acts like an amplifier, making any change in the neuron's output have a bigger or smaller effect on the network's error.

An interesting result of multiplying the neuron's change in output with its delta is that if the change in the output and the value of delta both have the same **sign** (that is, both are positive or negative), then the change in the error will be positive, meaning that the error will increase. If the change in the output and delta have opposite signs (that is, one

is negative and one is positive), then the change in the error will be negative, meaning that the error will decrease. That's the case we want, since our goal is always to make the error as small as possible.

For instance, suppose that neuron A has a delta of 2, and for some reason its output changes by −2 (say, changing from 5 to 3). Since the delta is positive and the change in output is negative, the change in the error will also be negative. In numbers, $2 \times -2 = -4$, so the error will drop by 4.

On the other hand, suppose the delta of A is −2, and its output changes by +2 (say from 3 to 5). Again, the signs are different, so the error will change by $-2 \times 2 = -4$, and again the error will reduce by 4.

But if the change in A's output is −2, and the delta is also −2, then the signs are the same. Since $-2 \times -2 = 4$, the error will increase by 4.

At the start of this section we said there were two key observations we wanted to note. The first, as we've been discussing, is that if a neuron's output changes, the error changes by a proportional amount.

The second key observation is: *this whole discussion applies just as well to the weights*. After all, the weights and the outputs are multiplied together. When we multiply two arbitrary numbers, such as a and b, then we make the result bigger by adding something to *either* the value of a or b. In terms of our network, we can say that *when any weight in our network changes, the error changes by a proportional amount*.

If we wanted, we could work out a delta for every weight. And that would be perfect. We would know just how to tweak each weight to make the error go down. We just add in a small number whose sign is opposite that of the weight's delta.

Finding those deltas is what backprop is for. We find them by first finding the delta for every neuron's output. We'll see below that with a neuron's delta, and its output, we can find the weight deltas.

We already know every neuron's outputs, so let's turn our attention to finding those neuron deltas.

The beauty of backpropagation is that finding those values is incredibly efficient.

18.5 A Tiny Neural Network

To get a handle on backprop, we'll use a tiny network that classifies 2D points into two categories, which we'll call class 1 and class 2. If the points can be separated by a straight line then we could do this job with just one perceptron, but we'll use a little network because it lets us see the general principles.

In this section we'll look at the network and give a label to everything we care about. That will make later discussions simpler and easier to follow.

Figure 18.8 shows our network. The inputs are the X and Y coordinates of each point, there are four neurons, and the outputs of the last two serve as the outputs of the network. We call their outputs the predictions P1 and P2. The value of P1 is the network's prediction of the likelihood that our sample (that is, the X and Y at the input) belongs to class 1, and P2 is its prediction of the likelihood that the sample belongs to class 2. These aren't actually probabilities because they won't necessarily add up to 1, but whichever is larger is the network's preferred choice of category for this input. We could make them into probabilities (by adding a **softmax** layer, as discussed in Chapter 17), but that would just make the discussion more complicated without adding anything useful.

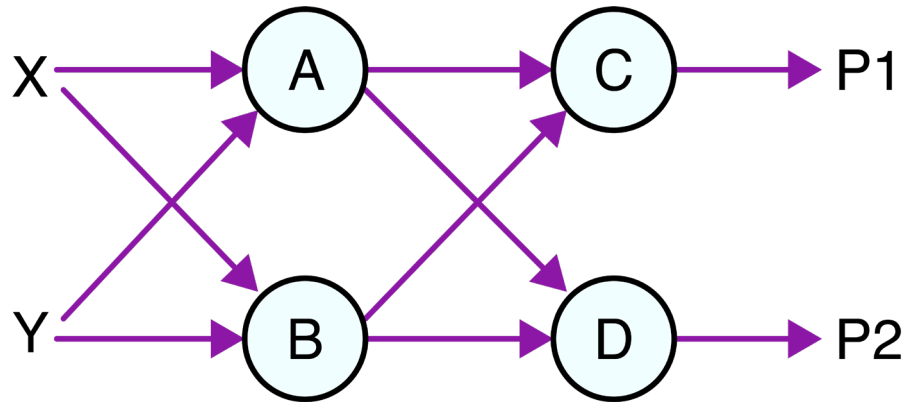

Figure 18.8: A simple network. The input has two features, which we call X and Y. There are four neurons, ending with two predictions, P1 and P2. These predict the likelihoods (not the probabilities) that our sample belongs to class 1 or class 2, respectively.

Let's label the weights. As usual, we'll imagine that the weights are sitting on the wires that connect neurons, rather than stored inside the neurons. The name of each weight will be the name of the neuron providing that value at its output followed by the neuron using that value as input. Figure 18.9 shows the names of all 8 weights in our network.

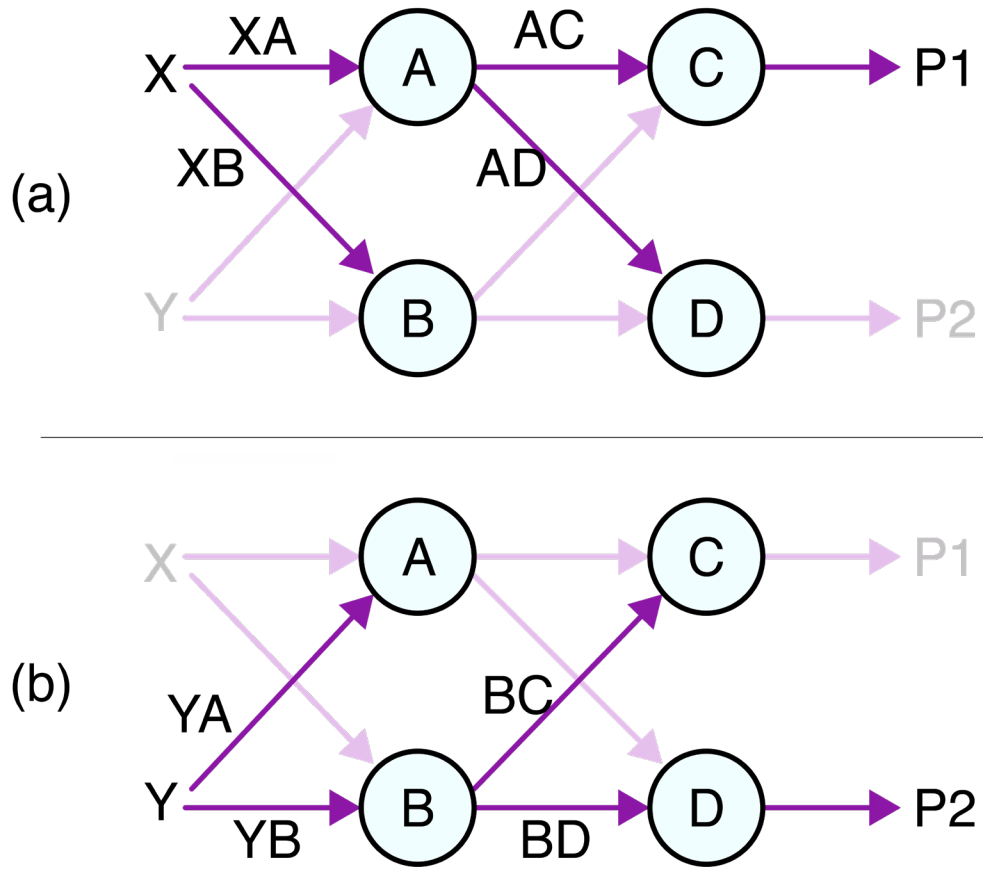

Figure 18.9: Giving names to each of the 8 weights in our tiny network. Each weight is just the name of the two neurons it connects, with the starting neuron (on the left) first, and then the destination neuron (on the right) second. For the sake of consistency, we pretend that X and Y are "neurons" when it comes to naming the weights, so *XA* is the name of the weight that scales the value of X going into neuron A.

This is a tiny **deep-learning network** with two **layers**. The first layer contains neurons A and B, and the second contains neurons C and D, as shown in Figure 18.10.

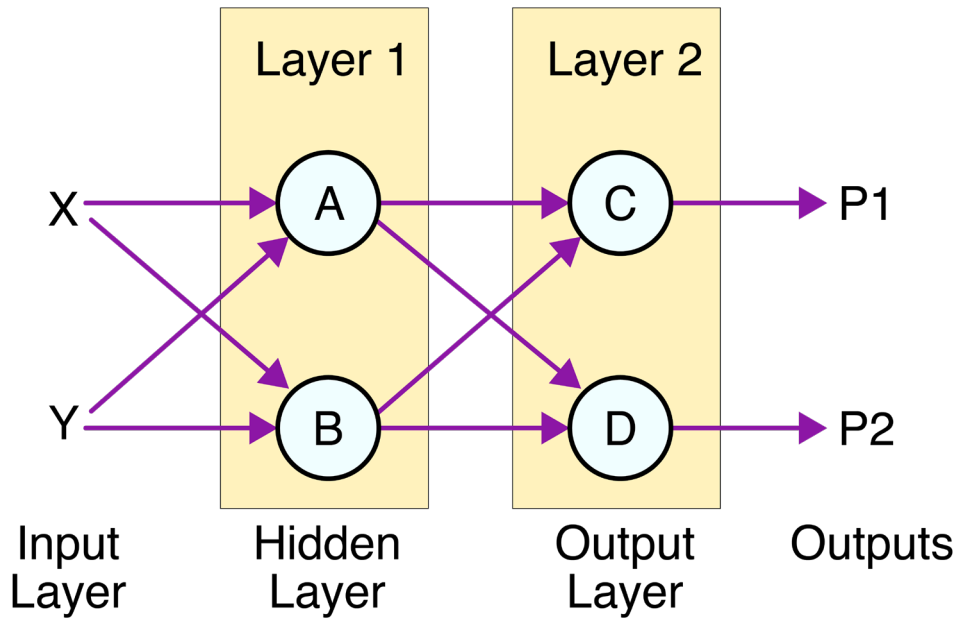

Figure 18.10: Our tiny neural network has 2 layers. The input layer doesn't do any computing, so it's usually not included in the layer count.

Two layers is not a terribly deep network, and two neurons per layers is not a lot of computing power. We usually work with systems with more layers, and more neurons on each layer. Determining how many layers we should use for a given task, and how many neurons should be on each layer, is something of an art and an experimental science. In essence, we usually take a guess at those values, and then vary our choices to try to improve the results.

In Chapter 20 we'll discuss deep learning and its terminology. Let's jump ahead a little bit here and use that language for the various pieces of the network in Figure 18.10. The **input layer** is just a conceptual grouping of the inputs X and Y. These don't correspond to neurons, because these are just pieces of memory for storing the features in the sample that's been given to the network. When we count up the layers in a network, we don't usually count the input layer.

The **hidden layer** is called that because neurons A and B are "inside" the network, and thus "hidden" from a viewer on the outside, who can see only the inputs and outputs. The **output layer** is the set of neurons that provide our **outputs**, here P1 and P2. These layer names are a little asymmetrical because the input layer has no neurons, and the output layer does, but they're how the convention has developed.

Finally, we'll want to refer to the output and delta for every neuron. For this, we'll make little two-letter names by combining the neuron's name with the value we want to refer to. So *Ao* and *Bo* will be the names of the outputs of neurons A and B, and *Aδ* and *Bδ* will be the delta values for those two neurons.

Figure 18.11 shows these values stored with their neurons.

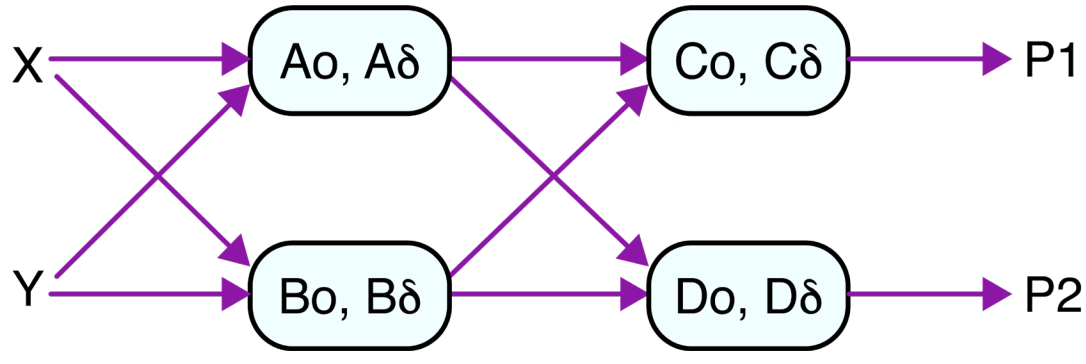

Figure 18.11: Our simple network with the output and delta values for each neuron.

We'll be watching what happens when neuron outputs change, causing changes to the error. We'll label the change in the output of neuron A as *Am*. We'll label the error simply *E*, and a change to the error as *Em*.

As we saw above, if we have a change *Am* in the output of neuron A, then multiplying that change by *Aδ* gives us the change in the error. That is, the change *Em* is given by $Am \times A\delta$. We'll think of the action of *Aδ* as multiplying, or scaling, the change in the output of neuron A, giving us the corresponding change in the error. Figure 18.12 shows the schematic setup we'll use for visualizing the way changes in a neuron's output are scaled by its delta to produce changes to the error.

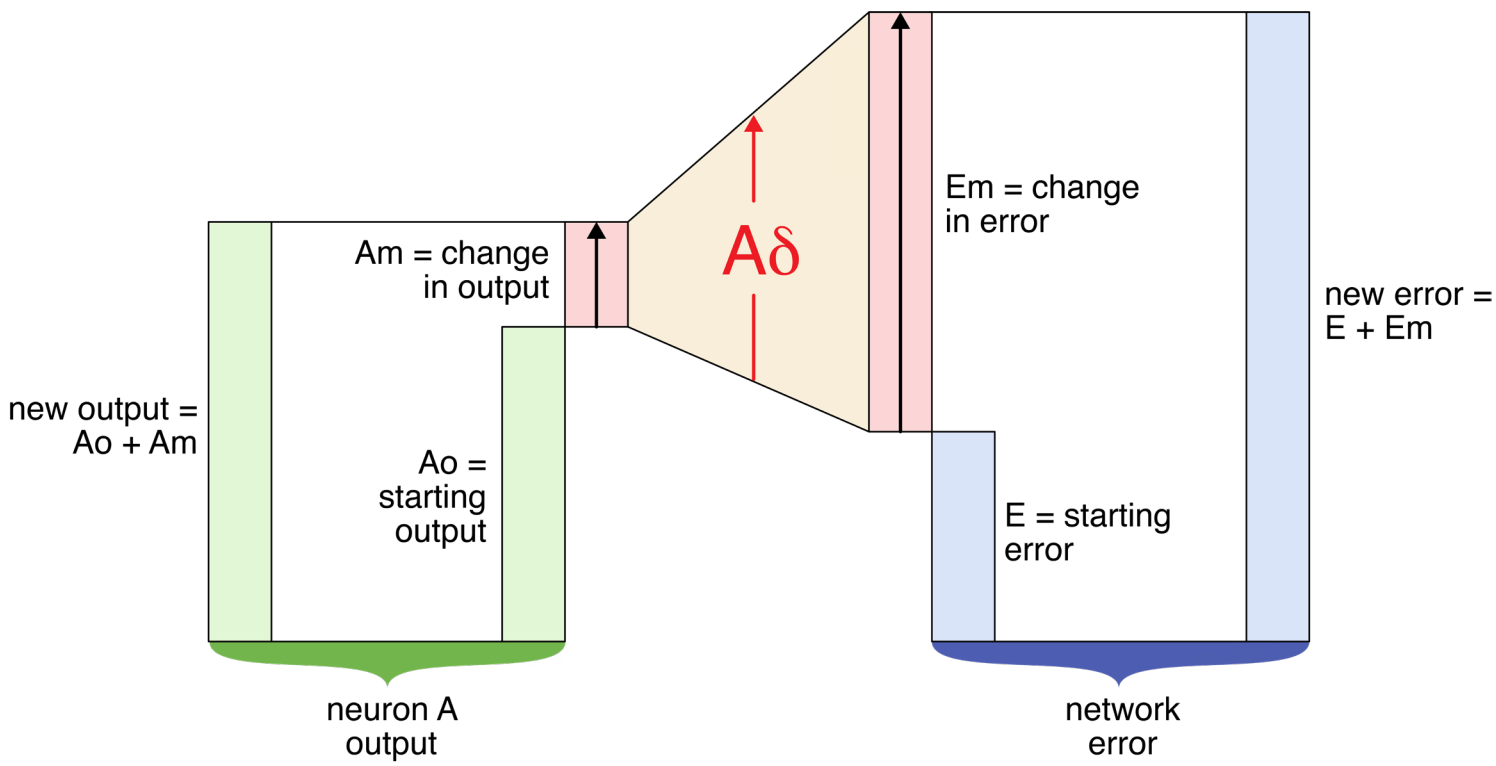

Figure 18.12: Our schematic for visualizing how changes in a neuron's output can change the network's error. Read the diagram roughly left to right.

At the left of Figure 18.12 we start with a neuron A. It starts with value *Ao*, but we change one of the weights on its inputs so that the output goes up by *Am*. The arrow inside the box for *Am* shows that this change is positive. This change is multiplied by $A\delta$ to give us *Em*, the change in the error. We show $A\delta$ as a wedge, illustrating the amplification of *Em*. Adding this change to the previous value of the error, *E*, gives us the new error *E+Em*. In this case, both *Am* and $A\delta$ are positive, so the change in the error $Am \times A\delta$ is also positive, increasing the error.

Keep in mind that the delta value $A\delta$ relates a *change* in a neuron's output to a *change* in the error. These are not relative or percentage changes, but the actual amounts. So if the output of A goes from 3 to 5, that's a change of 2, so the change in the error would be $A\delta \times 2$. If the output of A goes from 3000 to 3002, that's still a change of 2, and the error would change by the same amount, $A\delta \times 2$.

Now that we've labeled everything, we're finally ready to look at the backpropagation algorithm.

18.6 Step 1: Deltas for the Output Neurons

Backpropagation is all about finding the delta value for each neuron. To do that, we'll find gradients of the error at the end of the network, and then propagate, or move, those gradients back to the start. So we'll begin at the end: the output layer.

The outputs of neuron C and D in our tiny network give us the likelihoods that the input is in class 1 or class 2, respectively. In a perfect world, a sample that belongs to group 1 would produce a value of 1.0 for P1 and 0.0 for P2, meaning that the system is certain that it belongs to class 1 and simultaneously certain that it does *not* belong to class 2.

If the system's a little less certain, we might get P1=0.8 and P2=0.1, telling us that it's much more likely that the sample is in class 1 (remember that these aren't probabilities, so they probably won't sum to 1).

We'd like to come up with a single number to represent the network's error. To do that, we'll compare the values of P1 and P2 with the label for this sample.

The easiest way to make that comparison is if the label is **one-hot encoded**, as we saw in Chapter 12. Recall that one-hot encoding makes a list of numbers as long as the number of classes, and puts a 0 in every entry. Then it puts a 1 in the entry corresponding to the correct class. In our case, we have only two classes, so the encoder would always start with list of two zeros, which we can write as (0, 0). For a sample that belongs to class 1, it would put a 1 in the first slot, giving us (1, 0). A sample from class 2 would get the label (0, 1). Sometimes such a label is also called a **target**.

Let's put the predictions P1 and P2 into a list as well: (P1, P2). Now we can just compare the lists. There are lots of ways to do this. For example, a simple way would be to find the difference between corresponding elements from each list and then add up those differences. Figure 18.13 shows the idea.

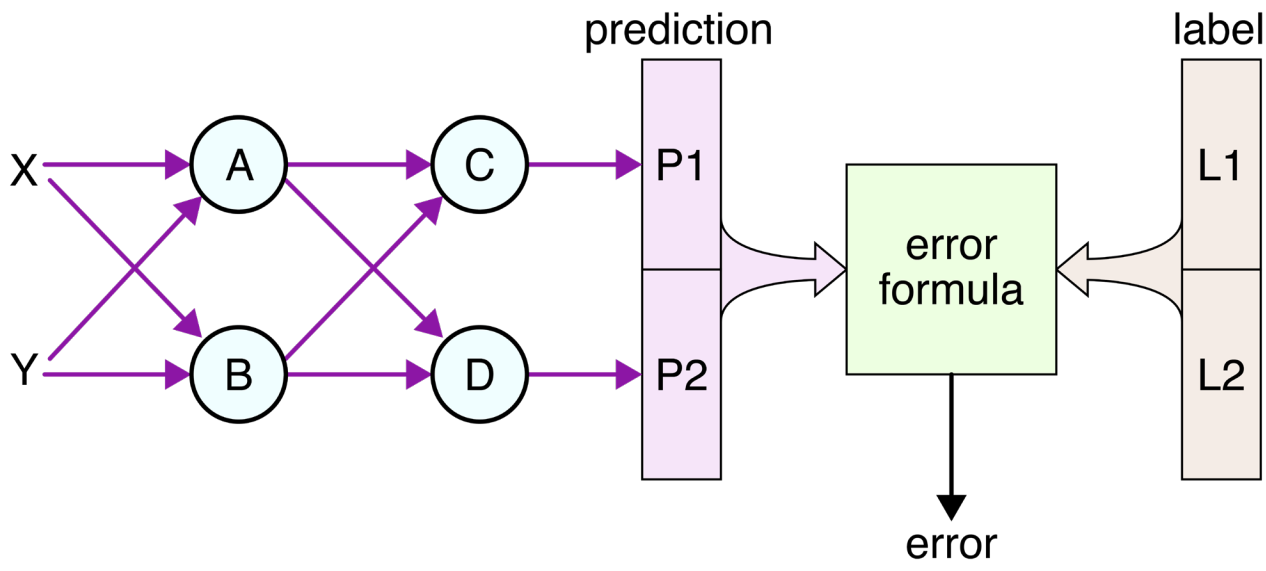

Figure 18.13: To find the error from a specific sample, we start by supplying the sample's features X and Y to the network. The outputs are the predictions P1 and P2, telling us the likelihoods that the sample is in class 1 and class 2, respectively. We compare those predictions with the one-hot encoded label, and from that come up with a number representing the error. If the predictions match the label perfectly, the error is 0. The bigger the mismatch, the bigger the error.

If the prediction list is identical to the label list, then the error is 0. If the two lists are close (say, (0.9, 0.1) and (1, 0)), then we'd want to come up with an error number that's bigger than 0, but maybe not enormous. As the lists become more and more different, the error should increase. The maximum error would come if the network is absolutely wrong, for example predicting (1, 0) when the label says (0, 1).

There are many formulas for calculating the network error, and most libraries let us choose among them. We'll see in Chapters 23 and 24 that this error formula is one of the critical choices that defines what our network is for. For instance, we'll choose one type of error formula if we're building a network to classify inputs into categories, another type of formula if our network is for predicting the next value in a

sequence, and yet another type of formula if we're trying to match the output of some other network. These formulas can be mathematically complex, so we won't go into those details here.

As Figure 18.13 shows, that formula for our simple network takes in 4 numbers (2 from the prediction and 2 from the label), and produces a single number as a result.

But all the error formulas share the property that when they compare a classifier's output to the label, a perfect match will give a value of 0, and increasingly incorrect matches will give increasingly large errors.

For each type of error formula, our library function will also provide us with its **gradient**. The gradient tells us how the error will change if we increase any one of the four inputs. This may seem redundant, since we know that we want the outputs to match the label, so we can tell how the outputs should change just by looking at them. But recall that the error can include other terms, like the regularization term we discussed above, so things can get more complicated.

In our simple case, we can use the gradient to tell us whether we'd like the output of C to go up or down, and the same for D. We'll pick the direction for each neuron that causes the error to decrease.

Let's think about drawing our error. We could also draw the gradient, but usually that's harder to interpret. When we draw the error itself, we can usually see the gradient just by looking at the slope of the error.

Unfortunately, we can't draw a nice picture of the error for our little network because it would require five dimensions (four for the inputs and one for the output). But things aren't so bad. We don't care about how the error changes when the label changes, because the label can't change. For a given input, the label is fixed. So we can ignore the two dimensions for the labels. That leaves us with just 3 dimensions.

And we can draw a 3D shape! So let's plot the error. Remember that we can visualize the gradient at any point just by imagining which way a drop of water would flow if we placed it on the surface of the error above that location.

Let's draw a 3D diagram that shows the error for any set of values P1 and P2, *for a given label*. That is, we'll set the value of the label, and explore the error for different values of P1 and P2. Every error formula will give us a somewhat different surface, but most will look roughly like Figure 18.14.

