

Improving the way neural networks learn

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

The techniques we'll develop in this chapter include: a better choice of cost function, known as [the cross-entropy cost function](#); four so-called "[regularization](#)" methods (L1 and L2 regularization, dropout, and artificial expansion of the training data), which make our networks better at generalizing beyond the training data; a [better method for initializing the weights](#) in the network; and a [set of heuristics to help choose good hyper-parameters](#) for the network. I'll also overview [several other techniques](#) in less depth.

The discussions are largely independent of one another, and so you may jump ahead if you wish. We'll also [implement](#) many of the techniques in running code, and use them to improve the results obtained on the handwriting classification problem studied in [Chapter 1](#).

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

The cross-entropy cost function

Most of us find it unpleasant to be wrong. Soon after beginning to learn the piano I gave my first performance before an audience. I was nervous, and began playing the piece an octave too low. I got

Neural Networks and Deep Learning

What this book is about

On the exercises and problems

- ▶ Using neural nets to recognize handwritten digits
- ▶ How the backpropagation algorithm works
- ▶ Improving the way neural networks learn
- ▶ A visual proof that neural nets can compute any function
- ▶ Why are deep neural networks hard to train?
- ▶ Deep learning
- Appendix: Is there a *simple* algorithm for intelligence?
- Acknowledgements
- Frequently Asked Questions

If you benefit from the book, please make a small donation. I suggest \$5, but you can choose the amount.

Donate



Alternately, you can make a donation by sending me Bitcoin, at address

1Kd6tXH5SDAmiFb49J9hknG5pqj7KStSAX

Sponsors



Deep Learning Workstations starting at \$6,999: [learn more](#)



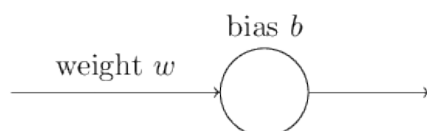
Deep Learning Workstations, Servers, and Laptops



Thanks to all the [supporters](#) who made the book possible, with especial thanks to Pavel Dudrenov. Thanks also to all the contributors to the [Bugfinder Hall of Fame](#).

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

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



We'll train this neuron to do something ridiculously easy: take the input 1

to the output 0

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

Winnovative PDF Tools Demo

To make things definite, I'll pick the initial weight to be 0.6 and the initial bias to be 0.9

. These are generic choices used as a place to begin learning, I wasn't picking them to be special in any way. The initial output from the neuron is 0.82

, so quite a bit of learning will be needed before our neuron gets near the desired output, 0.0

. Click on "Run" in the bottom right corner below to see how the neuron learns an output much closer to 0.0

. Note that this isn't a pre-recorded animation, your browser is actually computing the gradient, then using the gradient to update the weight and bias, and displaying the result. The learning rate is $\eta = 0.15$

, which turns out to be slow enough that we can follow what's happening, but fast enough that we can get substantial learning in just a few seconds. The cost is the quadratic cost function, C , introduced back in Chapter 1. I'll remind you of the exact form of the cost function shortly, so there's no need to go and dig up the definition. Note that you can run the animation multiple times by

Resources

[Michael Nielsen on Twitter](#)

[Book FAQ](#)

[Code repository](#)

[Michael Nielsen's project announcement mailing list](#)

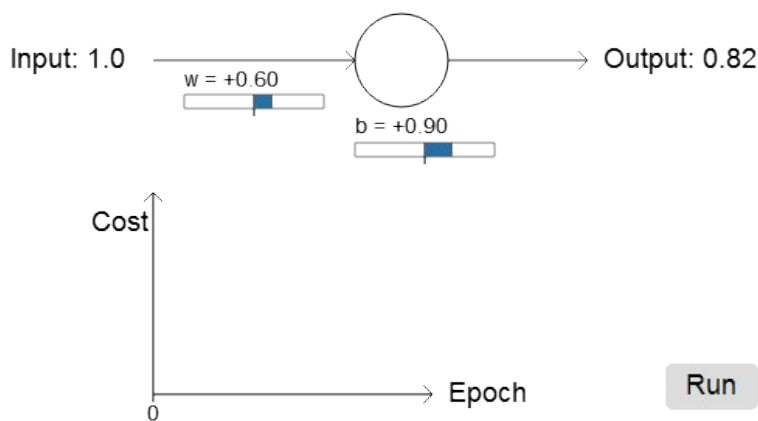
[Deep Learning](#), book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

cognitivemedium.com



By [Michael Nielsen](#) / Jun 2019

clicking on "Run" again.



As you can see, the neuron rapidly learns a weight and bias that drives down the cost, and gives an output from the neuron of about 0.09