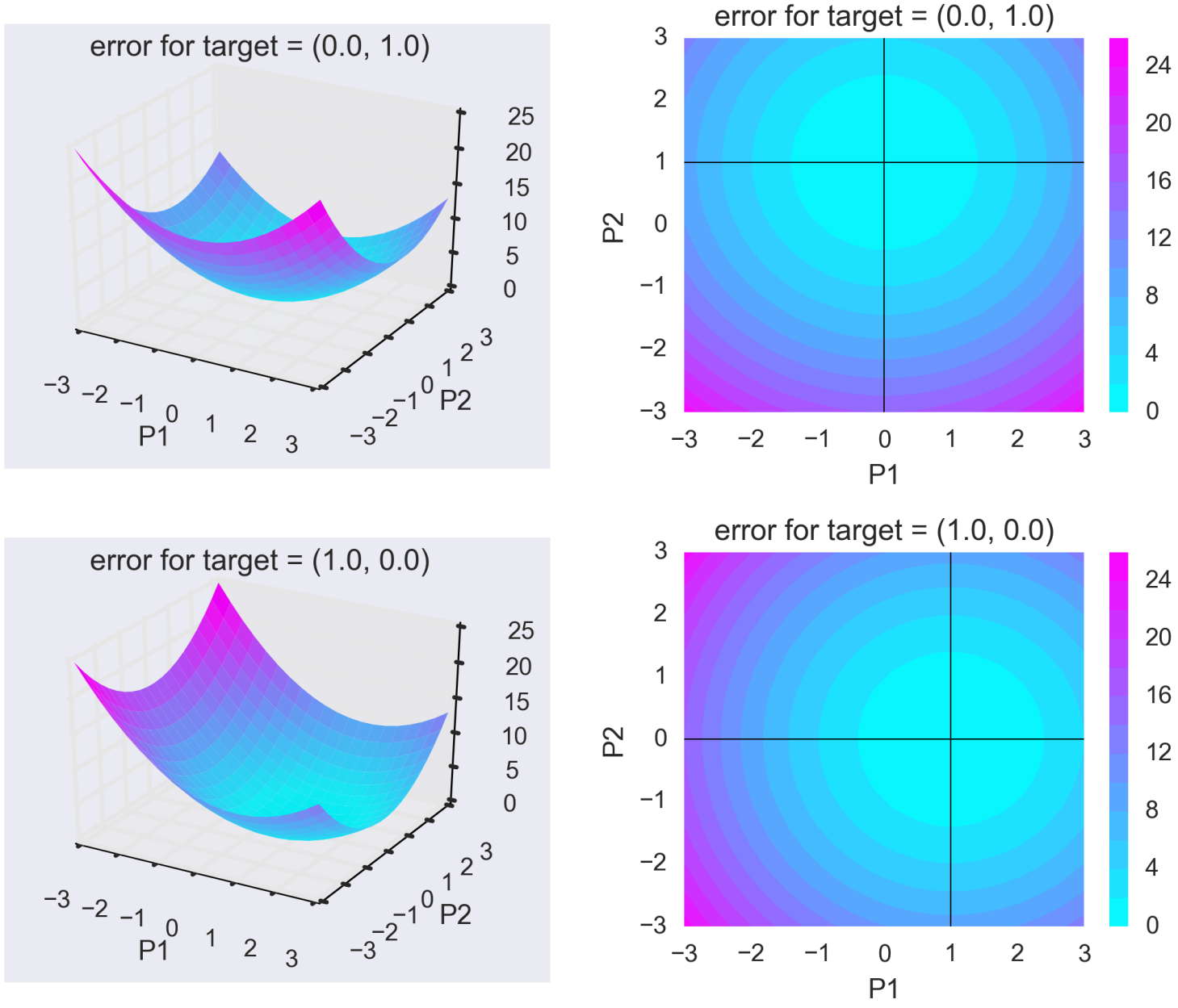

Figure 18.14: Visualizing the error of our network, given a label (or target) and our two predictions, P1 and P2. Top row: The label is (0, 1), so the error is a bowl with its bottom at P1 = 0 and P2 = 1. As P1 and P2 diverge from those values, the error goes up. Left: The error for each value of P1 and P2. Right: A top-down view of the surface at the left, showing the height using colored contours. Bottom row: The label is (1, 0), so now the error is a bowl with the bottom at P1 = 1 and P2 = 0.

For both labels, the shape of the error surface is the same: a bowl with a rounded bottom. The only difference is the location of the bottom of the bowl, which is directly over the label. This makes sense, because our whole intention is to get P1 and P2 to match the label. When they do, we have zero error. So the bottom of the bowl has a value of 0, and it sits right on top of the label. The more different P1 and P2 are from the label, the more the error grows.

These plots let us make a connection between the gradient of this error surface and the delta values for the output layer neurons C and D. It will help to remember that P1, the likelihood of the sample belonging to class 1, is just another name for the output of C, which we also call *Co*. Similarly, P2 is another name for *Do*. So if we say that we want to see a specific change in the value of P1, we're saying that we want the output of neuron C to change in that way, and the same is true for P2 and D.

Let's look at one of these error surfaces a little more carefully so we can really get a feeling for it. Suppose we have a label of (1,0), like in the bottom row of Figure 18.14. Let's suppose that for a particular sample, output P1 has the value –1, and output P2 has the value 0. In this example, P2 matches the label, but we want P1 to change from –1 to 1.

Since we want to change P1 while leaving P2 alone, let's look at the part of the graph that tells us how the error will change by doing just that. We'll set P2=0 and look at the cross-section of the bowl for different values of P1. We can see it follows the overall bowl shape, as in Figure 18.15.

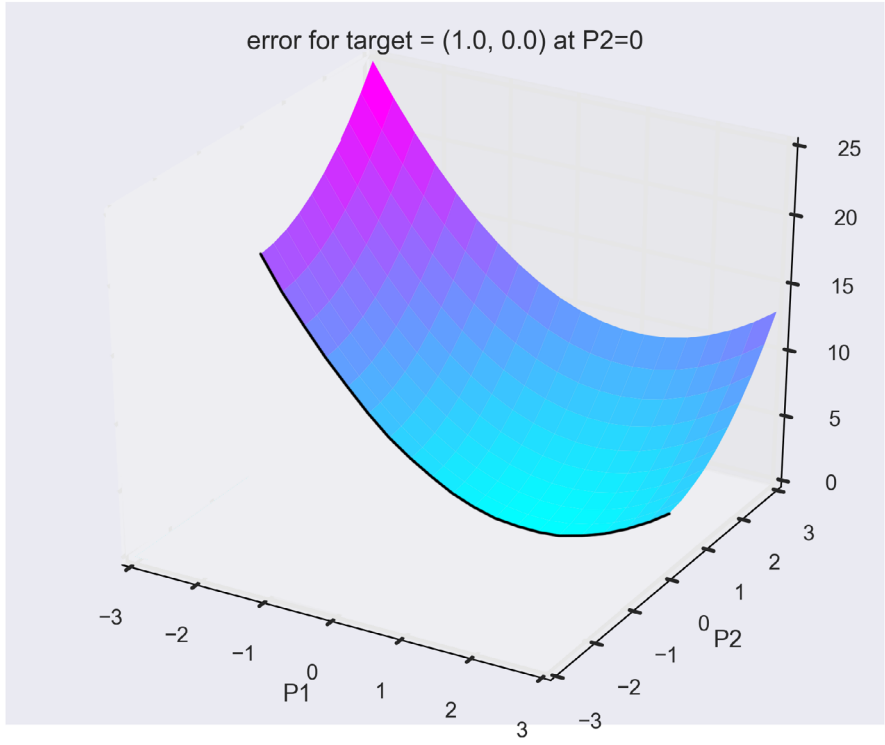

Figure 18.15: Slicing away the error surface from the bottom left of Figure 18.14, where the label is (1,0). The revealed cross-section of the surface shows us the values of the error for different values of P1 when P2 = 0.

Let's look just at this slice of the error surface, shown in Figure 18.16.

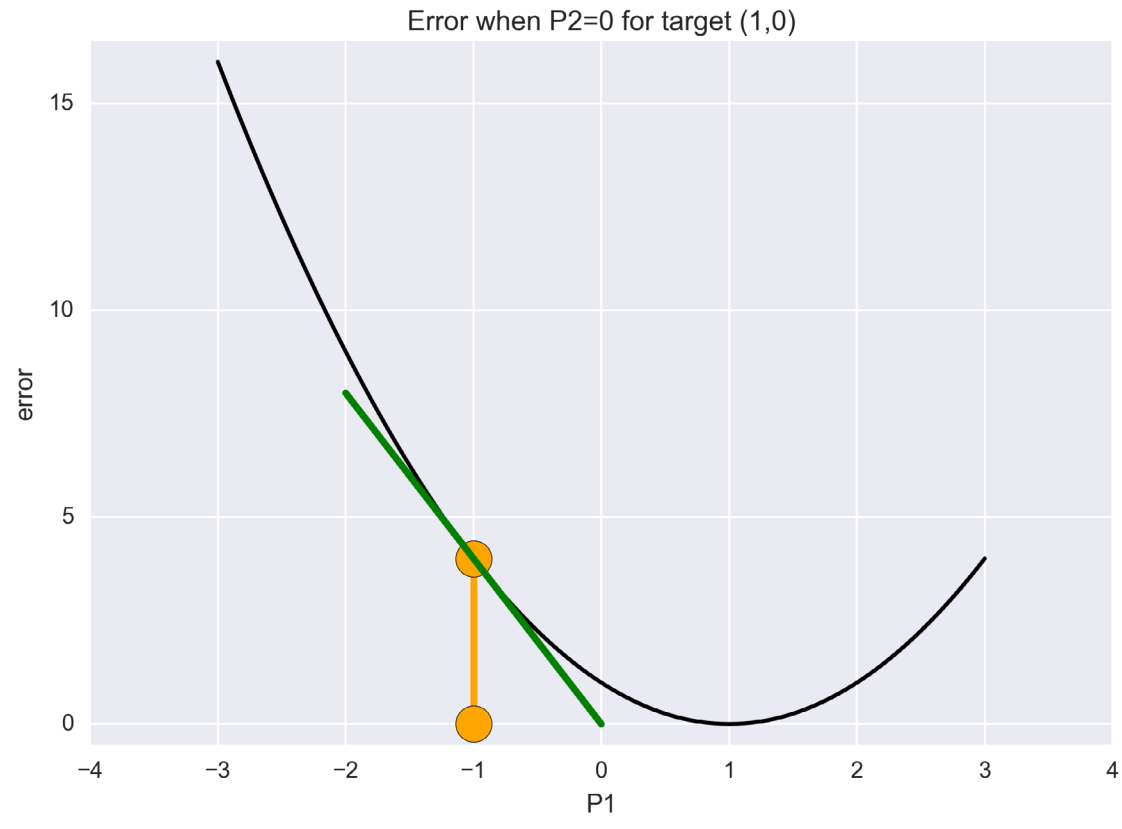

Figure 18.16: Looking at the cross-section of the error function shown in Figure 18.15, we can see how the error depends on different values of P1, when P2 is fixed at 0.

In Figure 18.16 we've marked the value P1= −1 with an orange dot, and we've drawn the derivative at the location on the curve directly above this value of P1. This tells us that if we make P1 more positive (that is, we move right from −1), the error in the network will decrease. But if we go too far and increase P1 beyond the value of 1, the error will start to increase again. The derivative is just the piece of the gradient that applies to only P1, and tells us how the error changes as the value of P1 changes, *for these values of P2 and the label*. As we can see from the figure, if we get too far away from −1 the derivative no longer matches the curve, but close to −1 it does a good job.

We'll come back to this idea again later: the derivative of a curve tells us what happens to the error if we move P1 *by a very small amount* from a given location. The smaller the move, the more accurate the derivative will be at predicting our new error. This is true for any derivative, or any gradient.

We can see this characteristic in Figure 18.16. If we move P1 by 1 unit to the right from −1, the derivative (in green) would land us at an error of 0, though it looks like the error for P1=0 (the value of the black curve) is really about 1. We *can* use the derivative to predict the results for large changes in P1, but our accuracy will go down the farther we move, as we just saw. In the interests of clear figures that are easy to read, we'll sometimes make large moves when the difference between where the derivative would land us, and where the real error curve tells us we should be, are close enough.

Let's use the derivative to predict the change in the error due to a change in P1. What's the slope of the green line in Figure 18.16? The left end is at about (−2, 8), and the right end is at about (0,0). Thus the line descends about 4 units for every 1 unit we move to the right, for a slope of −4/1 or −4. So if P1 changed by 0.5 (that is, it changed from −1 to −0.5), we'd predict that the error would go down by 0.5×−4=−2.

That's it! That's the key observation that tells us the value of $C\delta$.

Remember that P1 is just another name for *Co*, the output of C. We've found that a change of 1 in *Co* results in a change of −4 in the error. As we discussed, we shouldn't have too much confidence in this prediction after such a big change in P1. But for small moves, the proportion is right. For instance, if we were to increase P1 by 0.01, then we'd expect the error to change by −4×0.01=−0.04, and for such a small change in P1 the predicted change in the error should be pretty accurate. If we increased P1 by 0.02, then we'd expect the error to change by −4×0.02=−0.08. If we move P1 to the left, so it changed from −1 to, say, −1.1 we'd expect the error to change by −0.1×−4=0.4, so the error would increase by 0.4.

We've found that for any amount of change in *Co*, we can predict the change in the error by multiplying *Co* by −4.

That's exactly what we've been looking for! The value of $C\delta$ is −4. Note that this only holds for this label, and these values of *Co* and *Do* (or P1 and P2).

We've just found our first delta value, telling us how much the error will change if there's a change to the output of C. It's just the derivative of the error function measured at P1 (or *Co*).

Figure 18.17 shows what we've just described using our error diagram.

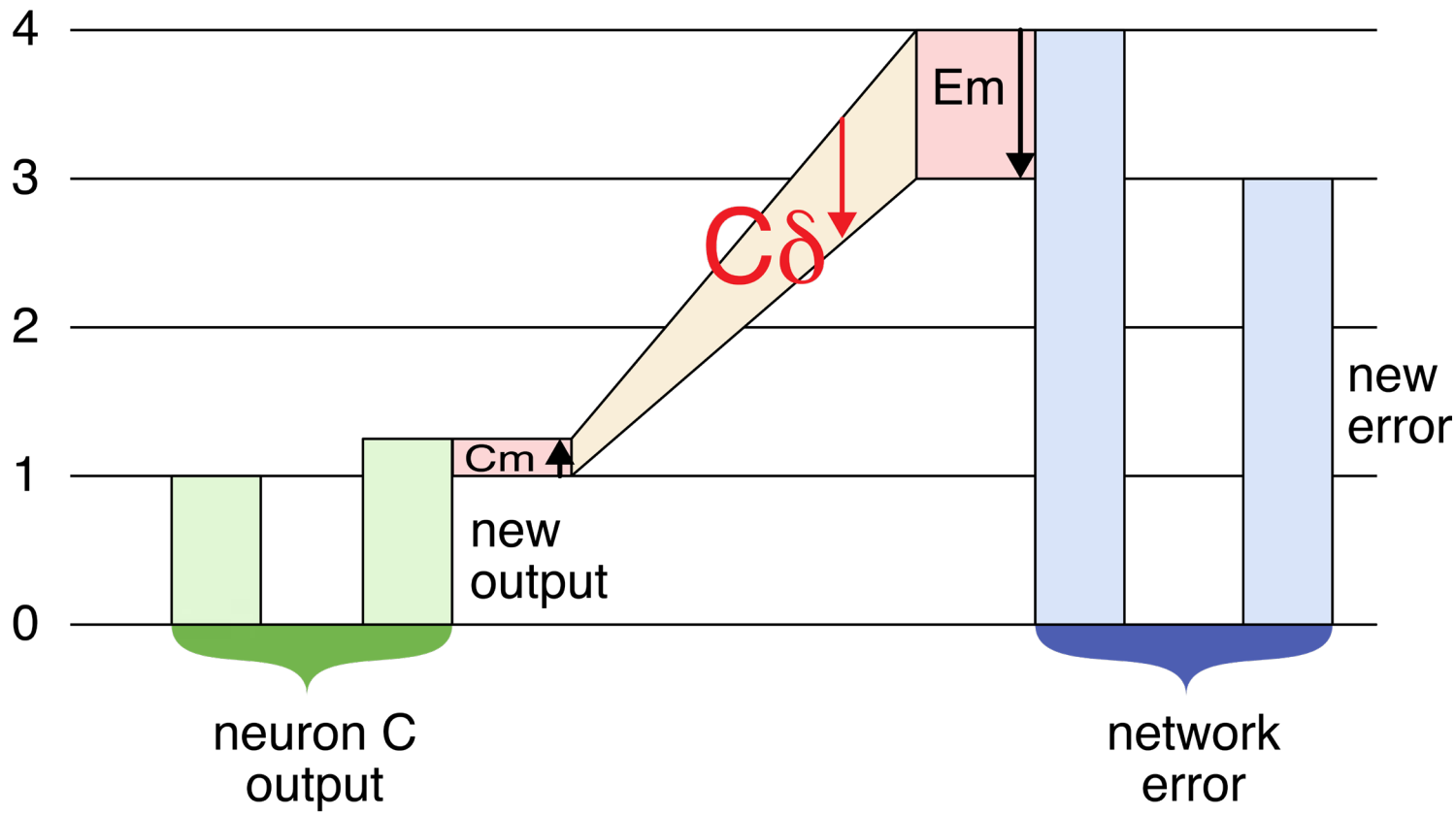

Figure 18.17: Our error diagram illustrating the change in the error from a change in the output of neuron C. The original output is the green bar at the far left. We imagine that due to a change in the inputs, the output of C increases by an amount *Cm*. This is amplified by multiplying it with *Cδ*, giving us the change in the error, *Em*. That is, $Em=Cm\times C\delta$. Here the value of *Cm* is about 1/4 (the upward arrow in the box for *Cm* tells us that the change is positive), and the value of *Cδ* is −4 (the arrow in that box tells us the value is negative). So $Em=-4\times 1/4=-1$. The new error, at the far right, is the previous error plus *Em*.

Remember that at this point we're not going to do anything with this delta value. Our goal right now is just to find the deltas for our neurons. We'll use them later.

We assumed above that P2 already had the right value, and we only needed to adjust P1. But what if they were both different than their corresponding label values?

Then we'd repeat this whole process for P2, to get the value of *Dδ*, or delta for neuron D. Let's do just that.

In Figure 18.18 we can see the slices of the error for an input with a corresponding label, or target, of (1,0) when P1 = −0.5, and P2 = 1.5.

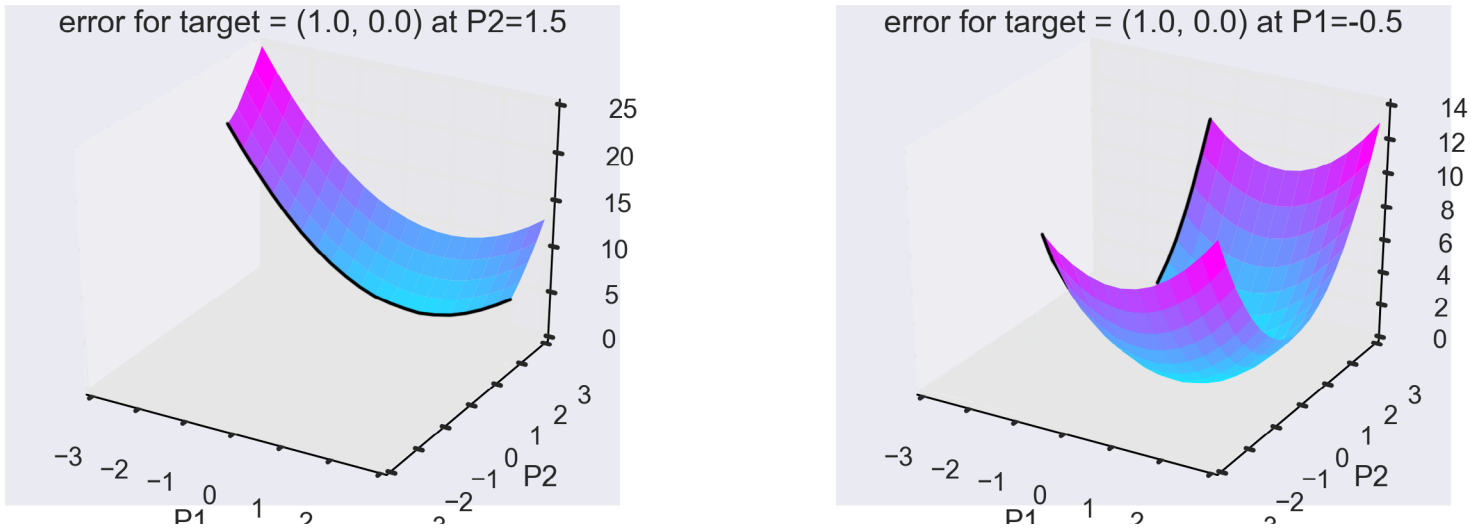

Figure 18.18: Slicing our error function when both P1 and P2 are different from the label. Left: The error due to different values of P1 when P2=1.5. Right: The error due to different values of P2 when P1 = −0.5.

The cross-section curves are shown in Figure 18.19. Notice that the curve for P1 has changed from Figure 18.16. That's because the change in P2 means we're taking the P1 cross-section at P2=1.5 instead of P2=0.

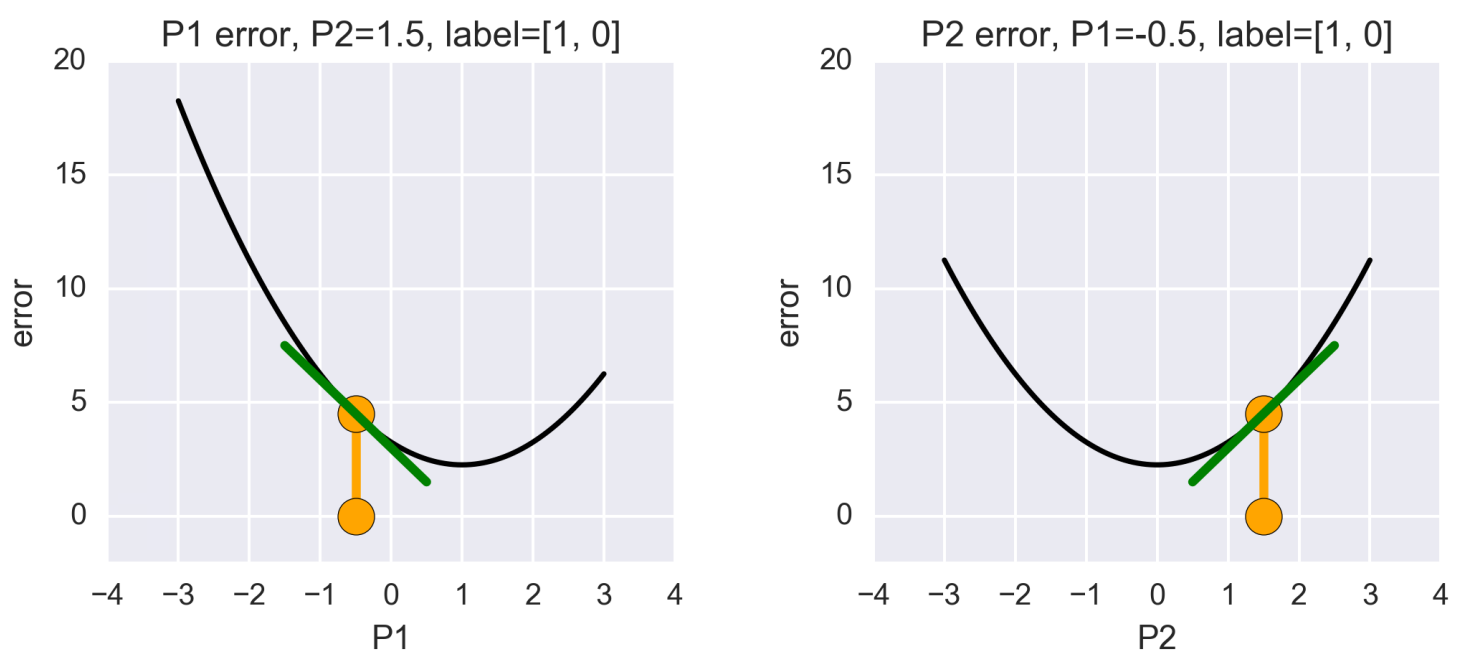

Figure 18.19: When neither P1 or P2 match the label, we can try to reduce the error by adjusting each one individually. In this example, the label is (1, 0). Left: The error for different values of P1 when P2 is 1.5. The smallest value is a little more than 2. The derivative tells us that we can get to that lower value by making P1 larger than its current value of −0.5 (that is, moving to the right). Right: The error for different values of P2 when P1 = −0.5. The current value of P2 is 1.5. The derivative tells us that we can make the error smaller by making P2 smaller (that is, moving to the left).

For the specific values in this figure, it looks like a change of about 0.5 in P1 would result in a change of about –1.5 in the error, so $C\delta$ is about –1.5/0.5 = –3. Instead of changing P1, what if we changed P2? Looking at the graph on the right, a change of about –0.5 (moving left this time, towards the minimum of the bowl) would result in a change of about –1.25 in the error, so $D\delta$ is about 1.25/-0.5 = 2.5. The positive sign here tells us that moving P2 to the right will cause the error to go up, so we want to move P2 to the left.

There are some interesting things to observe here. First, although both curves are bowl shaped, the bottoms of the bowls are at their respective label values. Second, because the current values of P1 and P2 are on opposite sides of the bottom of their respective bowls, their derivatives have opposite signs (one is positive, the other is negative).

The most important observation is that the minimum available error is not 0. In particular, the curves never get lower than a bit more than 2. That's because each curve looks at changing just one of the two values, while the other is left fixed. So even if P1 got to its ideal value of 1, there would still be error in the result because P2 is not at its ideal value of 0, and vice-versa.

This means that if we change just one of these two values, we'll never get down to the minimum error of 0. To get the error down to 0, both P1 and P2 have to work their way down to the bottom of their respective curves.

Here's the wrinkle: each time either P1 or P2 changes, we pick out a different cross-section of the error surface. That means we get different curves for the error, just as Figure 18.16, when P2=0, is different from Figure 18.19 when P2 = 1.5. Because the error curve has changed, the deltas change as well.

So if we change either value, we have to restart this whole process again from scratch before we know how to change the other value.

Well, not exactly. We'll see later that we actually *can* update both values at the same time, as long as we take very small steps. But that's about as far as we can cheat before we risk driving the error up again. Once we've taken our small steps, we have to evaluate the error surface again to find new curves and then new derivatives before we can adjust P1 and P2 again.

We've just described the general way to compute the delta values for the output neurons (we'll look at hidden neurons in a moment). In practice, we often use a particular measure of error that makes things easier, because we can write down a super simple formula for the derivative, which in turns gives us an easy way to find the deltas. This error measure is called the **quadratic cost function**, or the **mean squared error** (or **MSE**) [Neilsen15a]. As usual, we won't get into the mathematics of this equation. What matters for us now is that this choice of function for the error means that the derivative at any neuron's value (that is, the neuron's delta value) is particularly easy to calculate. The delta for an output neuron is the difference between the neuron's value and the corresponding label entry [Seung05]. Figure 18.20 shows the idea graphically.

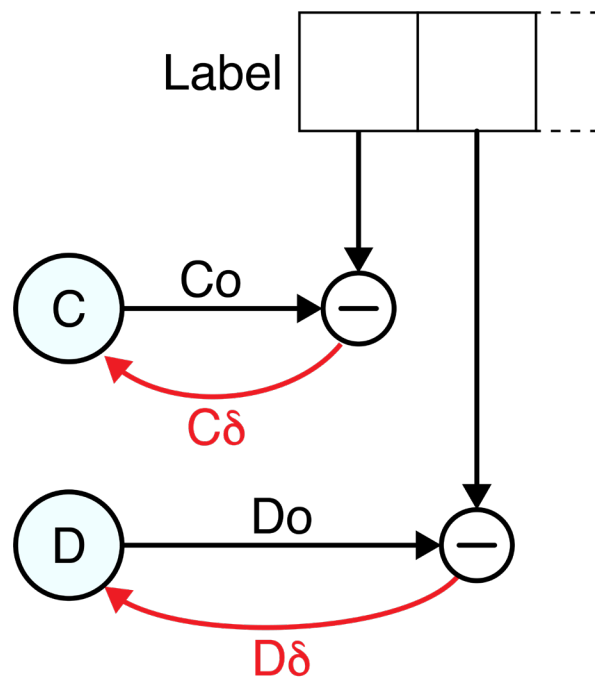

Figure 18.20: When we use the quadratic cost function, the delta for any output neuron is just the value in the label minus the output of that neuron. As shown in red, we save that delta value with its neuron.

This little calculation matches up nicely with our bowl-shaped pictures above. Remember that *Co* and P1 are two names for the same value, as are *Do* and P2.

Let's consider *Co* (or P1) when the first label is 1. If *Co*=1, then 1–*Co*=0, so a tiny change to *Co* will have no effect on the output error. That makes sense, because we're at the bottom of the bowl where it's flat.

Suppose that *Co*=2. Then the difference 1–*Co*=–1, telling us that a change to *Co* will change the error by the same amount, but with opposite sign (for example, a change of 0.2 in *Co* would result in a change of –0.2 to the error). If *Co* is much larger, say *Co*=5, then 1–*Co*=–4, telling us that any change to *Co* will be amplified by a factor of –4 in the change to the error. That also makes sense, because we're now at a very steep part of the bowl where a small change in the input (*Co*) will cause a big change in the output (the change in the error). We've been using large numbers for convenience, but remember that the derivative only tells us what happens if we take a very small step.

The same thought process holds for neuron D, and its output *Do* (or *P2*).

We've now completed the first step in backpropagation: we've found the delta values for all the neurons in the output layer.

We've seen that the delta for an output neuron depends on the value in the label and the neuron's output. If the neuron's output changes, the delta will change as well.

So "the delta" is a temporary value that changes with every new label, and every new output of the neuron. Following this observation, any time we update the weights in our network, we'll need to calculate new deltas.

Remember that our goal is to find the deltas for the weights. When we know the deltas for all the neurons in a layer, we can update all the weights feeding into that layer.

Let's see how that's done.

18.7 Step 2: Using Deltas to Change Weights

We've seen how to find a delta value for every neuron in the output layer. If the output of any of those neurons changes by some amount, we multiply that change by the neuron's delta to find the change to the output.

We know that a change to the neuron's output can come from a change in an input, which in turn can come from a change in a previous neuron's output or the weight connecting that output to this neuron. Let's look at these inputs.

We'll focus on output neuron C, and the value it receives from neuron A on the previous layer. Let's use the temporarily name *V* to refer to the value that arrives into C from A. It has a value given by the output of A, or *Ao*, and the weight between A and C, or *AC*, so the value *V* is *Ao*×*AC*. This setup is shown in Figure 18.21.

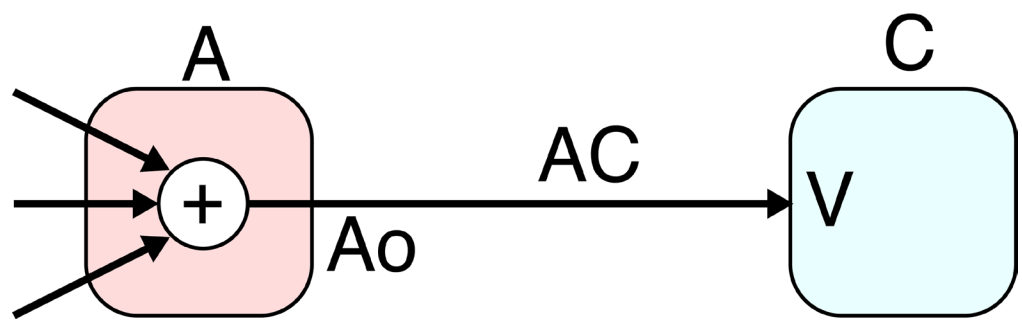

Figure 18.21: The value coming into neuron C, which we're calling V for the moment, is *Ao* (the output of A) times the weight *AC*, or *Ao*×*AC*.

If *V* changes for any reason, we know that the output of C will change by *V* as well (since C is just adding up its inputs and passing on that sum). Since the change in C is *V*, the network's error will change by *V*×*Cδ*, since we built *Cδ* to do just that.

There are only two ways the value of *V* can change in this setup: either the output of A changes or the value of the weight changes.

Since the output of A is computed by the neuron automatically, there's not much we can do to adjust that value directly. But we can change the weight, so let's look at that.

Let's modify the weight *AC* by adding some new value to it. We'll call that new value *ACm*, so the new value arriving at C is $Ao\times(AC+ACm)$. This is shown in Figure 18.22.

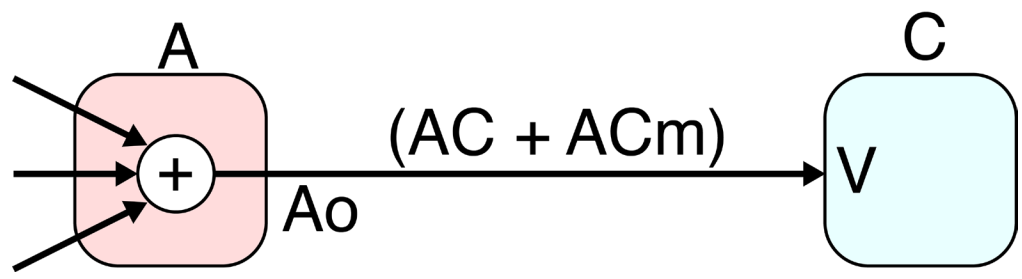

Figure 18.22: If we change the weight *AC*, then this will change the value of *V*. Here, the value of *V* coming into neuron C is $Ao\times(AC+ACm)$.

If we subtract the old value $Ao\times AC$ from the new value $Ao\times(AC+ACm)$, we find that the change in the value coming into C due to our modifying the weight is $Ao\times ACm$.

Since this change goes into C, the change will then get multiplied by $C\delta$, telling us the change in the network's error as a result of modifying the weight.

Hold on a second. That's what we've wanted this whole time, to find out what a change in the weight would do to the error. Have we just found that?

Yup, we have. We've achieved our goal! We discovered how a change to a weight would affect the network's error.

Let's look at that again.

If we change the weight *AC* by adding *ACm* to it, then we know the change in the network's error is given by the change in C, $(Ao\times ACm)$, times the delta for C, or $C\delta$. That is, the change in error is $(Ao\times ACm)\times C\delta$.

Let's suppose we increase the weight by 1. Then $ACm=1$, so the error will change by $Ao\times C\delta$.

So every change of +1 to the weight *AC* will cause a change of *Ao*×*Cδ* to the network error. If we add 2 to the weight *AC*, we'll get double this change, or 2×(*Ao*×*Cδ*). If we add 0.5 to the weight, we'll get half as much, or 0.5×(*Ao*×*Cδ*).

We can turn this around, and say that if we want to increase the error by 1, we should add 1/(*Ao*×*Cδ*) to the weight. But we want to make the error go down. The same logic says that we can reduce the error by 1 if we instead *subtract* the value 1/(*Ao*×*Cδ*) from the weight. So now we know how to change this weight in order to reduce the overall error.

We've found what to do to the weight *AC* to decrease the error in this little network. To subtract *Ao*×*Cδ* from the error, we subtract 1 from *AC*.

Figure 18.23 shows that every change of 1 in the weight *AC* leads to a corresponding change of *Ao*×*Cδ* in the network error in our network. And subtracting that value leads to a change of –*Ao*×*Cδ* in the error.

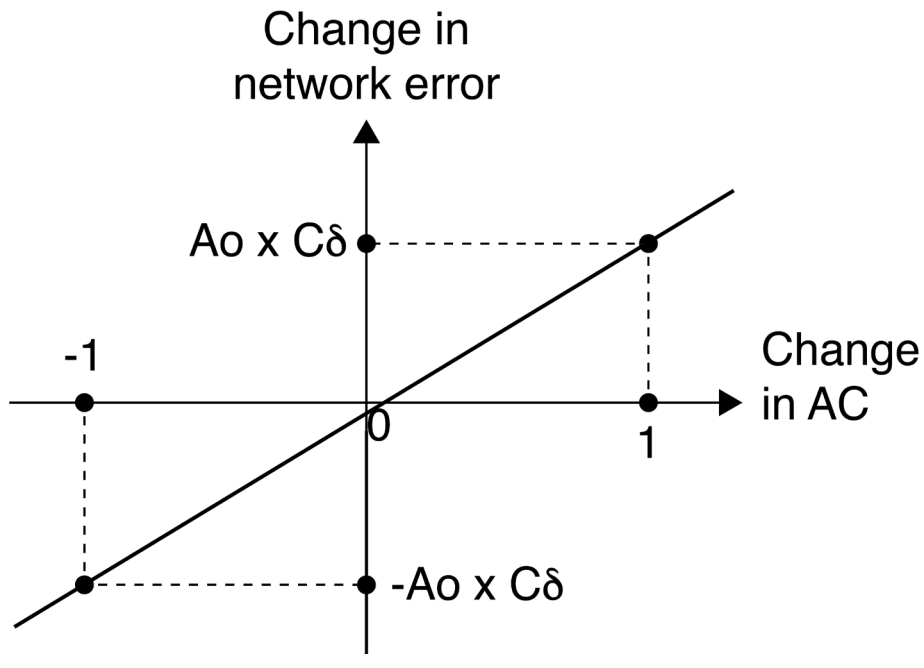

Figure 18.23: For each change in the value of the weight *AC*, we can look up the corresponding change in our network's error. When *AC* changes by 1, the network error changes by *Ao*×*Cδ*.

We can summarize this process visually with a new convention for our diagrams. We've been drawing the outputs of neurons as arrows coming out of a circle to the right. Let's draw deltas using arrows coming out of the circles to the left, as in Figure 18.24.

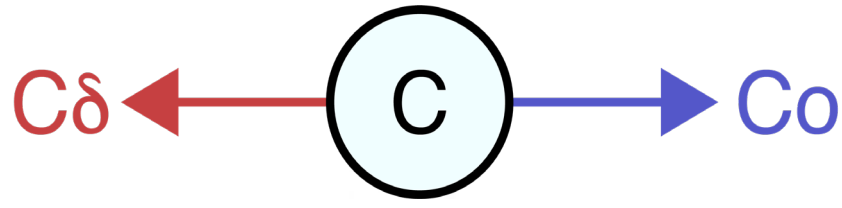

Figure 18.24: Neuron C has an output *Co*, drawn with an arrow pointing right, and a delta *Cδ*, drawn with an arrow pointing left.

With this convention, the whole process for finding the updated value for a weight is summarized in Figure 18.25. Showing subtraction in a diagram like this is hard, because if we have a "minus" node with two incoming arrows, it's not clear which value is being subtracted from the other (that is, if the inputs are x and y, are we computing $x-y$ or $y-x$?). Our approach to computing $AC-(Ao \times C\delta)$ is to find $Ao \times C\delta$, multiply that by -1, and then add that result to AC.

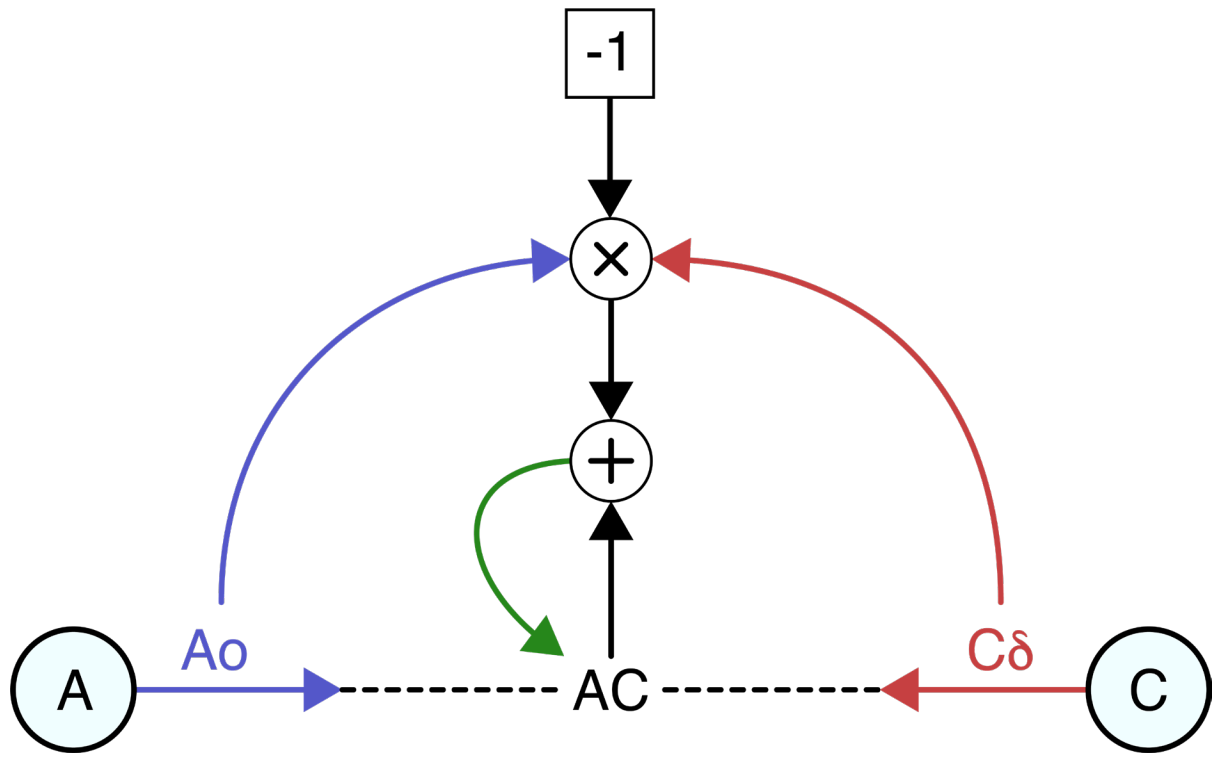

Figure 18.25: Updating the value of weight *AC*. We start with the output *Ao* from neuron A and the delta *Cδ* from neuron C, and multiply them together. We'd like to subtract this from the current value of *AC*. To show this clearly in the diagram, we instead multiply the product by −1, and then add it to *AC*. The green arrow is the update step, where this result becomes the new value of *AC*.

Figure 18.25 is the goal of this chapter, and the reason we computed *Cδ*.

This is how our network learns.

Figure 18.25 tells us how to change the weight *AC* to bring down the error. The diagram says that to reduce the error, we should add $-Ao \times C\delta$ to the value of *AC*.

Success!

If we change the weights for both output neurons C and D to reduce the error by 1 from each neuron, we'd expect the error to go down by −2. We can predict this because the neurons sharing the same layer don't rely on each other's outputs. Since C and D are both in the output layer, C doesn't depend on *Do* and D doesn't depend on *Co*. They do depend on the outputs of neurons on previous layers, but right now we're just focusing on the effect of changing weights for C and D.

It's wonderful that we know how to adjust the last two weights in the network, but how about all the other weights? To use this technique, we need to figure out the deltas for all the neurons in all the remaining layers. Then we can use Figure 18.25 to improve all the weights in the network.

And this brings us to the remarkable trick of backpropagation: we can use the neuron deltas at one layer to find all the neuron deltas for its preceding layer. And as we've just seen, knowing the neuron deltas and the neuron outputs tells us how to update all the weights coming into that neuron.

Let's see how to do that.

18.8 Step 3: Other Neuron Deltas

Now that we have the delta values for the output neurons, we will use them to compute the deltas for neurons on the layer just before the output layer. In our simple model, that's just neurons A and B. Let's focus for the moment just on neuron A, and its connection to neuron C.

What happens if *Ao*, the output of A, changes for some reason? Let's say it goes up by *Am*. Figure 18.26 follows the chain of actions from this change in *Ao* to the change in *Co* to the change in the error.

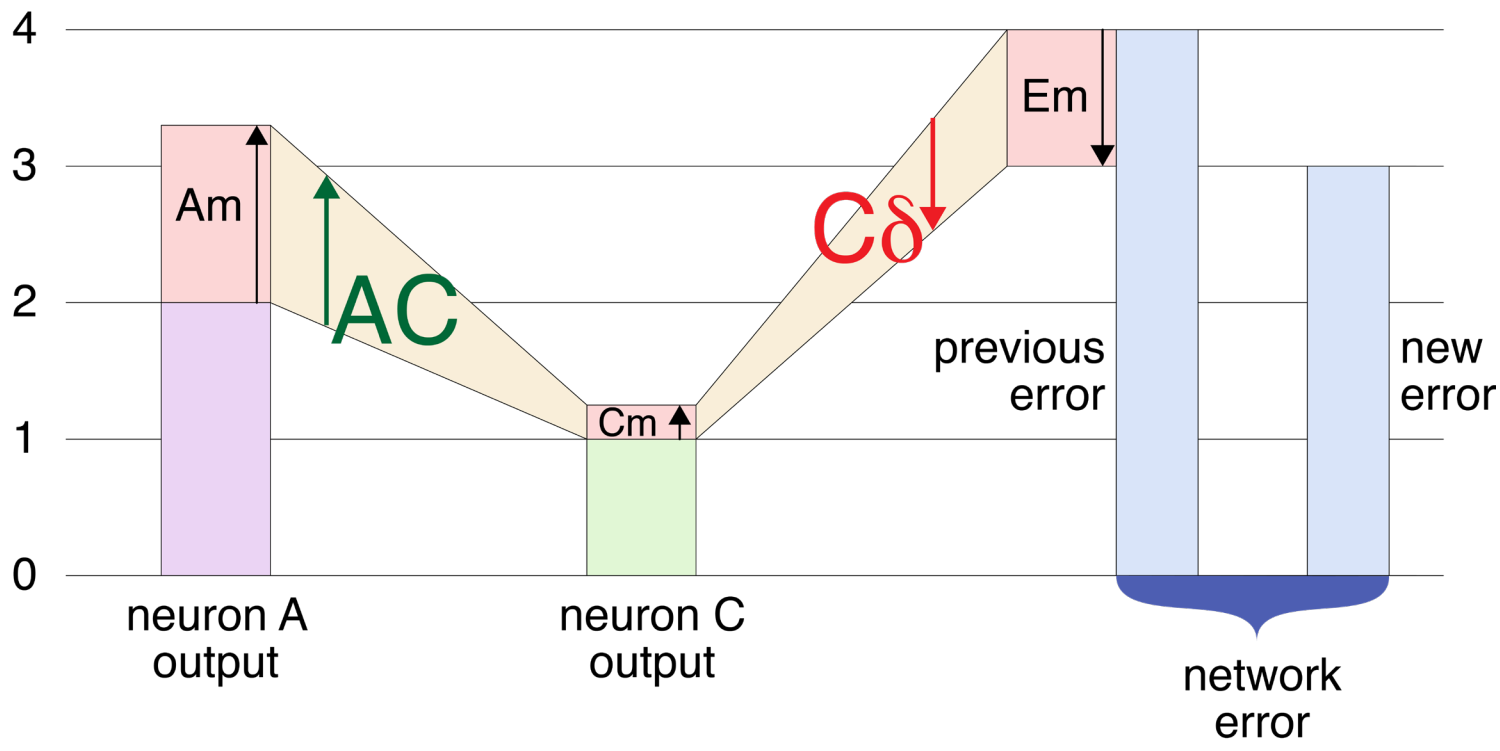

Figure 18.26: Following the results if we change the output of neuron A. Read the diagram left to right. The change to A, shown as *Am*, is multiplied by the weight AC and is added to the values accumulated by neuron C. This raises the output of C by *Cm*. As we know, this change in C can be multiplied by *Cδ* to find the change in the network error. In this example, *Am*=5/4, and *AC*=1/5, so *Cm*=5/4×1/5=1/4. The value of *Cδ* is −4, so the change in the error is 1/4×−4=−1.

We know that the neuron C adds together its weighted inputs and then passes on the sum (since we're ignoring the activation function for now). So if nothing else changes in C except for the value coming from A, that's the only source of any change in *Co*, the output of C. We represent that change to *Co* as the value *Cm*. As we saw before, we can predict the change in the network's error by multiplying *Cm* by *Cδ*.

So now we have a chain of operations from neuron A to neuron C and then to the error. The first step of the chain says that if we multiply the change in *Ao* (that is, *Am*) by the weight *AC*, we'll get *Cm*, the change in in the output of C. And we know from above that if we multiply *Cm* by *Cδ*, we get the change in the error.

So mushing this all together, we find that the error due to a change *Am* in the output of A is *Am*×*AC*×*Cδ*.

In other words, if we multiply the change in A (that is, Am) by $AC \times C\delta$, we get the change in the error due to the change in neuron A. That is, $A\delta = AC \times C\delta$.

We just identified the delta of A! Since the delta is the value that we multiply a neuron's change by to find the change in the error, and we just found that value is $AC \times C\delta$, then we've found $A\delta$.

Figure 18.27 shows this visually.

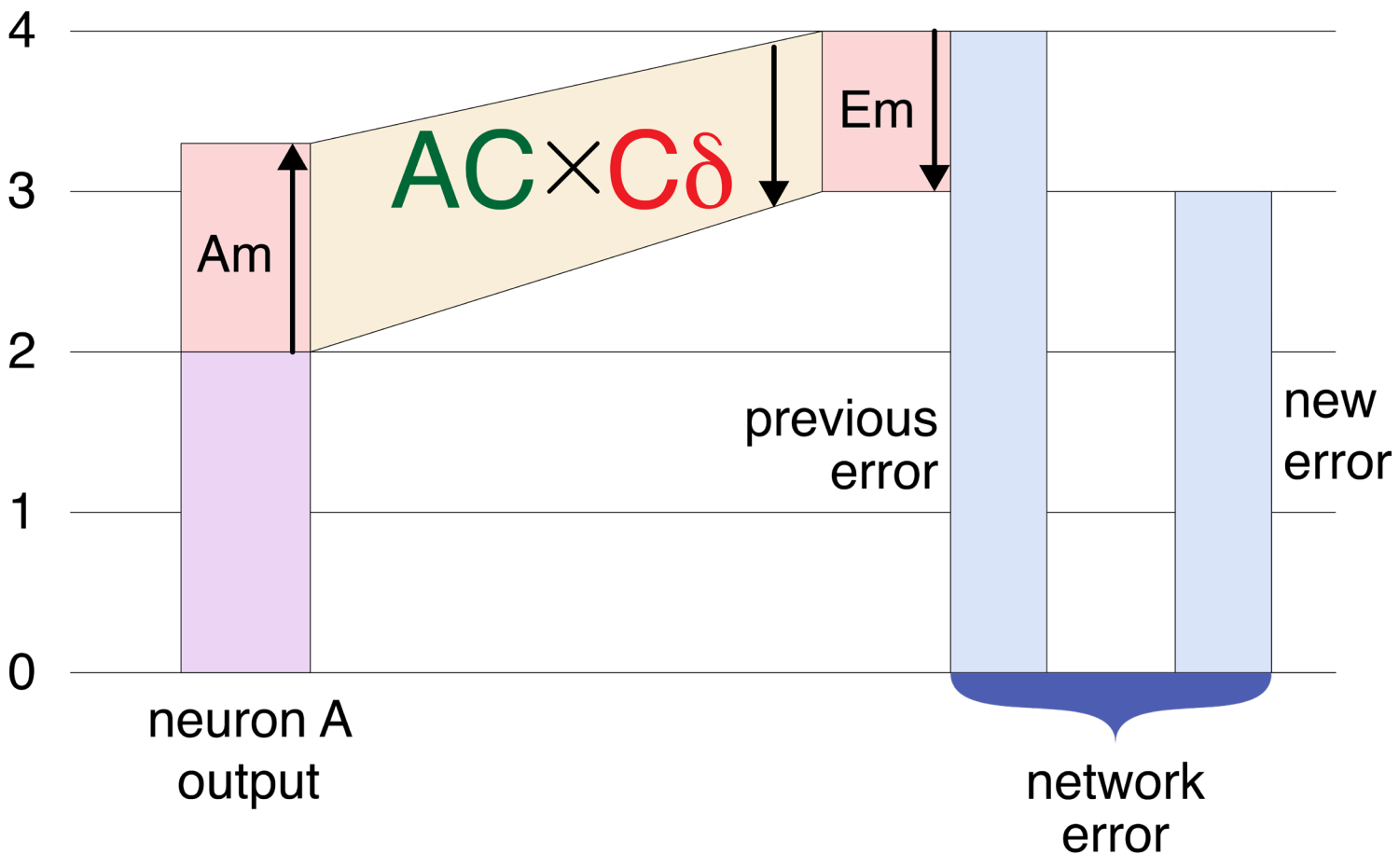

Figure 18.27: We can mush together the operations in Figure 18.26 into a more succinct diagram. In that figure, we saw that Am, the change in A, gets multiplied by the weight AC and then the value $C\delta$. So we can represent that as a single step, where we multiply Am by the two values AC and $C\delta$ multiplied together. As before, the value of Am=5/4, AC=1/5, and $C\delta$ is −4. So $AC \times C\delta$=−4/5, and multiplying that by Am=5/4 gives us a change in the error of −1.0.

This is kind of amazing. Neuron C has disappeared. It's literally out of the picture in Figure 18.27. All we needed was its delta, $C\delta$, and from that we could find $A\delta$, the delta for A. And now that we know $A\delta$, we can update all of the weights that feed into neuron A, and then... no, wait a second.

We don't really have $A\delta$ yet. We just have one piece of it.

At the start of this discussion we said we'd focus on neurons A and C, and that was fine. But if we now remember the rest of the network, we can see that neuron D also uses the output of A. If Ao changes due to Am, then the output of D will change as well, and that will also have an effect on the error.

To find the change in the error due to neuron D, caused by a change to neuron A, we can repeat the process above, just replacing neuron C with neuron D. So if Ao changes by Am, and nothing else changes, the change in the error due to the change in D is given by $AC \times D\delta$.

Figure 18.28 shows these two paths at the same time. This figure is set up slightly differently from the ones above. Here, the effect of a change in A on the error due to a change in C is shown by the path from the center of the diagram moving to the right. The effect of a change in A on the error due to a change in D is shown by the path from the center of the diagram and moving left.

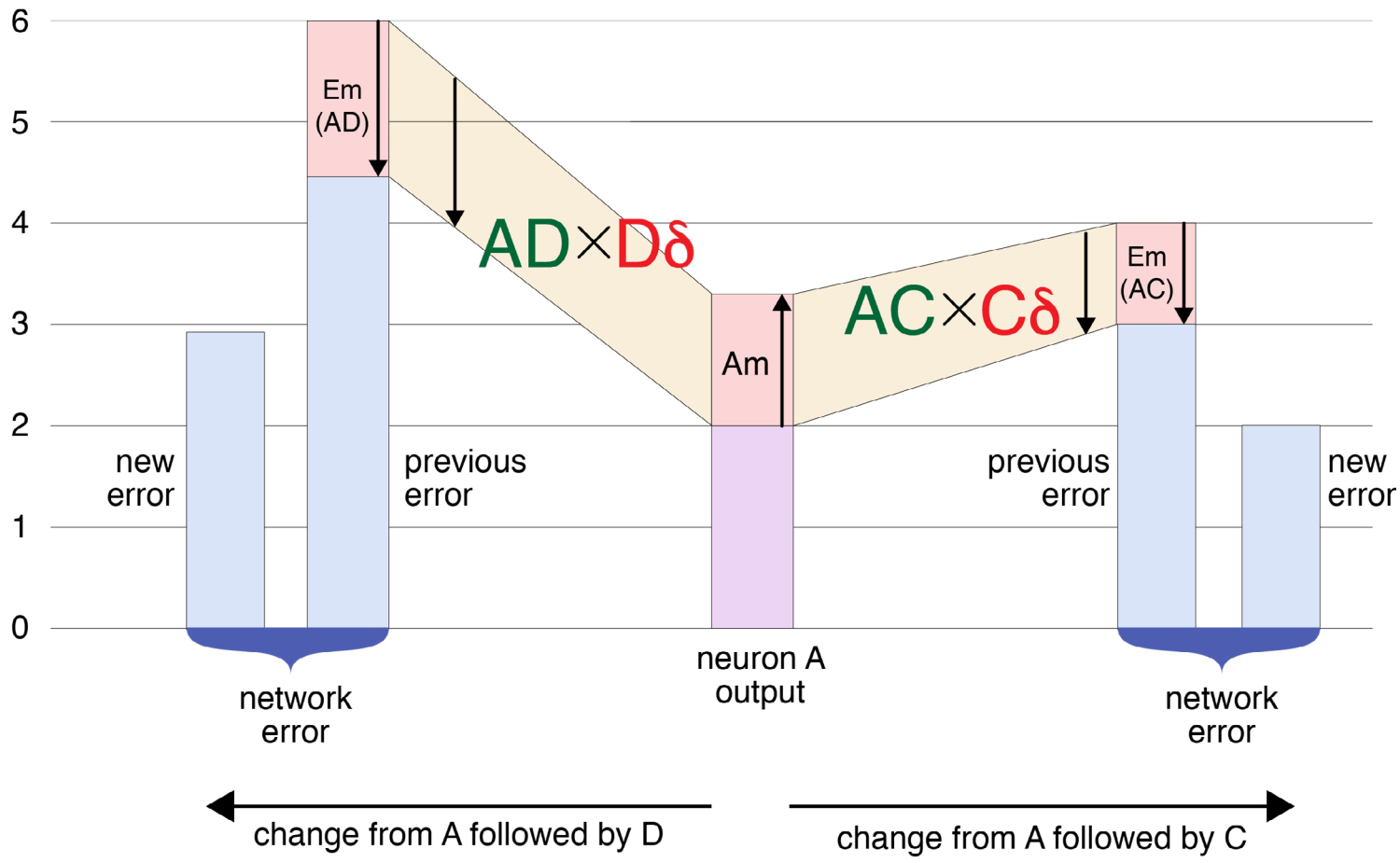

Figure 18.28: The output of neuron A is used by both neuron C and neuron D. In this figure, we've changed our left-to-right convention to show the same change in A, given by Am, affecting the final error by way of two different paths, one to the left and one to the right, each starting at neuron A in the center. The path through neuron C is shown going to the right, where *Am*, the change in the output of A, is scaled by $AC \times C\delta$ to get a change in the error labeled *Em*(*AC*). Moving from the center to the left, *Am* is scaled by $AD \times D\delta$ to get another change to the error, labeled *Em*(*AD*). The result is two separate changes to the error.

Figure 18.28 shows two separate changes to the error. Since neurons C and D don't influence each other, so their effects on the error are independent. To find the total change to the error, we just add up the two changes. Figure 18.29 shows the result of adding the change in error via neuron C and the change via neuron D.

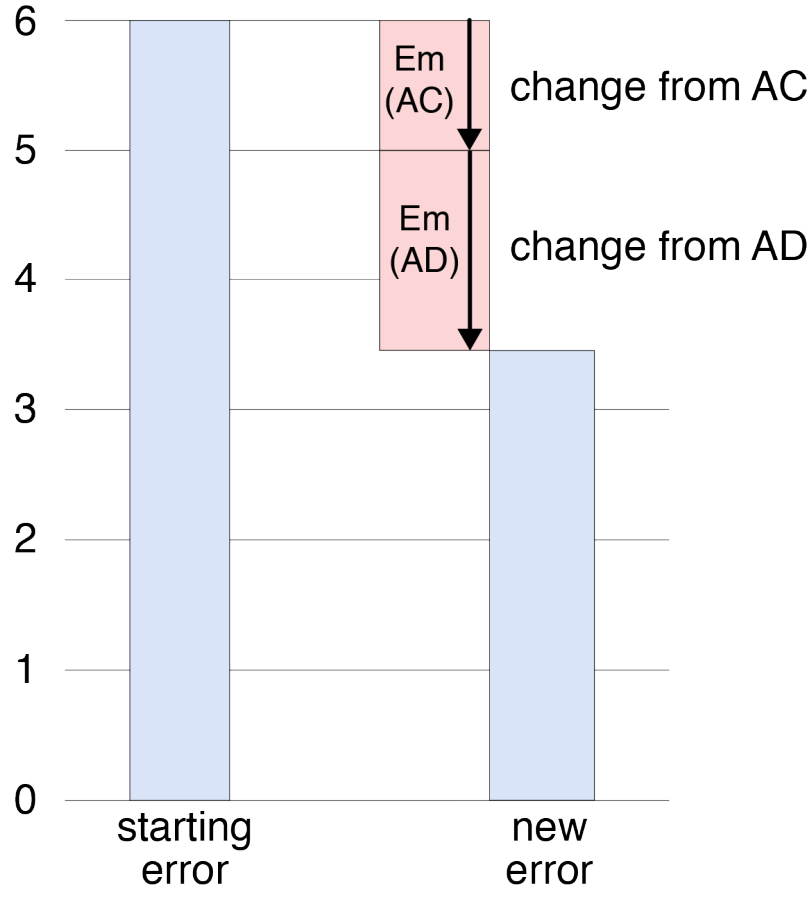

Figure 18.29: When the output of neuron A is used by both neuron C and neuron D, the resulting changes to the error add together.

Now that we've handled all the paths from A to the outputs, we can finally write the value for $A\delta$. Since the errors add together, as in Figure 18.29, we can just add up the factors that scale Am. If we write it out, this is $A\delta=(AC\times C\delta)+(AD\times D\delta)$.

Now that we've found the value of delta for neuron A, we can repeat the process for neuron B to find its delta.

We've actually done something far better than find the delta for just neurons A and B. We've found out how to get the value of delta for *every* neuron in *any* network, no matter how many layers it has or now many neurons there are!

That's because everything we've done involves nothing more than a neuron, the deltas of all the neurons in the next layer that use its value as an input, and the weights that join them. With nothing more than that, we can find the effect of a neuron's change on the network's error, even if the output layer is dozens of layers away.

To summarize this visually, let's expand on our convention for drawing outputs and deltas as right-pointing and left-pointing arrows to include the weights, as in Figure 18.30. We'll say that the weight on a connection multiplies either the output moving to the right, or the delta moving to the left, depending on which step we're thinking about.

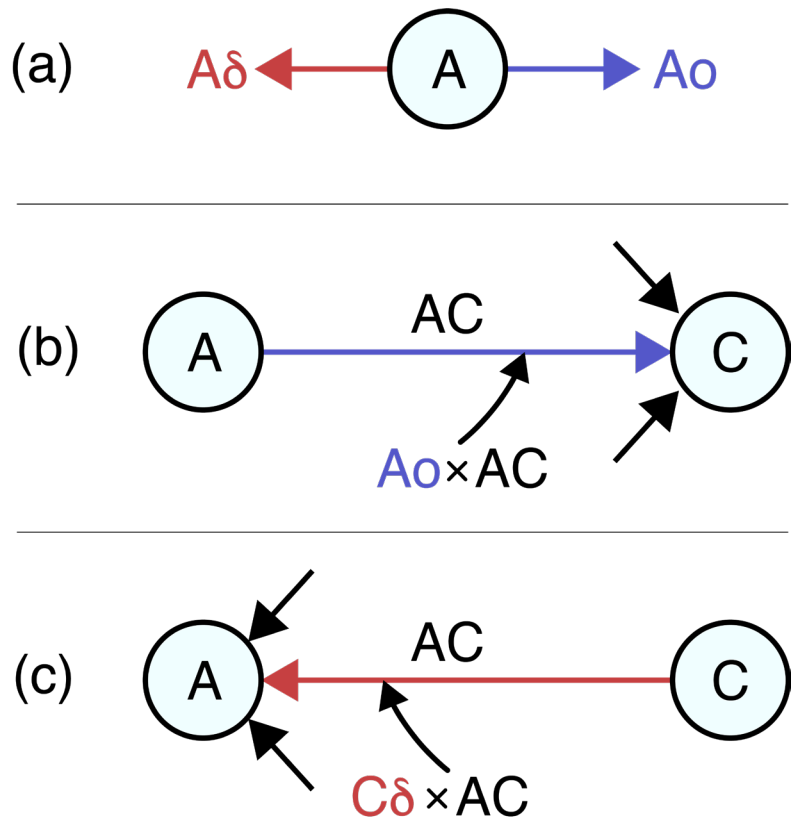

Figure 18.30: Drawing the values associated with neuron A. (a) Our convention is to draw the output *Ao* as an arrow coming out of the right of the neuron, and the delta *Aδ* as an arrow coming out of the left. (b) The output of A is multiplied by *AC* on its way to being used by C when we're evaluating a sample. (c) The delta of C is multiplied by *AC* on its to being used by A when we're computing delta values.

In other words, there is *one connection* with *one weight* joining neurons A and C. If the arrow points to the right, then the weight multiplies *Ao*, the output of A as it heads into neuron C. If the arrow points to the left, the weight multiplies *Cδ*, the delta of C, as it heads into neuron A.

When we evaluate a sample, we use the feed-forward, left-to-right style of drawing, where the output value from neuron A to neuron C travels over a connection with weight *AC*. The result is that the value *Ao*×*AC* arrives at neuron C where it's added to other incoming values, as in Figure 18.30(b).

When we later want to compute *Aδ*, we draw the flow from right-to-left. Then the delta leaving neuron C travels over a connection with weight *AC*. The result is that the value *Cδ*×*AC* arrives at neuron A where it's added to other incoming values, as in Figure 18.30(c).

Now we can summarize both the processing of a sample input, and the computation of the deltas, in Figure 18.31.

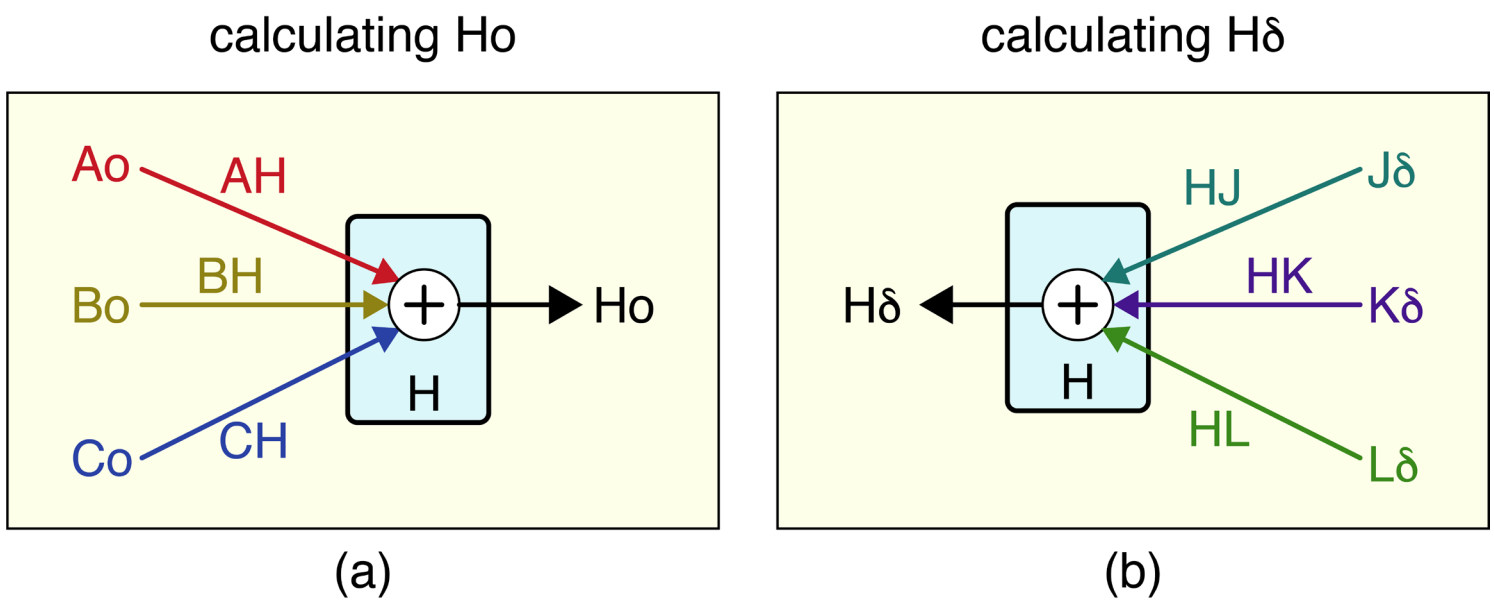

Figure 18.31: Calculating the output and delta for neuron H. Left: To calculate *Ho*, we scale the output of each preceding neuron by the weight of its connection and add the results together. Right: To calculate *Hδ*, we scale the delta of each following neuron by the connection's weight and add the results together.

This is pleasingly symmetrical. It also reveals an important practical result: when a neuron is connected to the same number of preceding and following neurons, calculating the delta for a neuron takes the same amount of work (and therefore the same amount of time) as calculating its output. So calculating deltas is as efficient as calculating output values. Even when the input and output counts are different, the amount of work involved is still close in both directions.

Note that Figure 18.31 doesn't require anything of neuron H except that it has inputs from a preceding layer that travel on connections with weights, and deltas from a following layer that travel on connections with weights. So we can apply the left half of Figure 18.31 and calculate the output of neuron H as soon as the outputs from the previous layer are available. And we can apply the right half of Figure 18.31 and calculate the delta of neuron H as soon as the deltas from the following layer are available.

This also tells us why we had to treat the output layer neurons as a special case: there are no "next layer" deltas to be used.

This process of finding the delta for every neuron in the network *is* the backpropagation algorithm.

18.9 Backprop in Action

In the last section we saw the backpropagation algorithm, which lets us compute the delta for every neuron that in a network.

Because that calculation depended on the deltas in the following neurons, and the output neurons don't have any of those, we had to treat the output neurons as a special case.

Once all the neuron deltas for any layer (including the output layer) have been found, we can then step back one layer (towards the inputs), and find the deltas all the neurons on that layer, and then step back again, compute all the deltas, step back again, and so on until we reach the input.

Let's walk through the process of using backprop to find the deltas for all the neurons in a slightly larger network.

In Figure 18.32 we show a new network with four layers. There are still two inputs and outputs, but now we have 3 hidden layers of 2, 4, and 3 neurons.

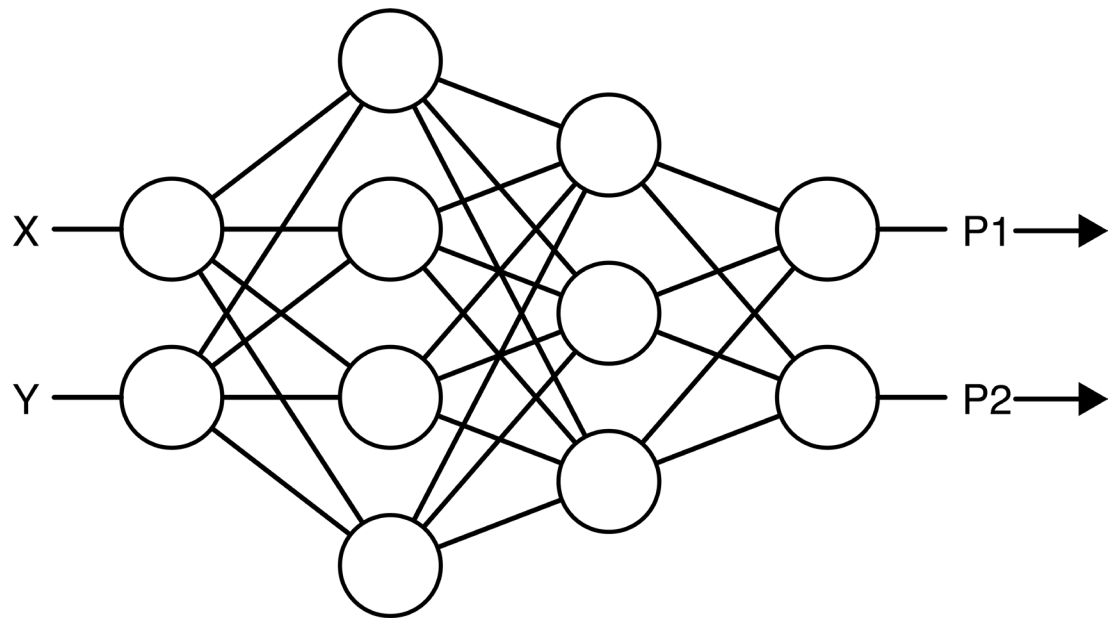

Figure 18.32: A new classifier network with 2 inputs, 2 outputs, and 3 hidden layers.

We start things off by evaluating a sample. We provide the values of its X and Y features to the inputs, and eventually the network produces the output predictions P1 and P2.

Now we'll start backpropagation by finding the error in the output neurons, as shown in Figure 18.33.

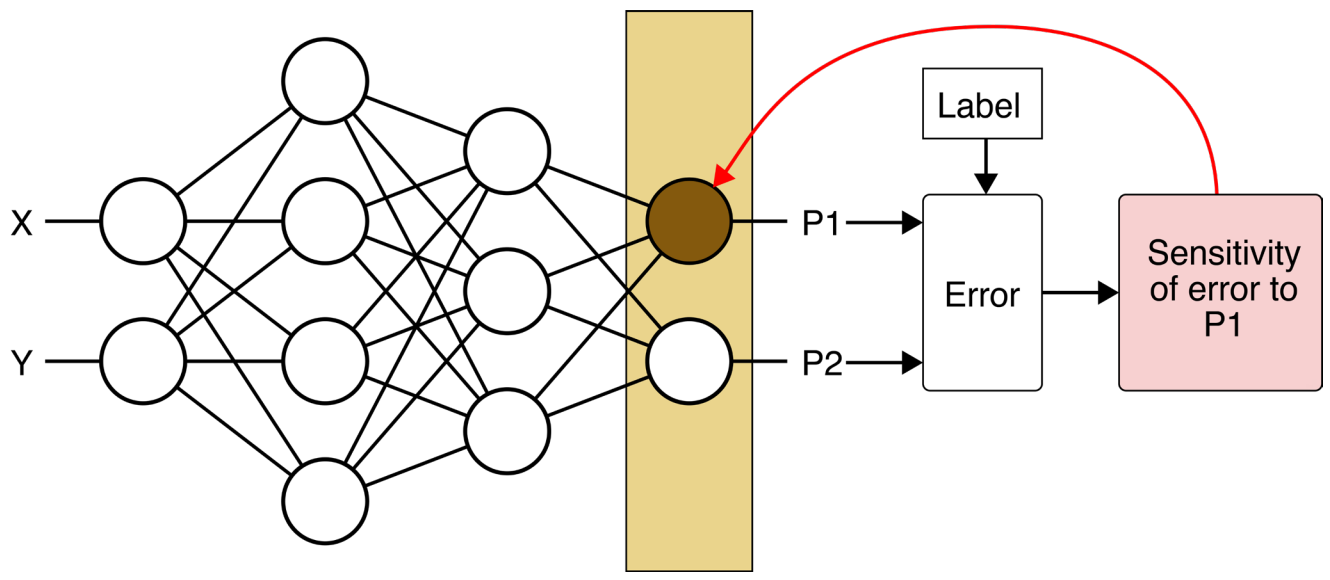

Figure 18.33: Computing the delta of the first output neuron in a new network. Using the general approach, we take the outputs of the output layer (here called P1 and P2) and compare them to the label to derive an error. From the outputs, the label, and the error value we find how much a change in P1 would change the error. That value is the delta stored at the neuron that produces P1.

We've begun arbitrarily with the upper neuron, which gives us the prediction we've labeled P1 (the likelihood that the sample is in class 1). From the values of P1 and P2 and the label, we can compute the error in the network's output. Let's suppose the network didn't get this sample perfectly predicted, so the error is greater than zero.

Using the error, the label, and the values of P1 and P2, we can compute the value of delta for this neuron. If we're using the quadratic cost function, this delta is just the value of the label minus the value of the neuron, as we saw in Figure 18.20. But if we're using some other function, it might be more complicated, so we've illustrated the general case.

Once we've computed the value of delta for this neuron, we store it with the neuron, and we're done with that neuron for now.

We'll repeat this process for all the other neurons in the output layer (here we have only one more). That finishes up the output layer, since we now have a delta for every neuron in the layer. Figure 18.34 summarizes these two neurons getting their deltas.

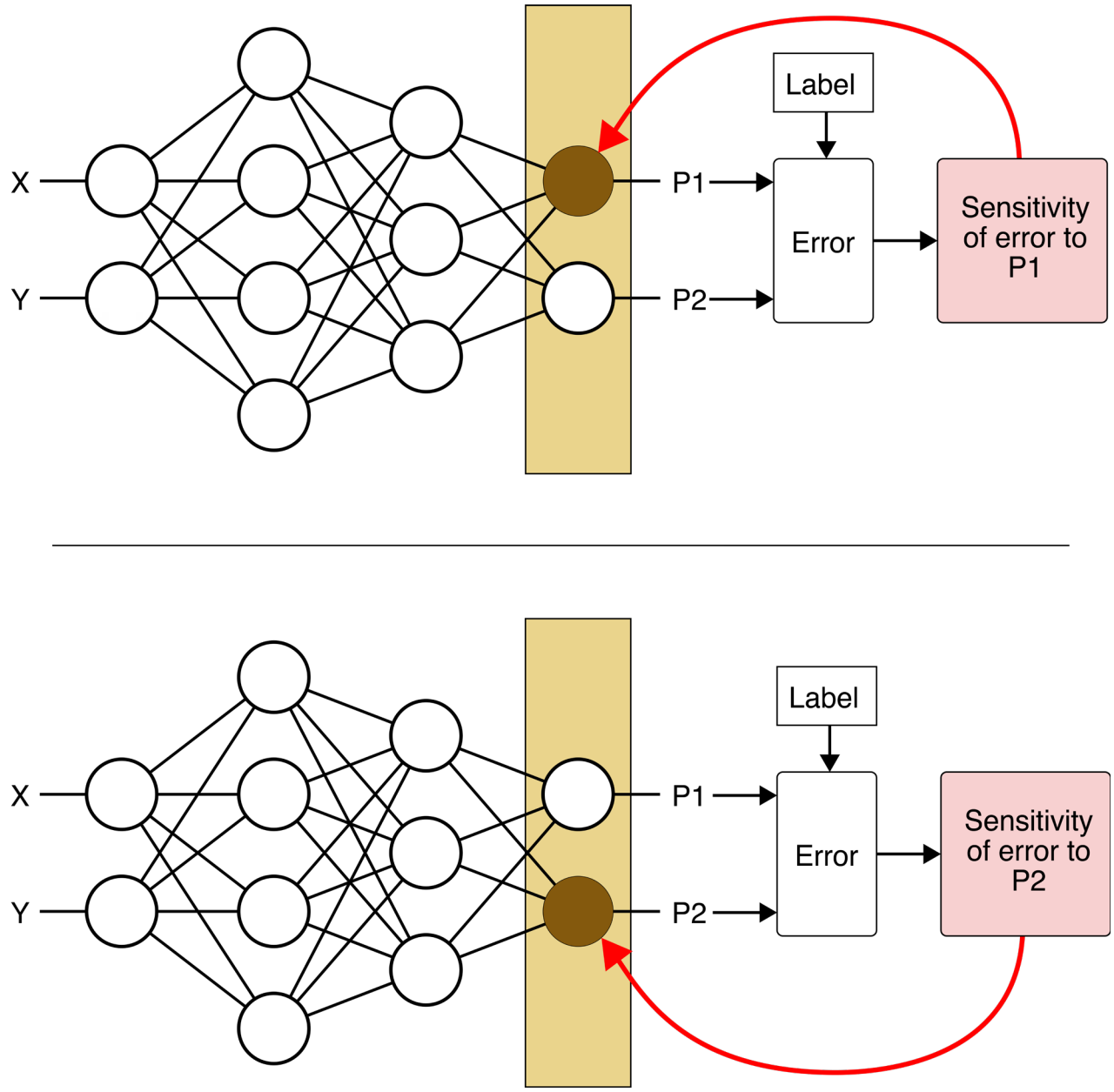

Figure 18.34: Summarizing the steps for finding the delta for both output neurons.

At this point we could start adjusting the weights coming into the output layer, but we usually break things up by first finding all the neuron deltas, and then adjusting all the weights. Let's follow that typical sequence here.

So we'll move backwards one step to the third hidden layer (the one with 3 neurons). Let's consider finding the value of delta for the topmost of these three, as in the left image of Figure 18.35.

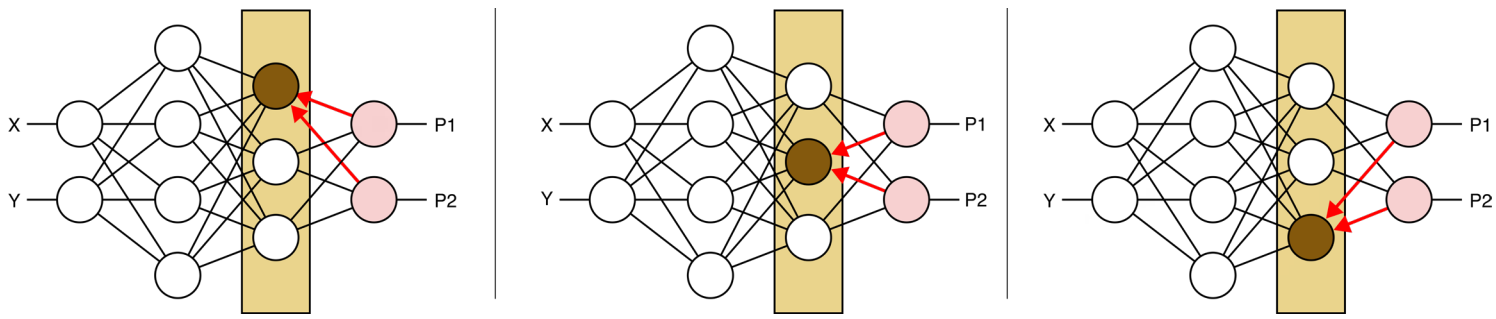

Figure 18.35: Using backpropagation to find the deltas for the next-to-last layer of neurons. To find the delta for each neuron, we find the deltas of the neurons that use its output, multiply those deltas by the corresponding weights, and add the results together.

To find the delta for this neuron, we follow the recipe of Figure 18.28 to get the individual contributions, and then the recipe of Figure 18.29 to add them together to get the delta for this neuron.

Now we just work our way through the layer, applying the same process to each neuron. When we've completed all the neurons in this 3-neuron layer, we take a step backwards and start on the hidden layer with 4 neurons.

And this is where things really become beautiful. To find the deltas for each neuron in this layer, we need only the weights to each neuron that uses this neuron's output, and the deltas for those neurons, which we just computed.

The other layers are irrelevant. We don't care about the output layer any more now. All we need are the deltas in the next layer's neurons, and the weights that get us to those neurons.

Figure 18.36 shows how we compute the deltas for the four neurons in the second hidden layer.

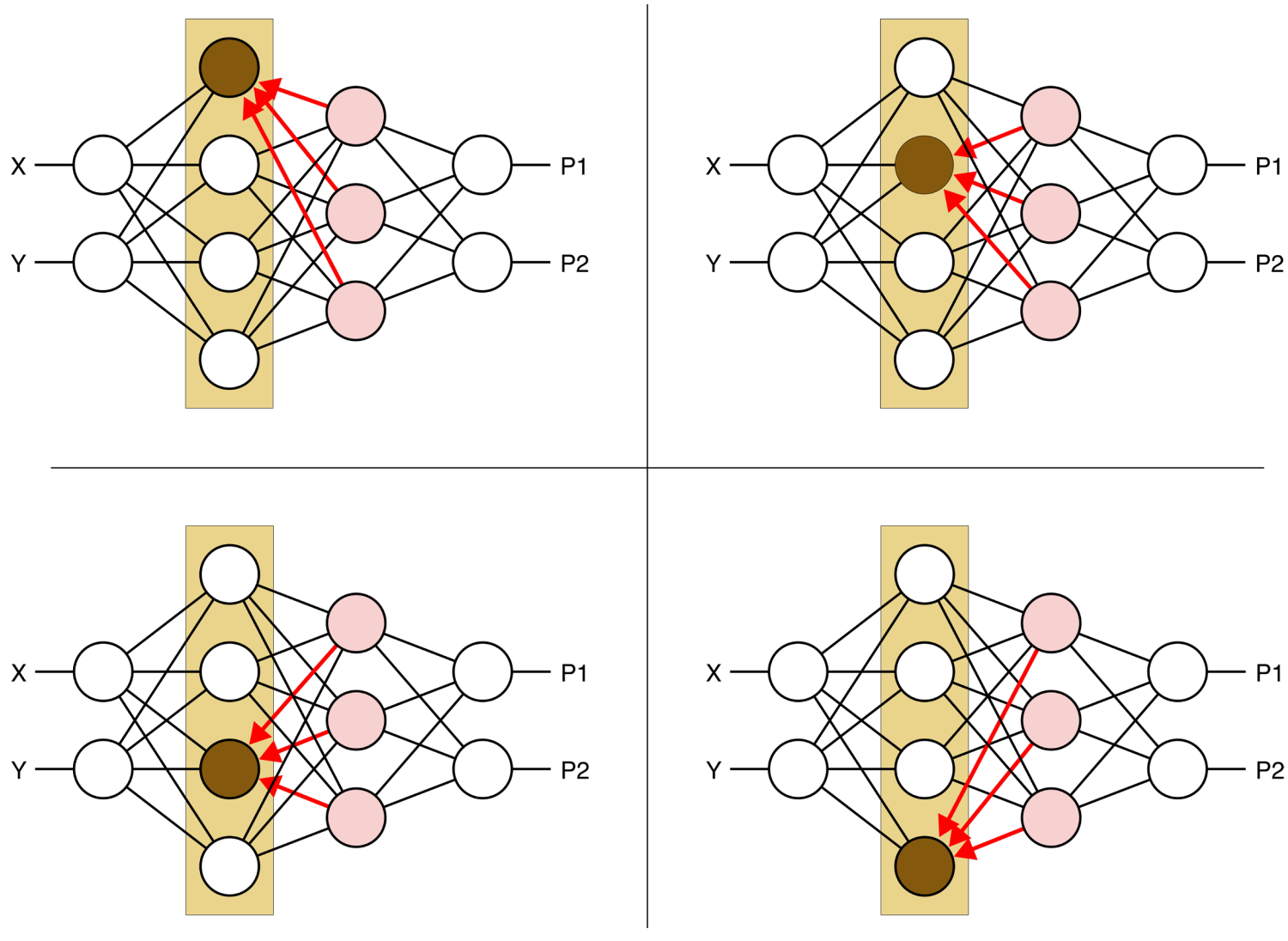

Figure 18.36: Using backprop to find the delta values for the second hidden layer.

When all 4 neurons have had deltas assigned to them, that layer is finished, and we take another step backwards.

Now we're at the first hidden layer with two neurons. Each of these connects to the 4 neurons on the next layer. Once again, all we care about now are the deltas in that next layer and the weights that connect the two layers. For each neuron we find the deltas for all the neurons that consume that neuron's output, multiply those by the weights, and add up the results, as shown in Figure 18.37.

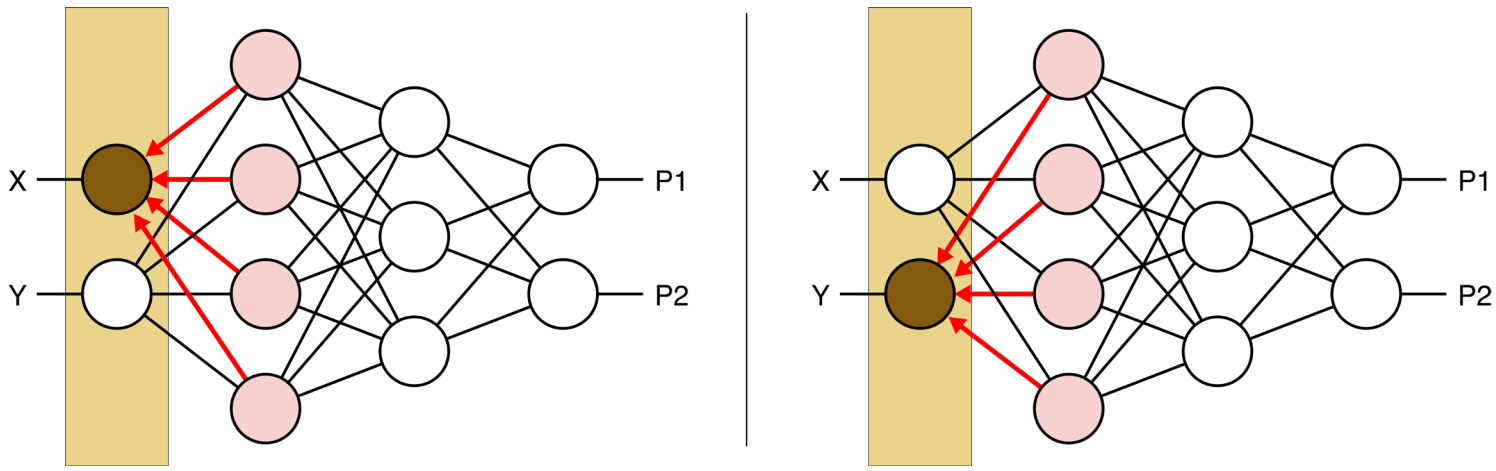

Figure 18.37: Using backprop to find the neurons for the first hidden layer.

When Figure 18.37 is complete, we've found the delta for every neuron in the network.

Now we'll adjust the weights. We'll run through the connections between neurons and use the technique we saw in Figure 18.25 to update for every weight to a new and improved value.

Figure 18.34 through Figure 18.37 show why the algorithm is called *backwards propagation*. We're taking the deltas from any layer and *propagating*, or moving, their information *backwards* one layer at a time, modifying it as we go.

As we've seen, computing each of these delta values is fast. It's just one multiplication per outgoing connection, and then adding those pieces together. That takes almost no time at all.

Backprop becomes highly efficient when we use parallel hardware like a GPU. Because the neurons on a layer of a feed-forward network don't interact, and the weights and deltas that get multiplied are already computed, we can use a GPU to multiply *all* the deltas and weights for *an entire layer* at once.

Computing an entire layer's worth of deltas in parallel saves us a lot of time. If each of our layers had 100 neurons, and we had enough hardware, computing all 400 deltas would take only the same time required to find 4 deltas.

The tremendous efficiency that comes from this parallelism is a key reason why backprop is so important.

Now we have all of the deltas, and we know how to update the weights. Once we actually do update the weights, we should re-compute all the deltas, because they're based on the weights.

We're just about done, but we need to make good on our earlier promise and put activations functions back into our neurons.

18.10 Using Activation Functions

Including the activation function in backpropagation is a small step. But understanding that step and why it's the right thing to do takes a bit of thinking. We left it off in our earlier discussion so we wouldn't get distracted, but let's now get the activation function back in there.

We'll start by thinking about a neuron during the feed-forward step, when we're evaluating a sample and data is flowing from left to right, producing neuron outputs as it goes.

When we calculate the output of a neuron, the sum of the weighted inputs goes through an activation function before it leaves the neuron, as shown in Figure 18.38.

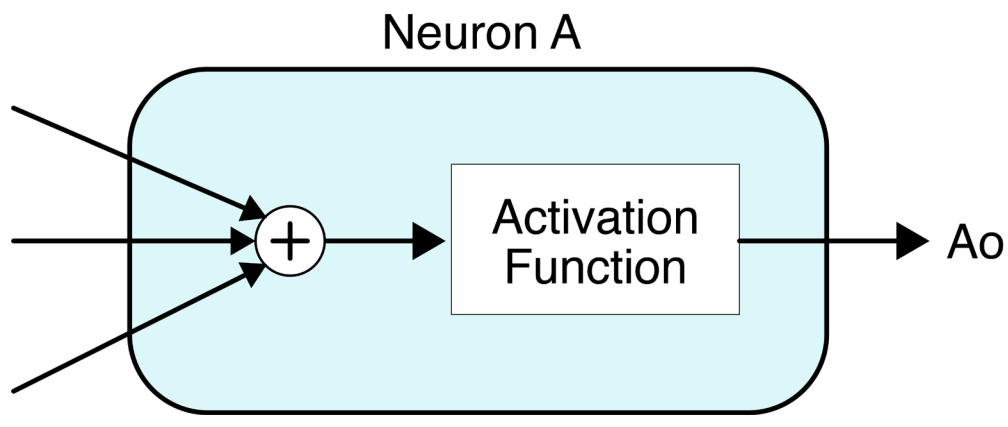

Figure 18.38: Neuron A, with its activation function.

To make things a bit more specific, let's choose an activation function. We'll pick the sigmoid, discussed in Chapter 17, because it's smooth and makes for clear demonstrations. We won't use any particular qualities of the sigmoid, so our discussion will be applicable to any activation function.

Figure 18.39 shows a plot of the sigmoid curve. Increasingly large positive values approach 1 ever more closely without quite getting there, and increasingly large negative values approach 0, but never quite get there, either. Rather than constantly refer to values with phrases like "very, very nearly 1" or "extremely close to 0," let's say for simplicity that input values that are greater than about 7 can be considered to get an output value of 1, while those less than −7 can be considered to get an output of 0. Values in the range (−7, 7) will be smoothly blended in the S-shaped function that gives the sigmoid its name.

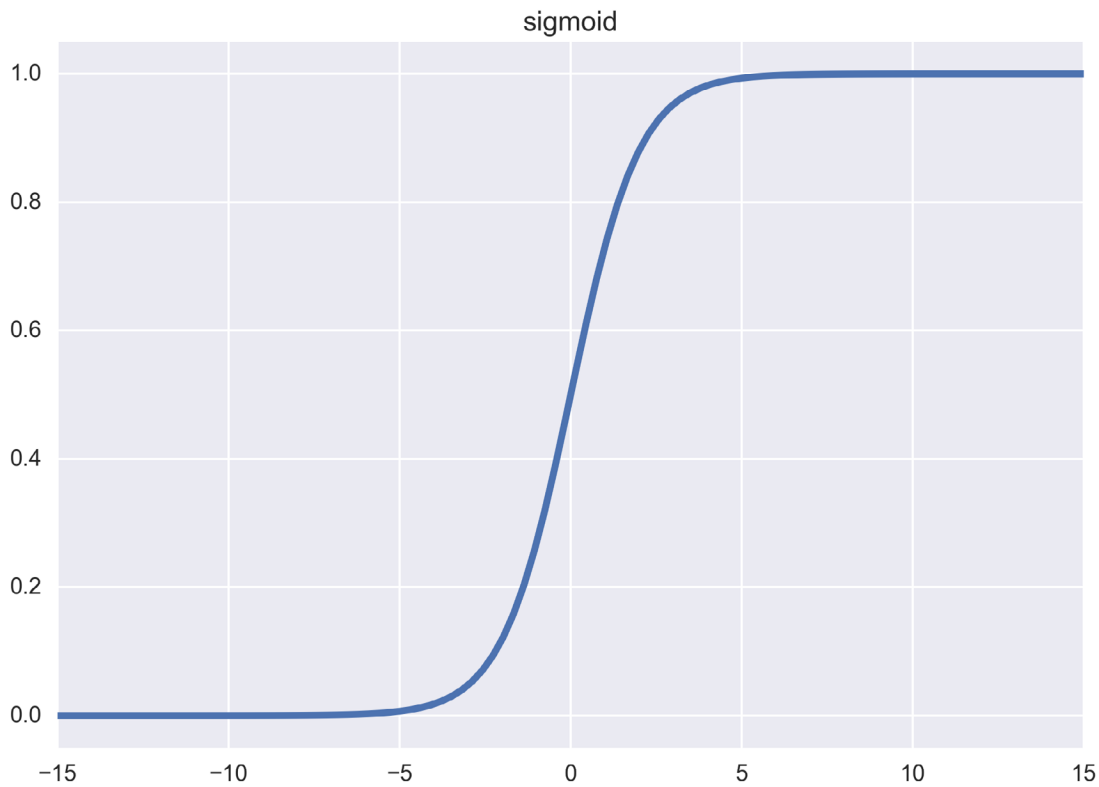

Figure 18.39: The sigmoid curve, plotted from −15 to 15. Values greater than about 7 or less than about −7 are very near to 1 and 0, respectively.

Let's look at neuron C in our original tiny four-neuron network of Figure 18.8. Neuron C gets one input from neuron A, and one from neuron B. For now, let's look just at the input from A, as in Figure 18.40. The value *Ao*, or the output of A, gets multiplied by the weight

AC before it gets summed with all the other inputs in C. Since we're focusing just on the pair of neurons A and C, we'll ignore any other inputs to C. The input value *Ao*×*AC* is then used as the input to the activation function to find the output value *Co*.

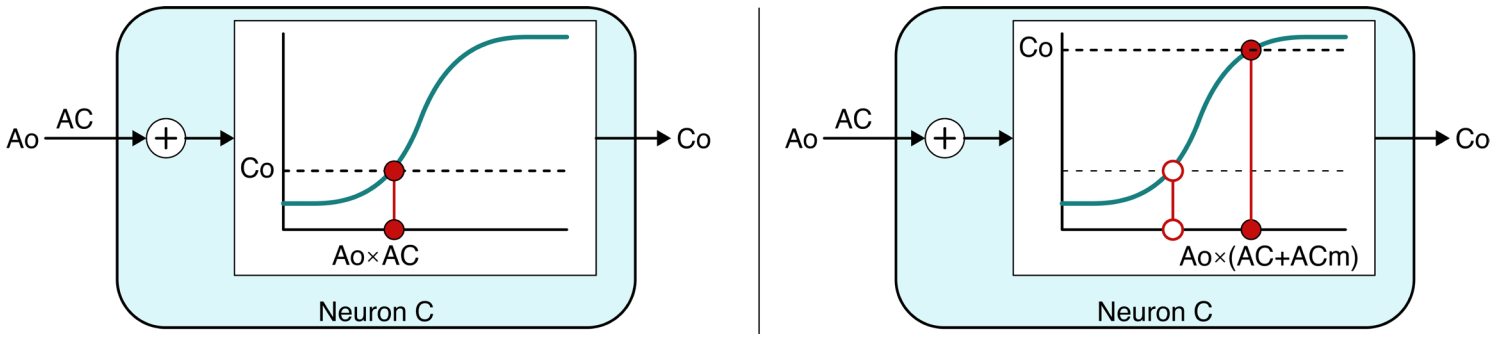

Figure 18.40: Ignoring the other inputs to A for the moment, the input to the activation function is given by multiplying the output of A and the weight *AC*. We can find the value of the activation function at that point, which gives us the output *Co*. Left: When the output of A is *Ao* and the weight is *AC*, the value into the activation function is *Ao*×*AC*. Right: When the output of A is *Ao* and the weight is *AC*+*ACm*, the value into the activation function is *Ao*×(*AC*+*ACm*). Here we show the values from the left plot as dots with white in the center, and the new values as filled-in dots. Note that the change in the output is substantial, because we're on a steep part of the curve.

We know that to reduce the error, we'll be adding some positive or negative number *ACm* to the value of the weight *AC*. The right diagram in Figure 18.40 shows the result of adding a positive value of *ACm*. In this case, the output *Co* changes by a lot, because we're on a steep part of the curve.

So this says that by adding *ACm* to the weight, we're going to have a *bigger* effect on the output error than we would have had without the activation function, because that function has turned our change of *ACm* into something larger.

Suppose instead our starting value of *Ao*×*AC* put us near a shallow part of the curve, and we add the same *ACm* to *AC*, as in Figure 18.41.

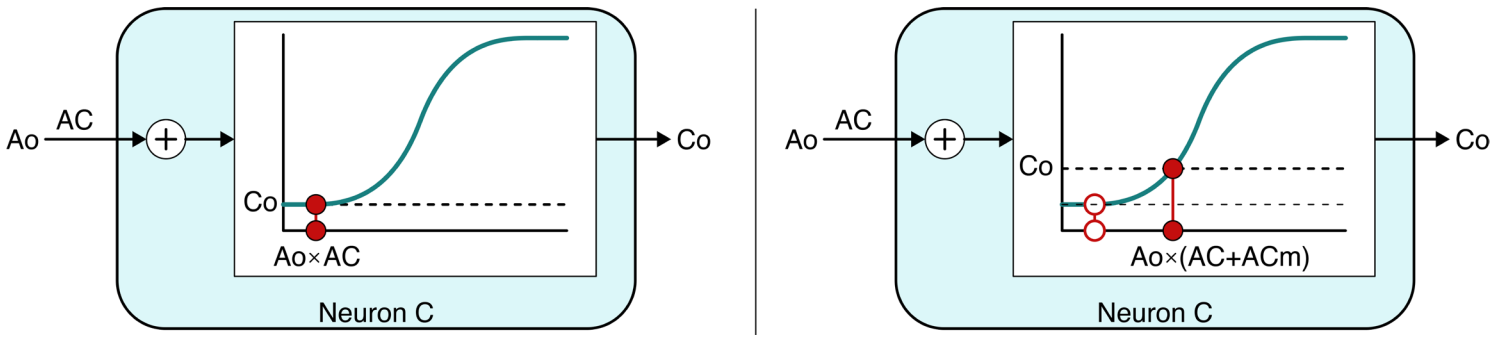

Figure 18.41: In a shallow part of the curve. Left: Before adding *ACm* as in Figure 18.40. Right: Adding this value results in a small change in the output of C.

Now adding the same value *ACm* to the weight causes a smaller change in *Co* than before. In this case, it's even less than *ACm* itself. The smaller change to the output means there will be a smaller change in the network error. In other words, adding *ACm* to the weight in this situation will result in a *smaller* change to the output error than we'd get if there was no activation function present.

What we'd love to have is something that can tell us, for any point on the activation function curve, how steep the curve is at that point. When we're at a point where the curve is going up steeply to the right, positive changes in the input will be amplified a lot, as in Figure 18.40. When we're at a point where the curve is going up shallowly to the right, as in Figure 18.41, such changes will be amplified by a little.

If we change the input with a negative value of *ACm*, and move our stating point to the left, then the amount of slope tells us how much the change in the error will decrease. We get the same situations as in Figure 18.40(b) and Figure 18.41(b), but with the starting and ending positions reversed.

Happily, we already know how to find the slope of a curve: that's just the derivative. Figure 18.42 shows the sigmoid, and its derivative.

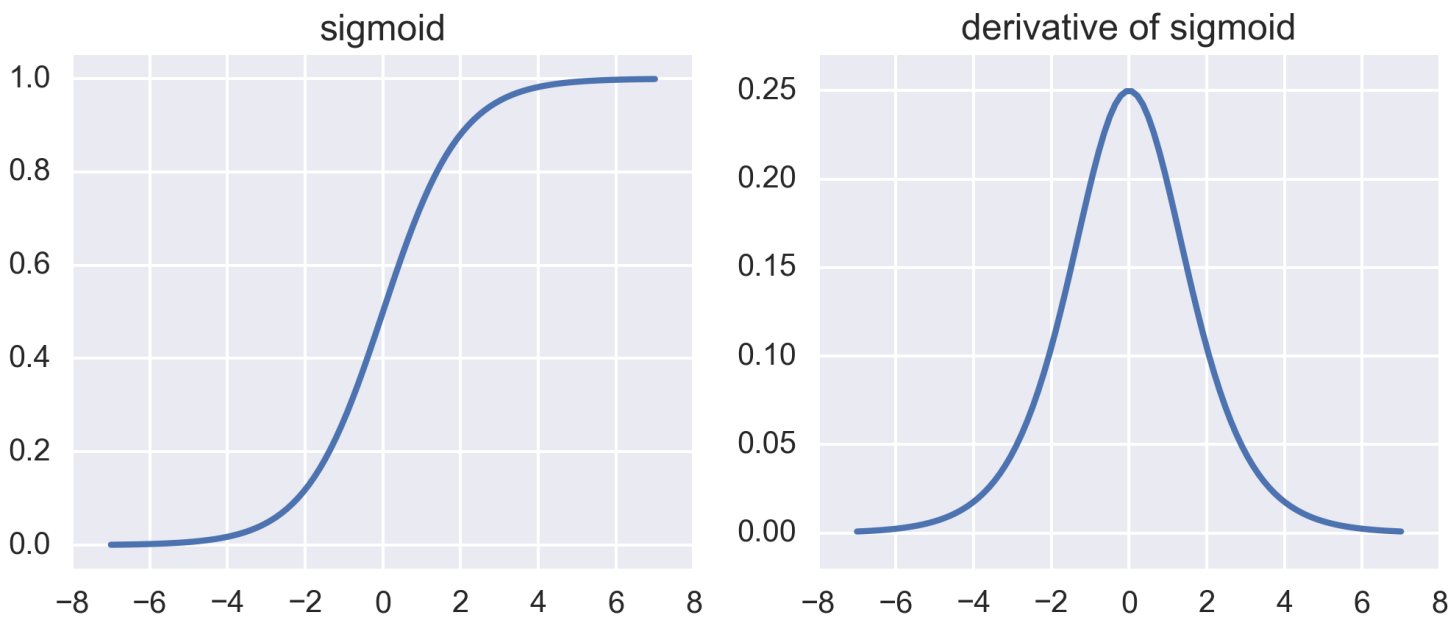

Figure 18.42: The sigmoid curve and its derivative. Note that the vertical scales are different for the two plots.

The sigmoid is flat at the left and right ends. So if we're in the flat regions and we move left or right a little bit, that will cause little to no change in the output of the function. In other words, the curve is flat, or has no slope, or has a derivative of 0. The derivative increases as the input moves from about −7 to 0 because the curve is getting steeper. Then the derivative decreases back to 0 again even as the input keeps going up because the curve becomes more shallow off to the right, approaching 1 but never quite getting there.

If we were to work through the math, we'd find that this derivative is exactly what we need to fix our prediction of the change to the error based on a change to a weight. When we're on a steep part of the curve, we want to crank up the value of delta for this neuron, because changes to its inputs will cause big changes in the activation function, and thus have big changes to the network error. When we're on a shallow part of the curve, then changes to the inputs will have little effect on the change in the output, so we want to make this neuron's delta smaller.

In other words, to account for the activation function, we just take the delta we normally compute, and multiply it by the derivative of the activation function, evaluated at the same point that we used during

the forward pass. Now the delta accounts for how the activation function will exaggerate or diminish the amount of change in the neuron's output, and hence its effect on the network error.

To keep things nicely bundled, we can perform this step immediately after computing the delta as we did before.

The whole business for one neuron is summarized in Figure 18.43. Here we imagine we have a neuron H. The top part is the forward pass, when we're evaluating a sample and computing this neuron's output. Following a common convention, we've given the name *z* to the result of the summation step. The value of the activation function is what we get when we look vertically upwards from the point *z* on the X axis. The bottom part of the figure is the backward pass, where we compute this neuron's delta. Again, we use *z* to find the derivative of the activation function, and we multiply our incoming sum by that before passing it on.

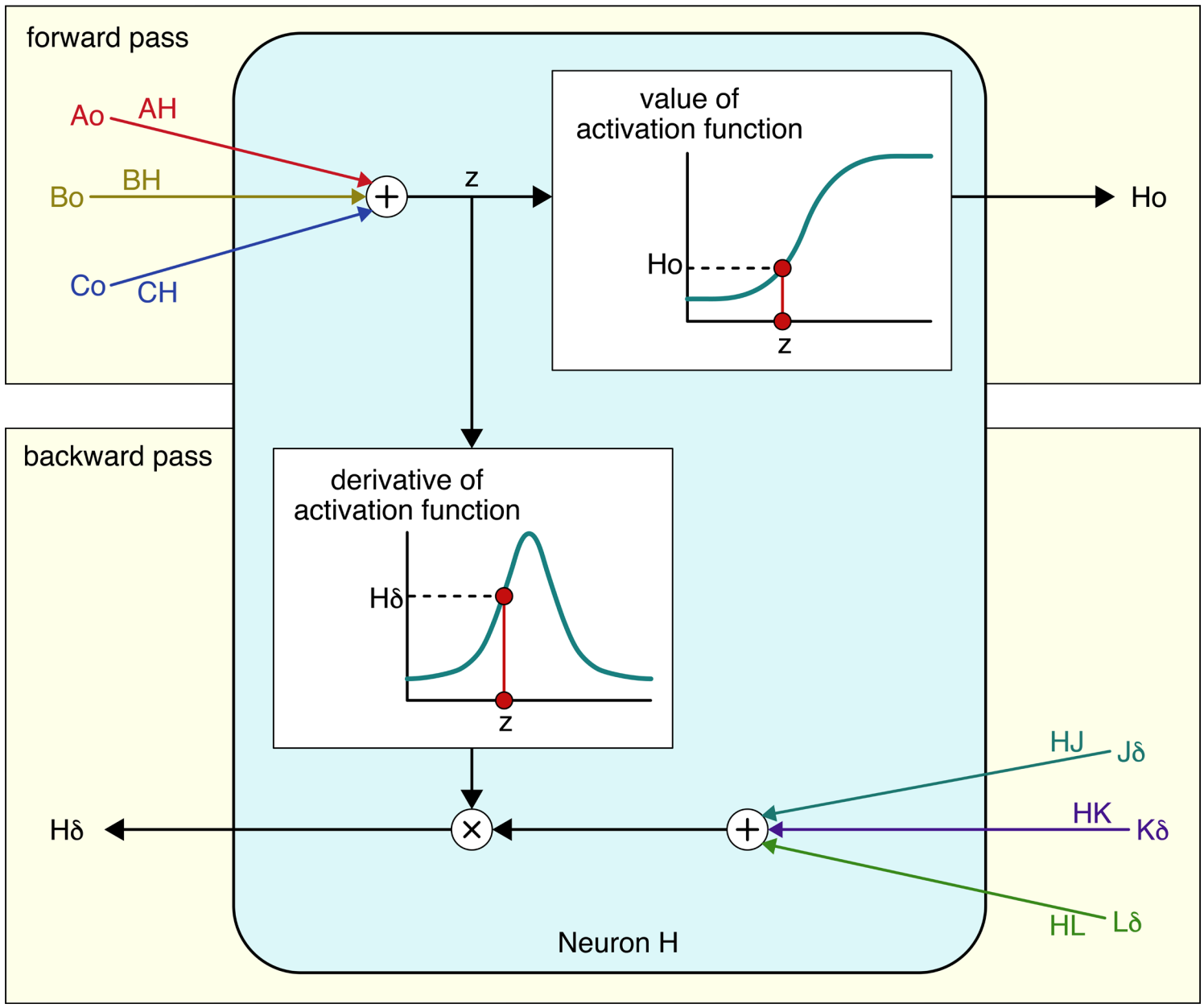

Figure 18.43: Network evaluation and backpropagation of deltas in a nutshell, for neuron H. Top: In the forward pass, the weighted inputs are added together, giving us a value we call *z*. The value of the activation function at *z* is our output *Ho*. Bottom: In the backward pass, the weighted deltas are added together, and we use the *z* from before to look up the derivative of the activation function. We multiply the sum of the weighted deltas by this value, giving us *Hδ*.

In this figure, neuron H has three inputs, coming from neurons A, B, and C, with output values *Ao*, *Bo*, and *Co*. During the forward pass, when we're finding the neuron's output, these are multiplied respectively by the weights *AH*, *BH*, and *CH*, and then added together. We've labeled this sum with the letter *z*. Now we look up *z* in the activation function, and its value is *Ho*, the output of this neuron.

Now when we run the backward pass to find the neuron's delta, we find the deltas of the neurons that use *Ho* as an input. Let's say they're neurons J, K and L. So we multiply their deltas $J\delta$, $K\delta$, and $L\delta$ by their respective weights *HJ*, *HK*, and *HL*, and add up those results.

Now we get the value of *z* from the forward pass, and use it to find the value of the derivative of the activation function. We multiply the sum we just found with this number, and the result is $H\delta$, the delta for this neuron.

This all goes for the output neurons too, if they have activation functions, only in the backward pass we use the error information rather than deltas from the next layer.

Notice how compact and local everything is. The forward pass depends only on the output values of the previous layer, and the weights that connect to them. The backward pass depends only on the deltas from the following layer, the weights that connect to them, and the activation function.

Now we can see why we were able to get away with ignoring the activation function throughout most of this chapter. We can pretend that we really did have an activation function all along: the **identity activation function**, shown in Figure 18.44. This has an output that's the same as its input. That is, it has no effect.

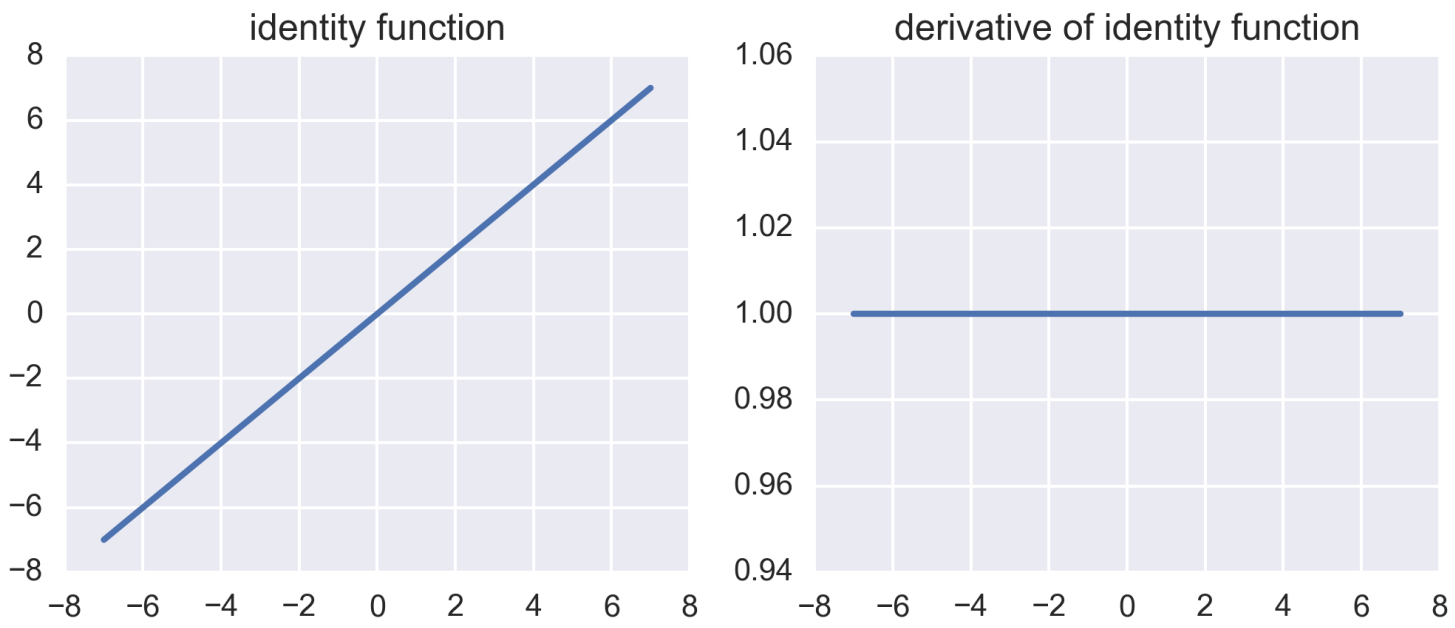

Figure 18.44: The identity function as activation function. Left: The identity produces as output the same value it received as input. Right: Its derivative is 1 everywhere, because the function has a constant slope of 1.

As Figure 18.44 shows, the derivative of the identify activation function is always 1 (that's because the function is a straight line with slope of 1 everywhere, and the derivative at any point is just the slope of the curve at that point). Let's think back on our discussion of backpropagation and include this identity activation within every neuron. During the forward pass, the outputs would be unchanged by this function, so it has no effect. During the backward pass, we'd always multiply the summed deltas by 1, again having no effect.

We said earlier that our results weren't limited to using the sigmoid. That's because we didn't use any special properties of sigmoid in our discussion, other than to assume it has a derivative everywhere. This is why activation functions are designed so that they have a derivative for every value (recall from Chapter 5 that library routines automatically apply mathematical techniques to take care of any spots that don't have a derivative).

Let's look at the popular ReLU activation function. Figure 18.45 shows the ReLU function and its derivative.

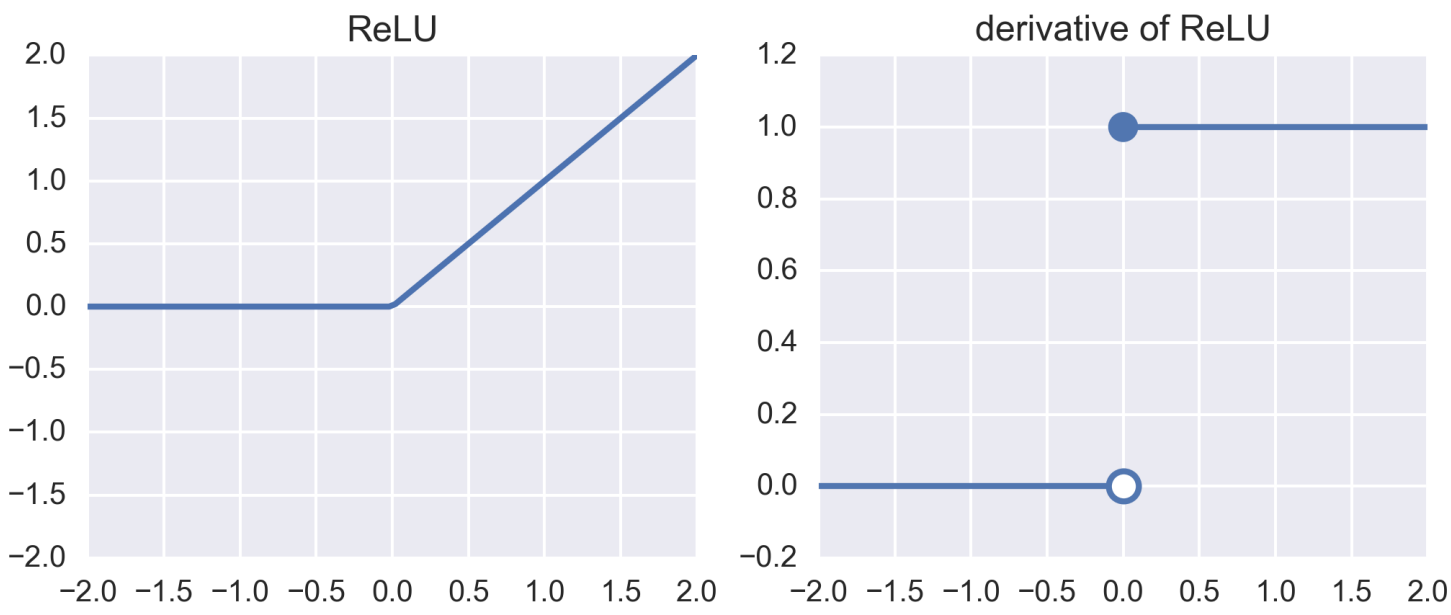

Figure 18.45: The ReLU function as an activation function. Left: ReLU returns its input when that value is 0 or larger, and otherwise returns 0. Right: Its derivative is a step function, 0 for inputs less than 0 and 1 for inputs greater than 0. The sudden jump at 0 is automatically managed for us by libraries so that we have a smooth derivative everywhere.

Everything we did with the sigmoid can be applied to the ReLU, without change. And the same goes for any other activation function.

That wraps things up. We've now put activation functions back into our neurons.

That brings us to the end of basic backpropagation.

Before we leave the discussion, though, let's look at a critical control that keeps things working well: the learning rate.

18.11 The Learning Rate

In our description of updating a weight, we multiplied the left neuron's output value and the right neuron's delta, and subtracted that from the weight (recall Figure 18.25 from much earlier).

But as we've mentioned a few times, changing a weight by a lot in a single step is often a recipe for trouble. The derivative is only accurate for very tiny changes in a value. If we change a weight by too much, we can jump right over the smallest value of the error, and even find ourselves increasing the error.

On the other hand, if we change a weight by too little, we might see only the tiniest bit of learning, slowing everything down. Still, that inefficiency is usually better than a system that's constantly over-reacting to errors.

In practice, we control the amount of change to the weights during every update with a hyperparameter called the **learning rate**, often symbolized by the lower-case Greek letter η (eta). This is a number between 0 and 1, and it tells the weights how much of their newly-computed value to use when they update.

When we set the learning rate to 0, the weights don't change at all. Our system will never change and never learn. If we set the learning rate to 1, the system will apply big changes to the weights, and might overshoot the mark. If this happens a lot, the network can spend its time constantly overshooting and then compensating, with the weights bouncing around and never settling into their best values. So we usually set the learning rate somewhere between these extremes.

Figure 18.46 shows how the learning rate is applied. We just scale the value of $-(Ao \times C\delta)$ by η before adding it back in to AC.

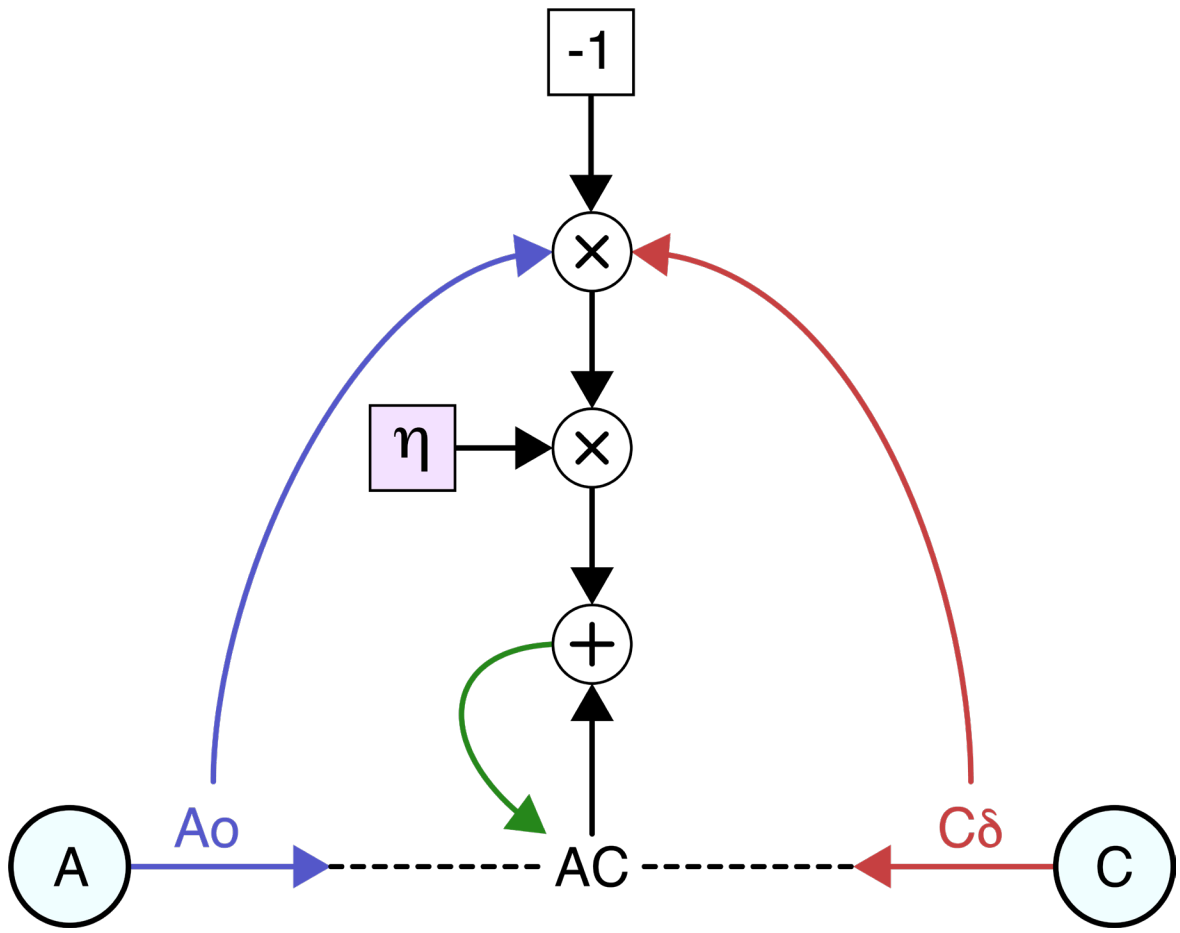

Figure 18.46: The learning rate helps us control how fast the network learns by controlling the amount by which weights change on each update. Here we see Figure 18.25 with an extra step that multiplies the value $-(Ao \times C\delta)$ by the learning rate δ before adding it to AC. When δ is a small positive number (say 0.01), then each change will be small, which often helps the network learn.

The best choice of the learning rate is dependent on the specific network we've built and the data we're training on. Finding a good choice of learning rate can be essential to getting the network to properly learn at all. Once the system is learning, changing this value can affect whether that process goes quickly or slowly. Usually we have to hunt for the best value of eta using trial and error. Happily, there are algorithms that automate the search for a good starting value for the learning rate, and other algorithms that fine-tune the learning rate as learning progresses. We'll see such algorithms in Chapter 19. As a general rule of thumb, and if none of our other choices direct us to a particular learning rate, we often start with a value around 0.01 and then train the network for a while, watching how well it learns. Then we then raise or lower it from there and train again, over and over, hunting for the value that learns most efficiently.

18.11.1 Exploring the Learning Rate

Let's see how backprop performs with different learning rates. We'll build a classifier to find the boundary between the two half-moons that we used in Chapter 15. Figure 18.47 shows our training data of about 1500 points. We pre-processed this data to give it zero mean and a standard deviation of 1 for each feature.

Figure 18.47: About 1500 points generated synthetically by the make_moons() routine in scikit-learn.

Because we have only two categories, we'll build a **binary classifier**. This lets us skip the whole one-hot encoding of labels and dealing with multiple outputs, and instead use just one output neuron. If the value is near 0, the input is in one class. If the output is near 1, the input is in the other class.

Our classifier will have 2 hidden layers, each with 4 neurons. These are essentially arbitrary choices that give us a network that's just complex enough for our discussion. Both layers will be fully-connected, so every neuron in the first hidden layer will send its output to every neuron in the second hidden layer. Figure 18.48 shows the idea.

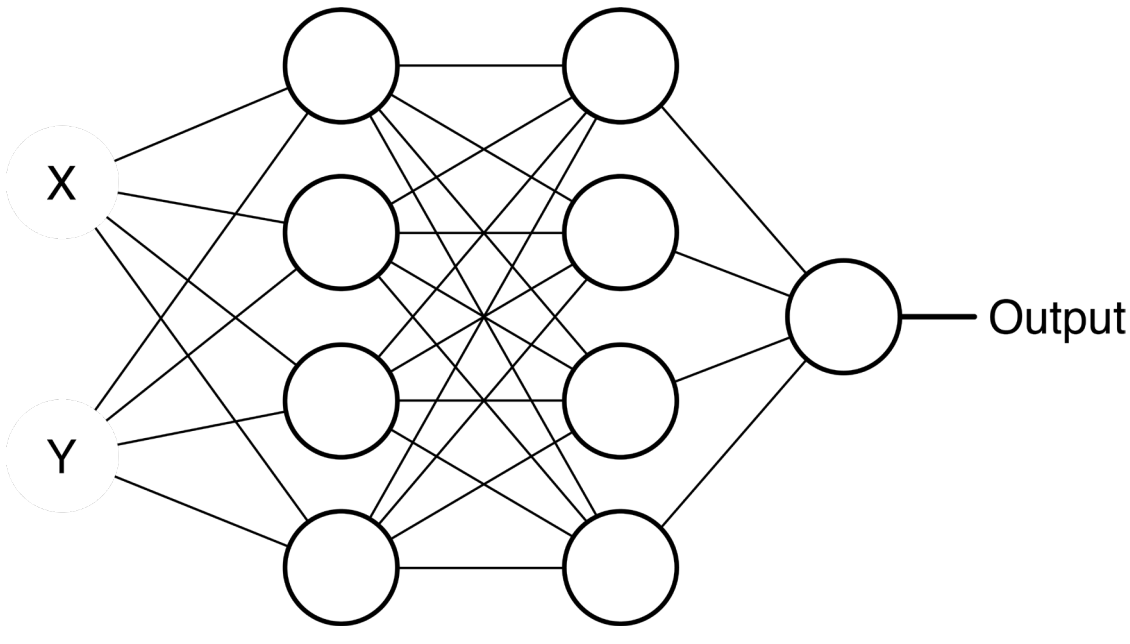

Figure 18.48: Our binary classifier takes in two values as input (the X and Y of each point). Each input goes into the 4 neurons on the first layer. Each of those 4 neurons connects to each of the 4 neurons on the next hidden layer. Then a single neuron takes the outputs of the second hidden layer and presents a single value for output. In this network, we've used ReLU activation functions for the neurons in the hidden layers, and a sigmoid activation function on the output neuron.

How many weights are in our network? There are 4 coming out of each of the 2 inputs, then 4×4 between the layers, and then 4 going into the output neuron. That gives us (2×4)+(4×4)+4=28. Each of the 9 neurons also has a bias term, so our network has 28+9=37 weights. They start with small random numbers. Our goal is to use backprop to adjust those 37 weights so that the number that comes out of the final neuron always matches the label for that sample.

As we discussed above, we'll evaluate one sample, calculate the error, compute the deltas with backprop, and then update the weights using the learning rate. Then we'll move on to the next sample. Note that if the error is 0, then the weights won't change at all. Each time we process all the samples in the training set, we say we've completed one **epoch** of training.

Our discussion of backprop mentioned how much we rely on making "small changes" to the weights. There are two reasons for this. The first is that the direction of change for every weight is given by the derivative

(or gradient) of the error at that weight. But as we saw, the gradient is only accurate very near the point we're evaluating. If we move too far, we may find ourselves increasing the error rather than decreasing it.

The second reason for taking small steps is that changes in weights near the start of the network will cause changes in the outputs of neurons later on, and we've seen that we use those neuron outputs to help compute the changes to the later weights. To prevent everything from turning into a terrible snarl of conflicting moves, we change the weights only by small amounts.

But what is "small"? For every network and data set, we have to experiment to find out. As we saw above, the size of our step is controlled by the **learning rate**, or **eta** (η). The bigger this value, the more each weight will move towards its new value.

Let's start with a really large learning rate of 0.5. Figure 18.49 shows the boundaries computed by our network for our test data.

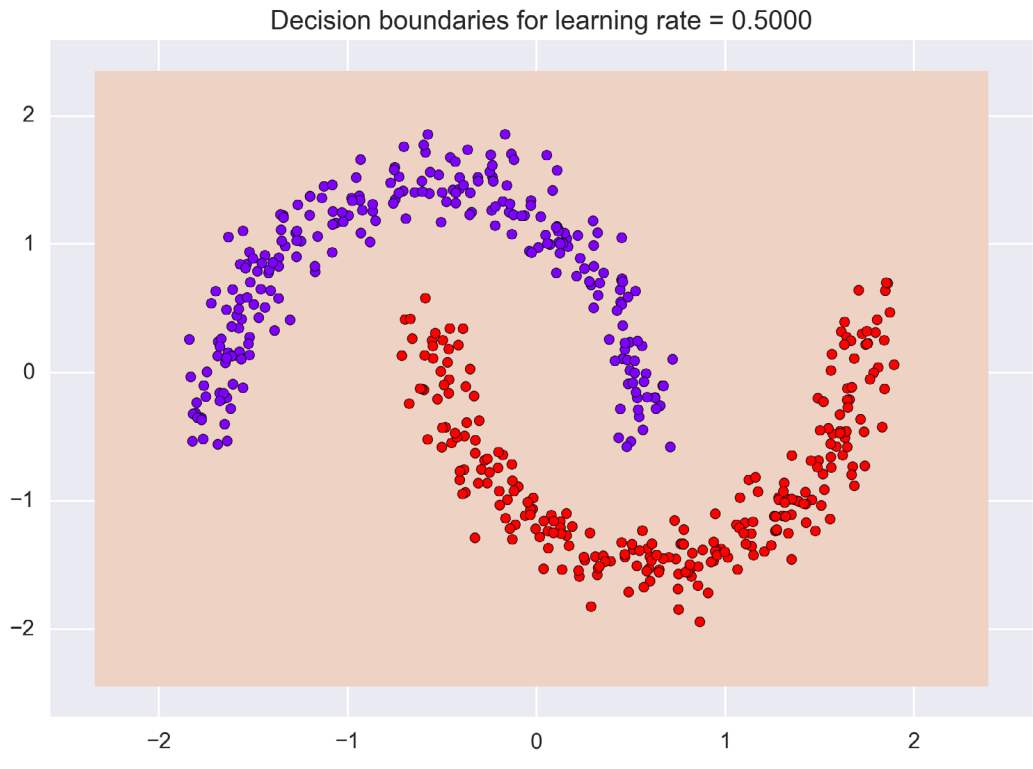

Figure 18.49: The boundaries computed by our network using a learning rate of 0.5.

This is terrible. Everything is being assigned to a single class, shown by the light orange background. If we look at the accuracy and error (or loss) after each epoch, we get the graphs of Figure 18.50.

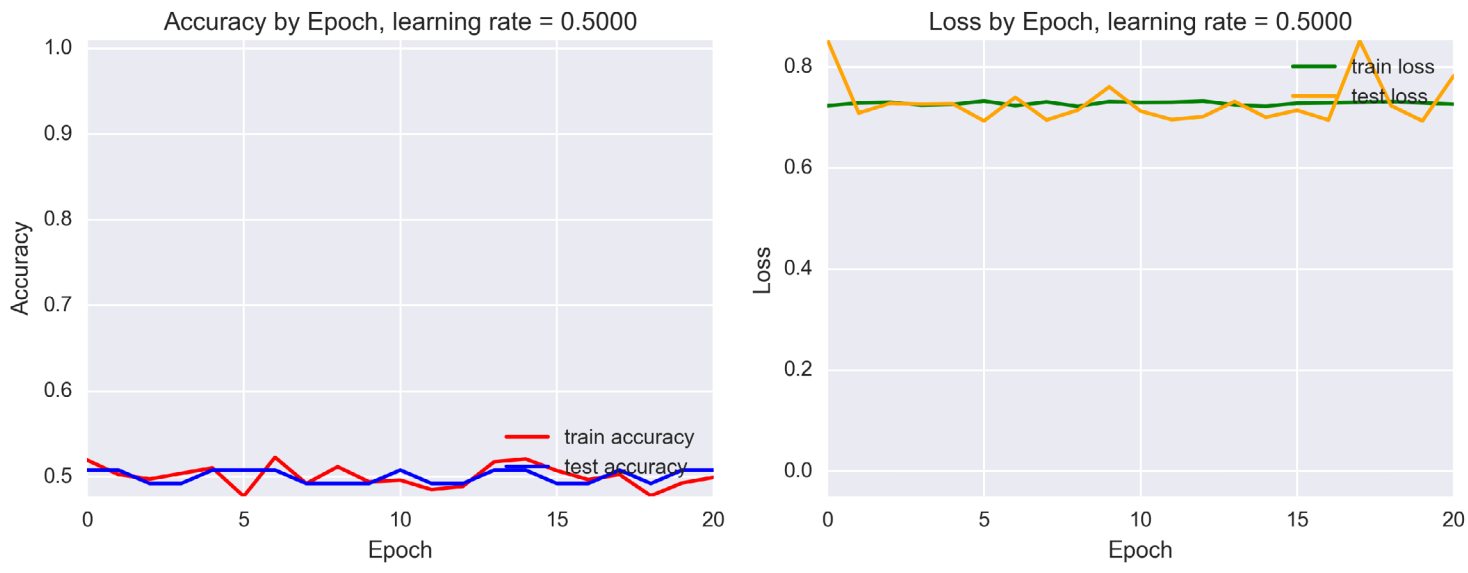

Figure 18.50: Accuracy and loss for our moons data with a learning rate of 0.5.

Things are looking bad. As we'd expect, the accuracy is just about 0.5, meaning that half the points are being misclassified. This makes sense, since the red and blue points are roughly evenly divided. If we assign them all to one category, as we're doing here, half of those assignments will be wrong. The loss, or error, starts high and doesn't fall. If we let the network run for hundreds of epochs it continues on in this way, never improving.

What are the weights doing? Figure 18.51 shows the values of all 37 weights during training.

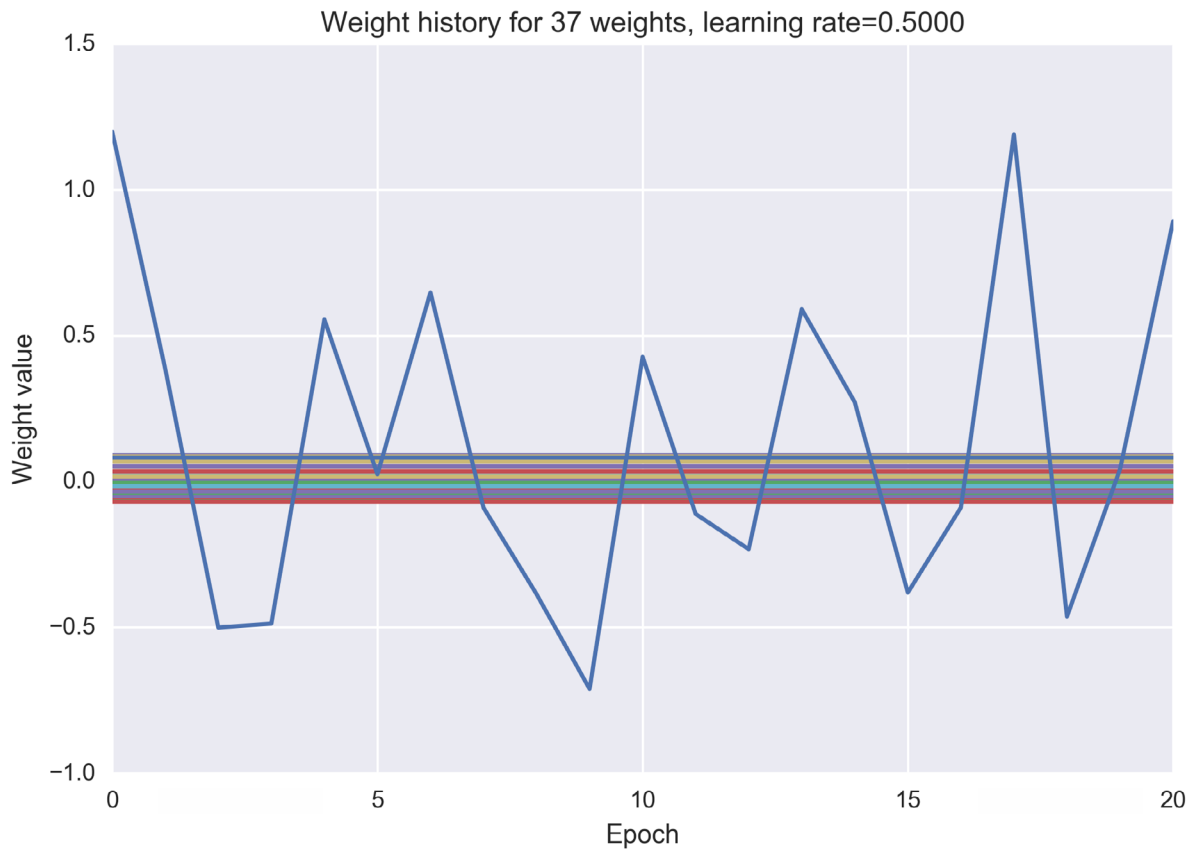

Figure 18.51: The weights of our network when using a learning rate of 0.5. One weight is constantly changing and overshooting its goal, while the others aren't making any visible changes.

Most of the weights don't seem to be moving at all, but they could be meandering a little bit. The graph is dominated by one weight that's jumping all over. That weight is one of those going into the output neuron, trying to move its output around to match the label. That weight goes up, then down, then up, jumping too far every time, then over-correcting by too much, then over-correcting for that, and so on.

These results are disappointing, but they're not shocking, because a learning rate of 0.5 is *big*.

Let's reduce the training rate by a factor of 10 to a more common value of 0.05. We'll change absolutely nothing else about the network and the data, and we'll even re-use the same sequence of pseudo-random numbers to initialize the weights. The new boundaries are shown in Figure 18.52.

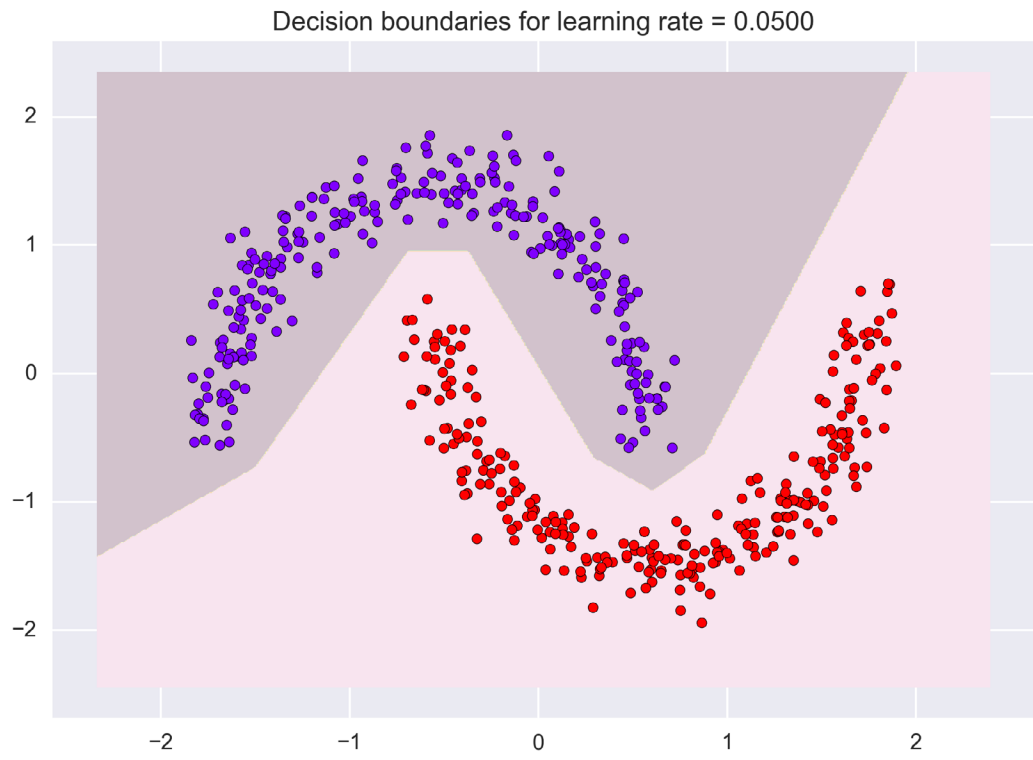

Figure 18.52: The decision boundaries when we use a learning rate of 0.05.

This is *much* better! This is great! Looking at the graphs in Figure 18.53 reveals that we've reached 100% accuracy on both the training and test sets after about 16 epochs.

Figure 18.53: Accuracy and loss for our network when using a learning rate of 0.05.

What are the weights doing? Figure 18.54 shows us their history. Overall, this is way better, because lots of weights are changing. Some weights are getting pretty large. In Chapter 20 we'll cover regularization

techniques to keep the weights small in deep networks, and later in this chapter we'll see why it's nice to keep the weights small (say, between –1 and 1). For the moment, let's just note that the weights have each learned a good value.

Figure 18.54: The weights in our network over time, using a learning rate of 0.05.

So that's success. Our network has learned to perfectly sort the data, and it did it in only 16 epochs, which is nice and fast. On a late 2014 iMac without GPU support, the whole training process took less than 10 seconds.

Just for fun, let's lower the learning rate down to 0.01. Now the weights will change even more slowly. Does this produce better results?

Figure 18.55 shows the decision boundary resulting from these tiny steps. The boundary seems to use more straight lines than the boundary in Figure 18.52, but both boundaries separate the sets perfectly. We might prefer the boundaries of Figure 18.52 in some situations, as they seem to better follow the shape of the data.

Figure 18.55: The decision boundaries for a learning rate of 0.01.

Figure 18.56 show our accuracy and loss graphs. Because our learning rate is so much slower, our network takes around 170 epochs to get to 100% accuracy, rather than the 16 in Figure 18.54.

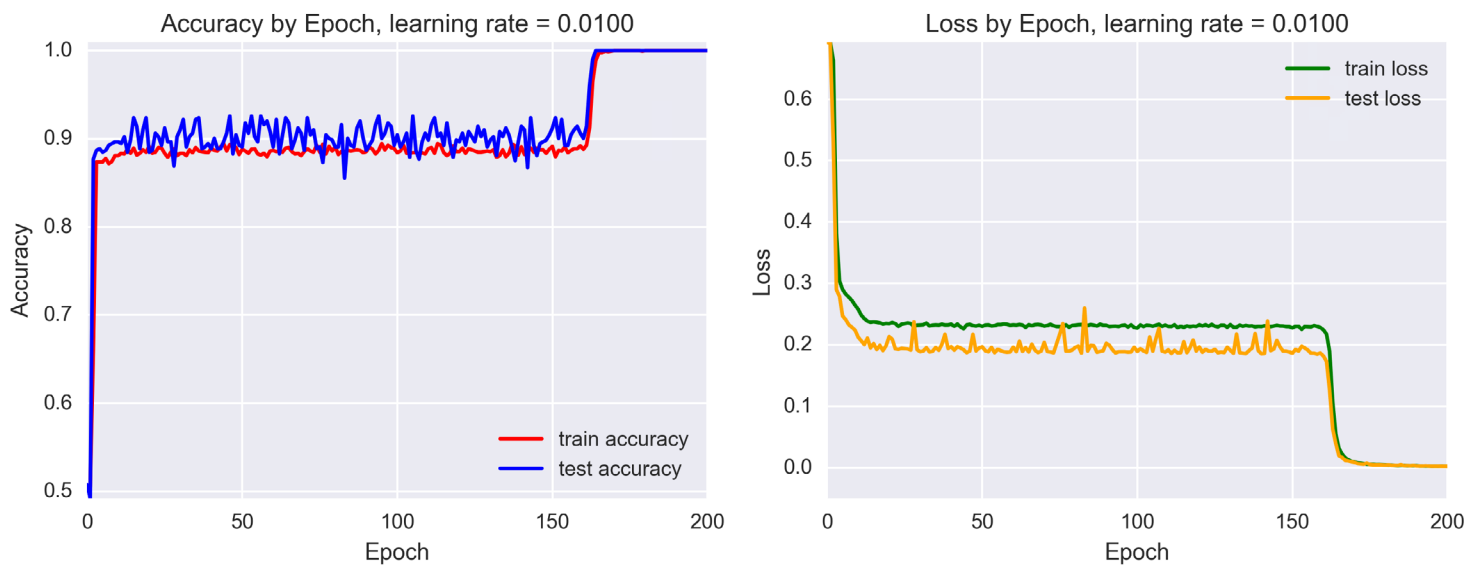

Figure 18.56: The accuracy and learning rate for our network using a learning rate of 0.01.

These graphs show an interesting learning behavior. After an initial jump, both the training and test accuracies reach about 90% and plateau there. At the same time, the losses hit about 0.2 (for the test data)

and 0.25 (for the training), and they plateau as well. Then around epoch 170, things improve rapidly again, with the accuracy climbing to 100% and the errors dropping to 0.

This pattern of alternating improvement and plateaus is not unusual, and we can even see a hint of it in Figure 18.53 where there's an imperfect plateau between epochs 3 and 8. These plateaus come from the weights finding themselves on nearly flat regions of the error surface, resulting in near-zero gradients, and thus their updates are very small.

Though our weights might be getting stuck in local minima, it's more common for them to get caught in a flat region of a saddle, like those we saw in Chapter 5 [Dauphin14]. Sometimes it takes a long time for one of the weights to move into a region where the gradient (or derivative) is large enough to give it a good push. When one weight gets moving, it's common to see the others kick in as well, thanks to the cascading effect of that first weight's changes on the rest of the network.

The values of the weights follow almost the same pattern over time, as shown in Figure 18.57. The interesting thing is that at least some of the weights are not flat, or on a plateau. They're changing, but very slowly. The system is getting better, but in tiny steps that don't show up in the performance graphs until the changes become bigger around epoch 170.

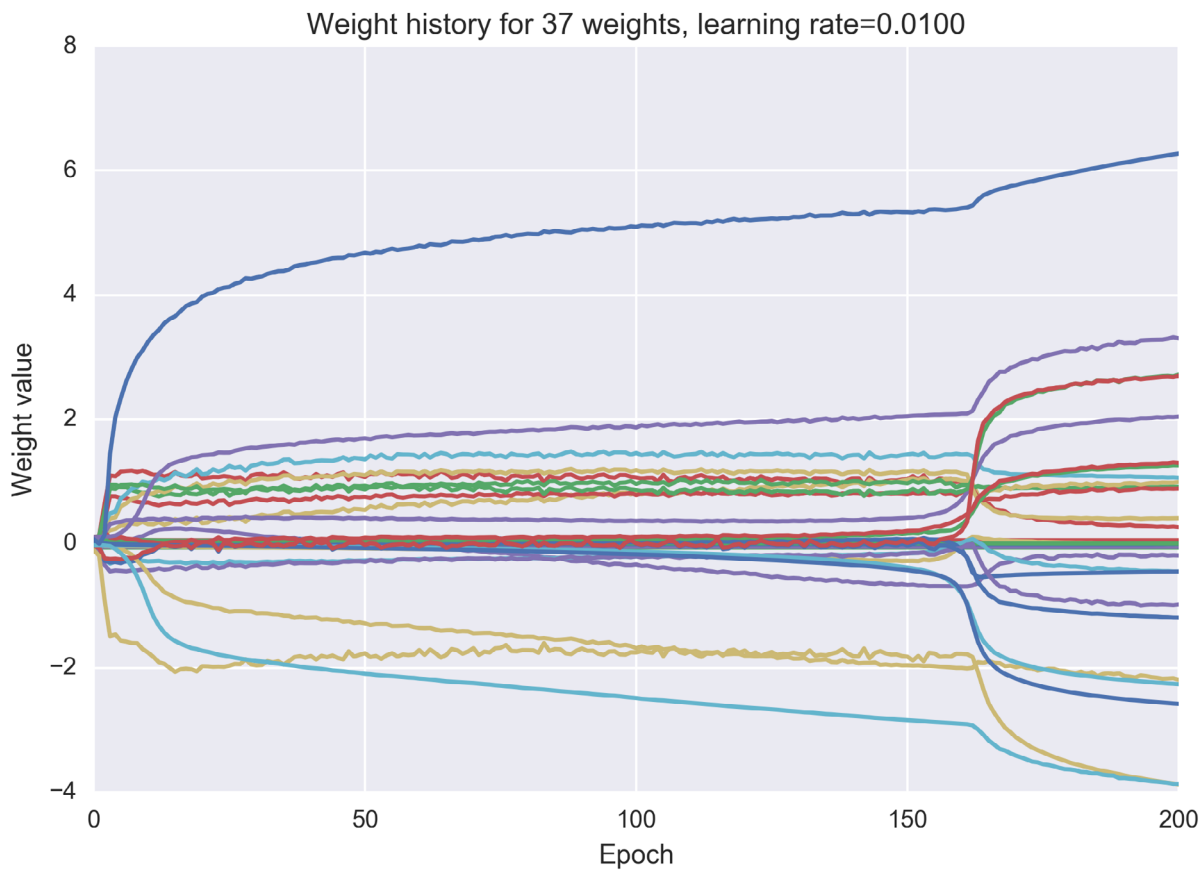

Figure 18.57: The history of our weights using a learning rate of 0.01.

The weights seem to be growing a little bit even at epoch 200. If we let the system continue, these weights will continue to slowly grow over time with no apparent effect on the accuracies or errors.

So was there any benefit to lowering the learning rate down to 0.01? Not really. Even at 0.05, the categorization was already perfect on both the training and test data. In this case, the smaller learning rate just meant the network took longer to learn.

This investigation has shown us how sensitive the network is to our choice of learning rate.

When we look for the best value for the learning rate, it can feel like we're the character of Goldilocks in recent versions of the fable *Goldilocks and the Three Bears* [Wikipedia17]. We're searching for something that's not too big, and not too small, but "just right."

When our learning rate was too big, the weights took steps that were too large, and the network never settled down or improved its performance. When the learning rate was too small, our progress was very slow, as the weights sometimes were creeping from one value to another at a glacial pace.

When the learning rate was just right, training was fast and efficient, and produced great results. In this case, they were perfect.

This kind of experimenting with the learning rate is part of developing nearly every deep learning network. The speed with which backprop changes the weights needs to be tuned to match the type of network and the type of data. Happily, we'll see in Chapter 19 that there are automatic tools that can handle the learning rate for us in sophisticated ways.

18.12 Discussion

Let's recap what we've seen, and then consider some of the implications of the backpropagation algorithm.

18.12.1 Backprop In One Place

To recap quickly, we start by running a sample through the network and calculate the output for every neuron, as in Figure 18.58(a).

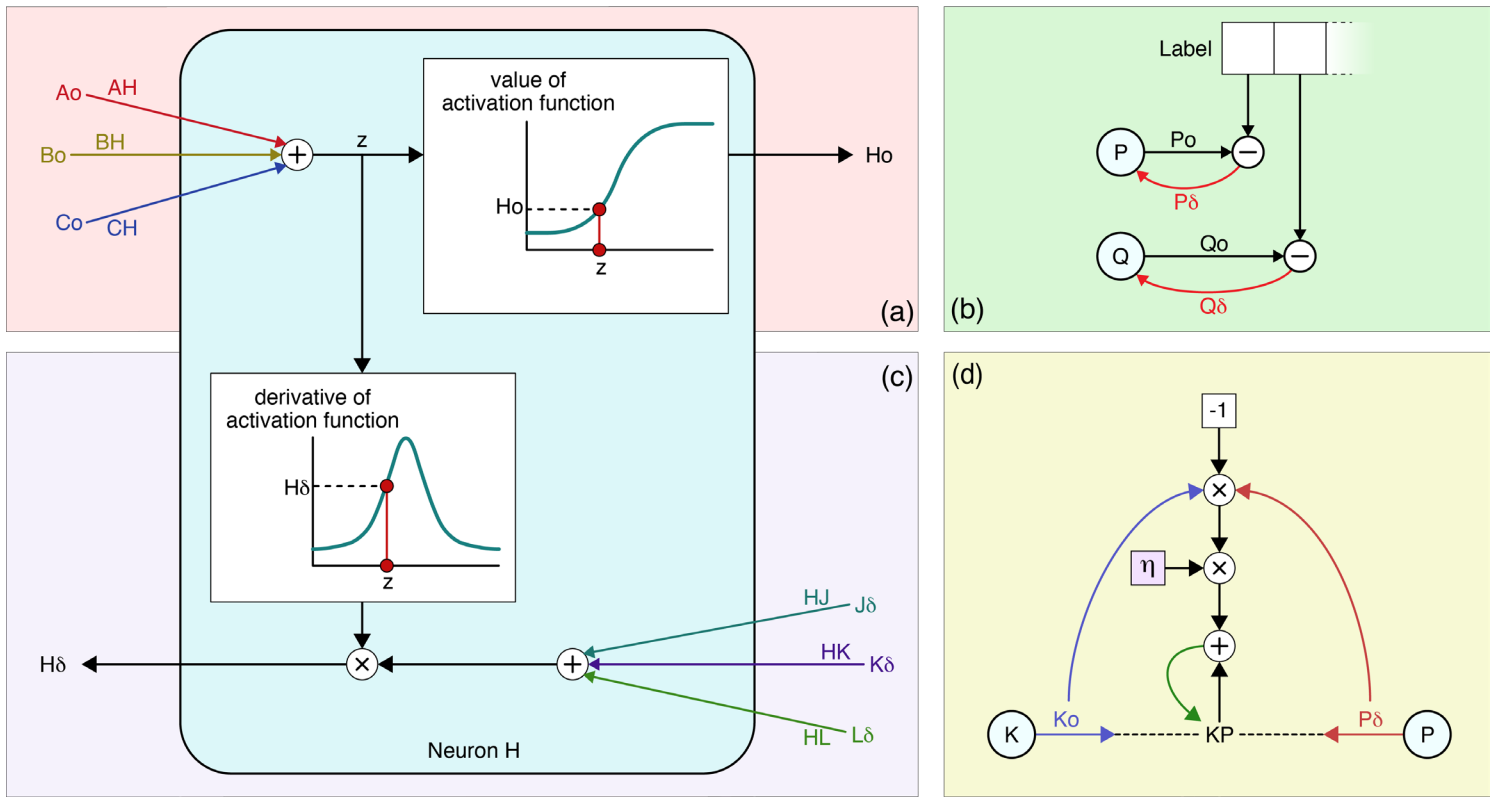

Figure 18.58: The backprop algorithm, along with weight update, in a nutshell. This figure collects Figure 18.20, Figure 18.43, and Figure 18.46 in one place. Part (a) shows the forward step, part (b) shows finding the deltas for the output neurons, part (c) steps propgates the deltas backwards, and step (d) updates the weights.

Then we kick off the backprop algorithm in Figure 18.58(b). We find the delta value for each of the output neurons, telling us that if the neuron's output changes by a certain amount, the error will change by that amount times the neuron's delta.

Now we take a step backwards in Figure 18.58(c) to the previous layer of the network, and find the deltas for all the neurons in that layer. We need only the deltas from the output layer, and the weights between the two layers.

Once all the deltas are assigned, we update the weights. Using the deltas and neuron outputs, we compute an adjustment to every weight. We scale that adjustment by the learning rate, and then add it to the current value of the weight to get the new, updated value of the weight, as in in Figure 18.58(d).

18.12.2 What Backprop Doesn't Do

There's a shorthand in some discussions of backprop that can be confusing. Authors sometimes say something like, "backpropagation moves the error backwards from the output layer to the input layer," or "backpropagation uses the error at each layer to find the error at the layer before."

This can be misleading because backprop is *not* "moving the error" at all. In fact, the only time we use "the error" is at the very start of the process when we find the delta values for the output neurons.

What's really going on involves the *change* in the error, which is represented by the delta values. These act as amplifiers of change in the neuron outputs, telling us that if an output value or weight changes by a given amount, we can predict the corresponding change in the error.

So backprop isn't moving "the error" backwards. But *something* is moving backwards. Let's see what that is.

18.12.3 What Backprop Does Do

The first step in the backprop process is to find the deltas for the output neurons. These are found from the **gradient** of the error. We sliced the gradient to get a look at the 2D curve for each prediction, and then we took the derivative of that curve, but that was just for ease of visualization and discussion. The derivatives are just pieces of what really matters: the gradient.

As we work our way backwards, the deltas continue to represent gradients. Every delta value represents a different gradient. For instance, the delta attached to a neuron C describes the gradient of the error with respect to the output of C, and the delta for A describes the gradient of the error with respect to changes in the output of A.

So when we change the weights, we're changing them in order to follow the gradient of the error. This is an example of **gradient descent**, which mimics the path that water takes as it runs downhill on a landscape.

We can say that backpropagation is an algorithm that lets us efficiently update our weights using gradient descent, since the deltas it computes describe that gradient.

So a nice way to summarize backprop is to say that it moves the gradient of the error backwards, modifying it to account for each neuron's contribution.

18.12.4 Keeping Neurons Happy

When we put activation functions back into the backprop algorithm, we concentrated on the region where the inputs are near 0.

That wasn't an accident. Let's return to the sigmoid function, and look at what happens when the value into the activation function (that is, the sum of the weighted inputs) becomes a very large number, say 10 or more. Figure 18.59 shows the sigmoid between values of −20 and 20, along with its derivative in the same range.

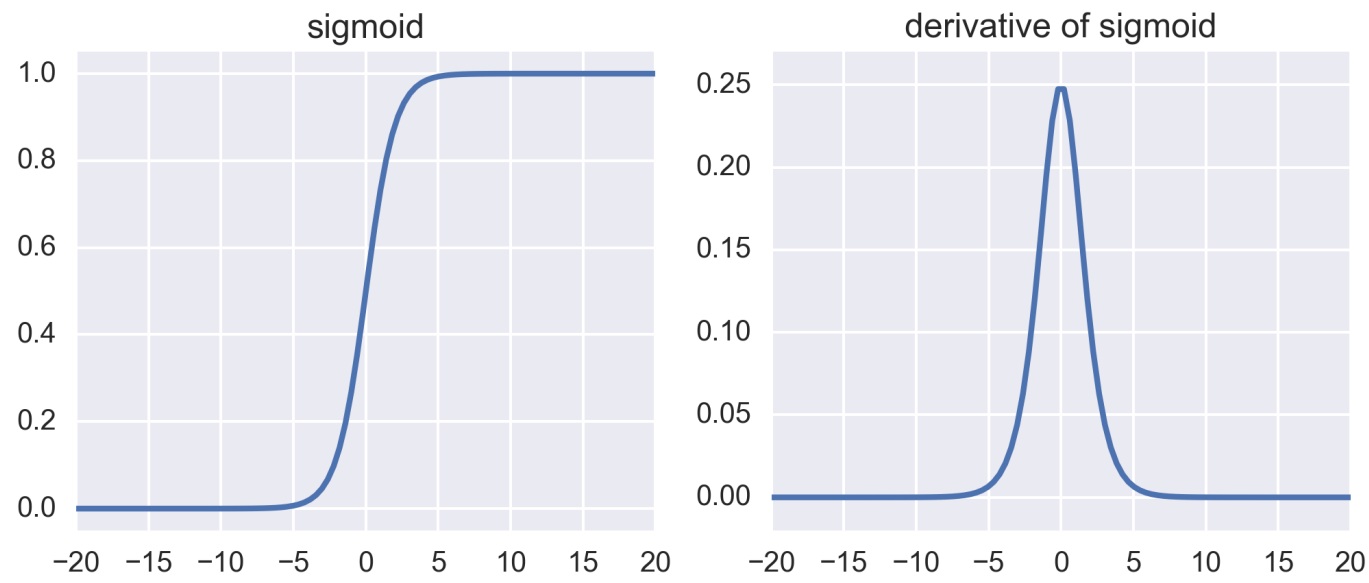

Figure 18.59: The sigmoid function becomes flat for very large positive and negative values. Left: The sigmoid for the range −20 to 20. Right: The derivative of the sigmoid in the same range. Note that the vertical scales of the two plots are different.

The sigmoid never quite reaches exactly 1 or 0 at either end, but it gets extremely close. Similarly, the value of the derivative never quite reaches 0, but as we can see from the graph it gets very close.

Figure 18.60 shows a neuron with a sigmoid activation function. The value going into the function, which we've labeled *z*, has the value 10, putting it in one of the curve's flat regions.

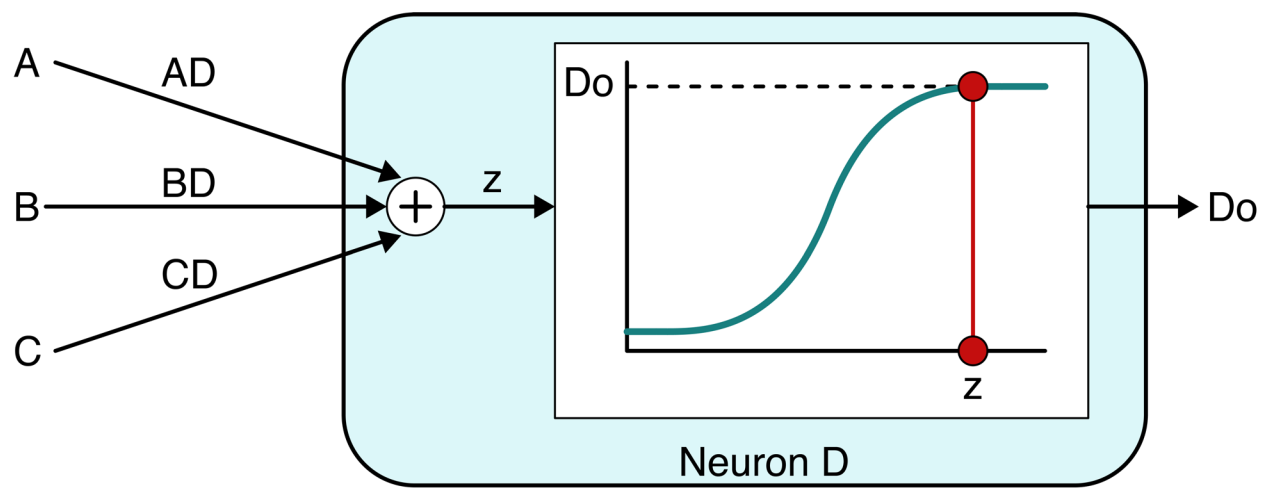

Figure 18.60: When we apply a large value (say 10) to the sigmoid, we find ourselves in a flat region and get back the value of 1.

From Figure 18.59, we can see that the output is basically 1.

Now suppose that we change one of the weights, as shown in Figure 18.61. The value z increases, so we move right on the activation function curve to find our output. Let's say that the new value of z is 15. The output is still basically 1.

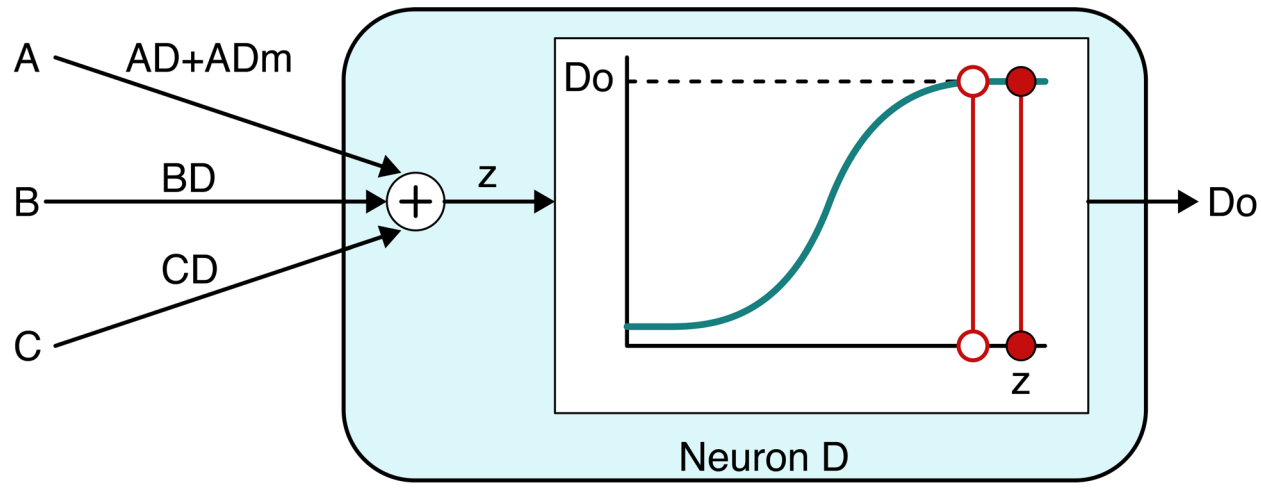

Figure 18.61: A big increase to the weight AD coming into this neuron has no effect on its output, because it just pushes us further to the right along the flat region of the sigmoid. The neuron's output was 1 before we added ADm to the weight AD, and it's still 1 afterwards.

If we increase the value of the incoming weight again, even by a lot, we'll still get back an output of 1. In other words, *changing the incoming weight has no effect on the output*. And because the output doesn't change, the error doesn't change.

We could have predicted this from the derivative in Figure 18.59. When the input is 15, the derivative is 0 (actually about 0.0000003, but our convention above says that we can call that 0). So changing the input will result no change in the output.

This is a terrible situation for any kind of learning, because we've lost the ability to improve the network by adjusting this weight. In fact, *none* of the weights coming into this neuron matter anymore (if we keep the changes small), because any changes to the weighted sum of the inputs, whether they make the sum smaller or larger, still lands us on a flat part of the function and thus there's no change to the output, and no change to the error.

The same problem holds if the input value is very negative, say less than –10. The sigmoid curve is flat in that region also, and the derivative is also essentially zero.

In both of these cases we say that this neuron has **saturated**. Like a sponge that cannot hold any more water, this neuron cannot hold any more input. The output is 1, and unless the weights, the incoming values, or both, move a lot closer to 0, it's going to stay at 1.

The result is that this neuron no longer participates in learning, which is a blow to our system. If this happens to enough neurons, the system could become crippled, learning more slowly than it should, or perhaps even not at all.

A popular way to prevent this problem is to use **regularization**. Recall from Chapter 9 that the goal of regularization is to keep the sizes of the weights small, or close to 0. Among other benefits, this has the value of keeping the sum of the weighted inputs for each neuron also small and close to zero, which puts us in the nice S-shaped part of the activation function. This is where learning happens. In Chapters 23 and 24 we'll see techniques for regularization in deep learning networks.

Saturation can happen with any activation function where the output curve becomes flat for a while (or forever).

Other activation functions can have their own problems. Consider the popular ReLU curve, plotted from –20 to 20 in Figure 18.62.

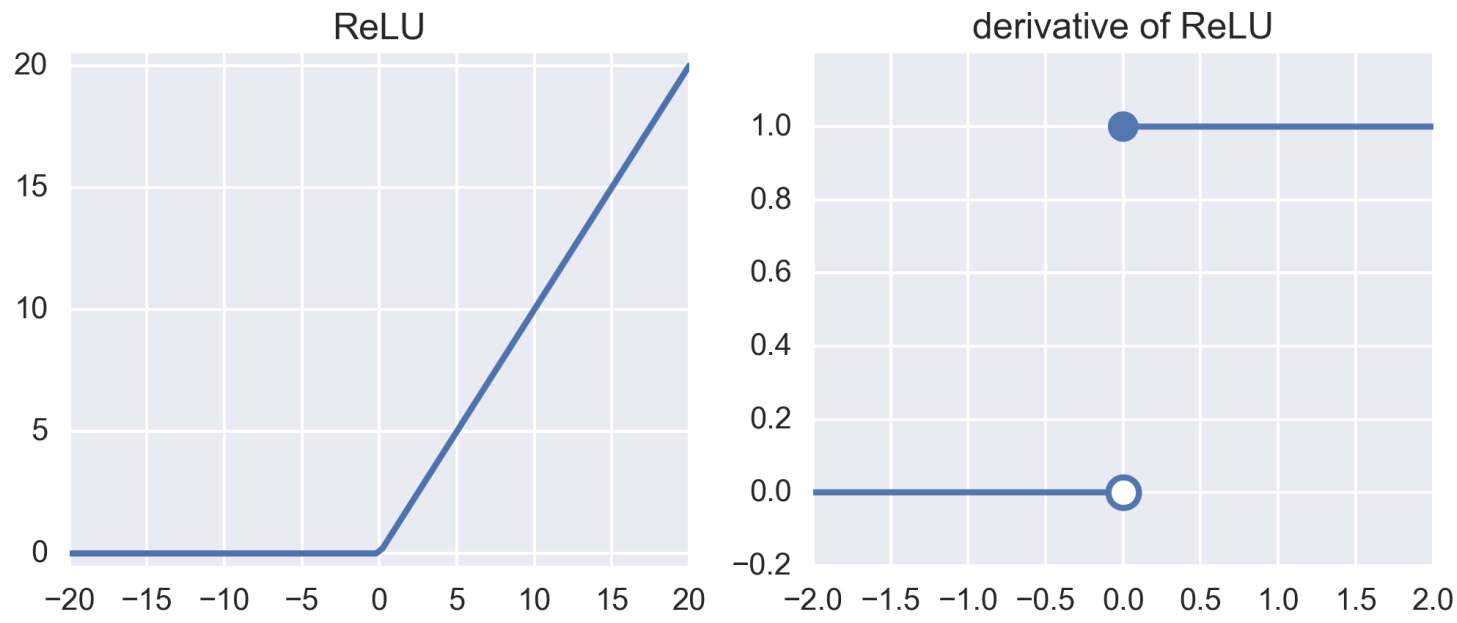

Figure 18.62: The ReLU activation function in the range −20 to 20. Positive values won't saturate the function, but negative values can cause it to die. Left: The ReLU function. Right: The derivative of ReLU.

As long as the input is positive, this function won't saturate, because the output is the same as the input. The derivative for positive inputs is 1, so the sum of the weighted inputs will be passed directly to the output without change.

But when the input is negative, the function's output is 0, and the derivative is 0 as well. Not only do changes make no difference to the output, but the output itself has ceased to make any contribution to the error. The neuron's output is 0 and unless the weights, inputs, or both change by a lot, it's going to stay at 0.

To characterize this dramatic effect, we say that this neuron has **died**, or is now **dead**.

Depending on the initial weights and the first input sample, one or more neurons could die the very first time we perform an update step. Then as training goes on, more neurons can die.

If a lot of neurons die during training, then our network is suddenly working with just a fraction of the neurons we thought it had. That cripples our network. Sometimes even 40% of our neurons can die off during training [Karpathy16].

When we build a neural network we choose the activation functions for each layer based on experience and expectations. In many situations, sigmoids or ReLUs feel like the right function to use, and in many circumstances they work great. But when a network learns slowly, or fails to learn, it pays to look at the neurons and see if some or many are saturated, dying, or dead. If so, we can experiment with our initial starting weights and our learning rate to see if we can avoid the problem. If that doesn't work, we might need to re-structure our network, choose other activation functions, or both.

18.12.5 Mini-Batches

In our discussion above, we followed three steps for every sample: run the sample through the network, calculate all the deltas, and then adjust all the weights.

It turns out that we can save some time, and sometimes even improve our learning, by only adjusting the weights infrequently.

Recall from Chapter 8 that the full training set of samples is sometimes called a **batch** of samples. We can break up that batch into smaller **mini-batches**. Usually the size of our mini-batch is picked to match whatever parallel hardware we have available. For instance, if we our hardware (say a GPU) can evaluate 16 samples simultaneously, then our mini-batch size will be 16. Common mini-batch sizes are 16, 32, and 64, though they can go higher.

The idea is that we run a mini-batch of samples through the network in parallel, and then we compute all the deltas in parallel. We'll average together all the deltas, and use those averages to then perform a single update to the weights. So instead of updating the weights after every sample, they're updated after a mini-batch of 16 samples (or 32, 64, etc.).

This gives us a big increase in speed. It can also improve learning, because the changes to the weights are smoothed out a little by the averaging over the whole mini-batch. This means if there's one weird

sample in the set, it can't pull all the weights in an unwanted direction. The deltas for that weird sample get averaged with the other 31 or 63 samples in the mini-batch, reducing its impact.

18.12.6 Parallel Updates

Since each weight depends only on values from the neurons at its two ends, every weight's update step is completely independent from every other weight's update step.

When we carry out the same steps for independent pieces of data, that's usually our cue to use parallel processing.

And indeed, most modern implementations will, if parallel hardware is available, update all the weights in the network simultaneously. As we just discussed, this update will usually happen after each mini-batch of samples.

This is an enormous time-saver, but it comes at a cost. As we've discussed, changing any weight in the network will change the output value for every neuron that's downstream from that weight. So changes to the weights near the very start of the network can have enormous ripple effects on later neurons, causing them to change their outputs by a lot.

Since the gradients represented by our deltas depend on the values in the network, changing a weight near the input means that we should really re-compute all the deltas for all the neurons that consume that value that weight modifies. That could mean almost every neuron in the network.

This would destroy our ability to update in parallel. It would also make backprop agonizingly slow, since we'd be spending all of our time re-evaluating gradients and computing deltas.

As we've seen, the way to prevent chaos is to use a "small enough" learning rate. If the learning rate is too large, things go haywire and don't settle. If it's too small, we waste a lot of time taking overly tiny

steps. Picking the "just right" value of the learning rate preserves the efficiency of backprop, and our ability to carry out its calculations in parallel.

18.12.7 Why Backprop Is Attractive

A big part of backprop's appeal is that it's so efficient. It's the fastest way that anyone has thought of to figure out how to most beneficially update the weights in a neural network.

As we saw before, and summarized in Figure 18.43, running one step of backprop in a modern library usually takes about as long as evaluating a sample. In other words, consider the time it takes to start with new values in the inputs, and flow that data through the whole network and ultimately to the output layer. Running one step of backprop to compute all the resulting deltas takes about the same amount of time.

That remarkable fact is at the heart of why backprop has become a key workhorse of machine learning, even though we usually have to deal with issues like a fiddly learning rate, saturating neurons, and dying neurons.

18.12.8 Backprop Is Not Guaranteed

It's important to note that there's no guarantee that this scheme is going to learn anything! It's not like the single perceptron of Chapter 10, where we have ironclad proofs that after enough steps, the perceptron will find the dividing line it's looking for.

When we have many thousands of neurons, and potentially many millions of weights, the problem is too complicated to give a rigorous proof that things will always behave as we want.

In fact, things often do go wrong when we first try to train a new network. The network might learn glacially slowly, or even not at all. It might improve for a bit and then seem to suddenly take a wrong turn

and forget everything. All kinds of stuff can happen, which is why many modern libraries offer visualization tools for watching the performance of a network as it learns.

When things go wrong, the first thing many people try is to crank the learning rate to a very small value. If everything settles down, that's a good sign. If the system now appears to be learning, even if it's barely perceptible, that's another good sign. Then we can slowly increase the learning rate until it's learning as quickly as possible without succumbing to chaos.

If that doesn't work, then there might be a problem with the design of the network.

This is a complex problem to deal with. Designing a successful network means making a lot of good choices. For instance, we need to choose the number of layers, the number of neurons on each layer, how the neurons should be connected, what activation functions to use, what learning rate to use, and so on. Getting everything right can be challenging. We usually need a combination of experience, knowledge of our data, and experimentation to design a neural network that will not only learn, but do it efficiently.

In the following chapters we'll see some architectures that have proven to be good starting points for wide varieties of tasks. But each new combination of network and data is its own new thing, and requires thought and patience.

18.12.9 A Little History

When backpropagation was first described in the neural network literature in 1986 it completely changed how people thought about neural networks [Rumelhart86]. The explosion of research and practical benefits that followed were all made possible by this surprisingly efficient technique for finding gradients.

But this wasn't the first time that backprop had been discovered or used. This algorithm, which has been called one of the 30 "great numerical algorithms" [Trefethen15], has been discovered and re-discovered by different people in different fields since at least the 1960's. There are many disciplines that use connected networks of mathematical operations, and finding the derivatives and gradients of those operations at every step is a common and important problem. Clever people who tackled this problem have re-discovered backprop time and again, often giving it a new name each time.

Excellent capsule histories are available online and in print [Griewank12] [Schmidhuber15] [Kurenkov15] [Werbos96]. We'll summarize some of the common threads here. But history can only cover the published literature. There's no knowing how many people have discovered and re-discovered backprop, but didn't publish it.

Perhaps the earliest use of backprop in the form we know it today was published in 1970, when it was used for analyzing the accuracy of numerical calculations [Linnainmaa70], though there was no reference made to neural networks. The process of finding a derivative is sometimes called *differentiation*, so the technique was known as **reverse-mode automatic differentiation**.

It was independently discovered at about the same time by another researcher who was working in chemical engineering [Griewank12].

Perhaps its first explicit investigation for use in neural networks was made in 1974 [Werbos74], but because such ideas were out of fashion, that work wasn't published until 1982 [Schmidhuber15].

Reverse-mode automatic differentiation was used in various sciences for years. But when the classic 1986 paper re-discovered the idea and demonstrated its value to neural networks the idea immediately became a staple of the field under the name **backpropagation** [Rumelhart86].

Backpropagation is central to deep learning, and it forms the foundation for the techniques that we'll be considering in the remainder of this book.

18.12.10 Digging into the Math

This section offers some suggestions for dealing with the math of backpropagation. If you're not interested in that, you can safely skip this section.

Backpropagation is all about manipulations to numbers, hence its description as a "numerical algorithm." That makes it a natural for presenting in a mathematical context.

Even when the equations are stripped down, they can appear formidable [Neilsen15b]. Here are a few hints for getting through the notation and into the heart of the matter.

First, it's essential to master each author's notation. There are a lot of things running around in backpropagation: errors, weights, activation functions, gradients, and so on. Everything will have a name, usually just a single letter. A good first step is to scan through the whole discussion quickly, and notice what names are given to what objects. It often helps to write these down so you don't have to search for their meanings later.

The next step is to work out how these names are used to refer to the different objects. For example, each weight might be written as something like w^l_{jk}, referring to the weight that links neuron number k on layer l to neuron j on layer l+1. This is a lot to pack into one symbol, and when there are several of these things in one equation it can get hard to sort out what's going on.

One way to clear the thickets is to choose values for all the subscripts, and then simplify the equations so each of these highly-indexed terms refers to just one specific thing (such as a single weight). If you think visually, consider drawing pictures showing just what objects are involved, and how their values are being used.

The heart of the backprop algorithm can be thought of, and written as, an application of the **chain rule** from calculus [Karpathy15]. This is an elegant way to describe the way different changes relate to one another, but it requires familiarity with multidimensional calculus. Luckily, there's a wealth of online tutorials and resources designed to help people come up to speed on just this topic [MathCentre09] [Khan13].

We've seen that in practice the computations for outputs, deltas, and weight updates can be performed in parallel. They can also be *written* in a parallel form using the linear algebra language of vectors and matrices. For example, it's common to write the heart of the forward pass (without each neuron's activation function) with a matrix representing the weights between two layers. Then we use that matrix to multiply a vector of the neuron outputs in the previous layer. In the same way, we can write the heart of the backward pass as the transpose of that weight matrix times a vector of the following layer's deltas.

This is a natural formalism, since these computations consist of lots of multiplies followed by additions, which is just what matrix multiplication does for us. And this structure fits nicely onto a GPU, so it's a nice place to start when writing code.

But this linear algebra formalism can obscure the relatively simple steps, because one now has to deal with not just the underlying computation, but its parallel structure in the matrix format, and the proliferation of indices that often comes along with it. We can say that compacting the equations in this form is a type of optimization, where we're aiming for simplicity in both the equations and the algorithms they describe. When learning backprop, people who aren't already very familiar with linear algebra can reasonably feel that this is a form

of *premature optimization*, because (until it is mastered) it obscures, rather than elucidates, the underlying mechanics [Hyde09]. Arguably, only once the backprop algorithm is fully understood should it be rolled up into the more compact matrix form. Thus it may be helpful to either find a presentation that doesn't start with the matrix algebra approach, or try to pull those equations apart into individual operations, rather than big parallel multiplications of matrices and vectors.

Another potential hurdle is that the activation functions (and their derivatives) tend to get presented in different *ad hoc* ways.

To summarize, many authors start their discussions with either the chain rule or matrix forms of the basic equations, so that the equations appear tidy and compact. Then they explain why those equations are useful and correct. Such notation and equations can look daunting, but if we pull them apart to their basics we'll recognize the steps we saw in this chapter. Once we've unpacked these equations and then put them back together, we can see them as natural summaries of an elegant algorithm.

References

[Dauphin14] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, Yoshua Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", 2014. http://arxiv.org/abs/1406.2572

[Fullér10] Robert Fullér, "The Delta Learning Rule Tutorial", Institute for Advanced Management Systems Research, Department of Information Technologies, Åbo Adademi University, 2010. http://uni-obuda.hu/users/fuller.robert/delta.pdf

[Griewank12] Andreas Griewank "Who Invented the Reverse Mode of Differentiation?", Documenta Mathematica, Extra Volume ISMP 389–400, 2012 http://www.math.uiuc.edu/documenta/vol-ismp/52_griewank-andreas-b.pdf

[Hyde09] Randall Hyde, "The Fallacy of Premature Optimization," ACM Ubiquity, 2009. http://ubiquity.acm.org/article.cfm?id=1513451

[Karpathy15] Andrej Karpathy, "Convolutional Neural Networks for Visual Recognition", Stanford CS231n course notes, 2015. http://cs231n.github.io/optimization-2/

[Karpathy16] Andrej Karpathy, "Yes, You Should Understand Backprop", Medium, 2016. https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b

[Khan13] Khan Academy, "Chain rule introduction", 2013. https://www.khanacademy.org/math/ap-calculus-ab/product-quotient-chain-rules-ab/chain-rule-ab/v/chain-rule-introduction

[Kurenkov15] Andrey, Kurenkov, "A 'Brief' History of Neural Nets and Deep Learning, Part 1", 2015. http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning/

[Linnainmaa70] S. Linnainmaa, S., "The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors", Master's thesis, University of Helsinki, 1970.

[MathCentre09] Math Centre, "The Chain Rule", Math Centre report mc-TY-chain-2009-1, 2009. http://www.mathcentre.ac.uk/resources/uploaded/mc-ty-chain-2009-1.pdf

[NASA12] NASA, "Astronomers Predict Titanic Collision: Milky Way vs. Andromeda", NASA Science Blog, Production editor Dr. Tony Phillips, 2012. https://science.nasa.gov/science-news/science-at-nasa/2012/31may_andromeda

[Neilsen15a] Michael A. Nielsen, "Using Neural Networks to Recognize Handwritten Digits", Determination Press, 2015. http://neuralnetworksanddeeplearning.com/chap1.html

[Neilsen15b] Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015. http://neuralnetworksanddeeplearning.com/chap2.html

[Rumelhart86] D.E. Rumelhart, G.E. Hinton, R.J. Williams, “Learning Internal Representations by Error Propagation”, in “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1”, pp. 318-362, 1986. http://www.cs.toronto.edu/~fritz/absps/pdp8.pdf

[Schmidhuber15] Jürgen Schmidhuber, “Who Invented Backpropagation?”, Blog post, 2015. http://people.idsia.ch/~juergen/who-invented-backpropagation.html

[Seung05] Sebastian Seung, “Introduction to Neural Networks”, MIT 9.641J course notes, 2005. https://ocw.mit.edu/courses/brain-and-cognitive-sciences/9-641j-introduction-to-neural-networks-spring-2005/lecture-notes/lec19_delta.pdf

[Trefethen15] Nick Trefethen, “Who Invented the Great Numerical Algorithms?” Oxford Mathematical Institute, 2015. https://people.maths.ox.ac.uk/trefethen/inventorstalk.pdf

[Werbos74] P. Werbos, “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”, PhD thesis, Harvard University, Cambridge, MA, 1974.

[Werbos96] Paul John Werbos, “The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting”, Wiley-Interscience, 1994.

[Wikipedia17] Wikipedia, “Goldilocks and the Three Bears”, 2017. https://en.wikipedia.org/wiki/Goldilocks_and_the_Three_Bears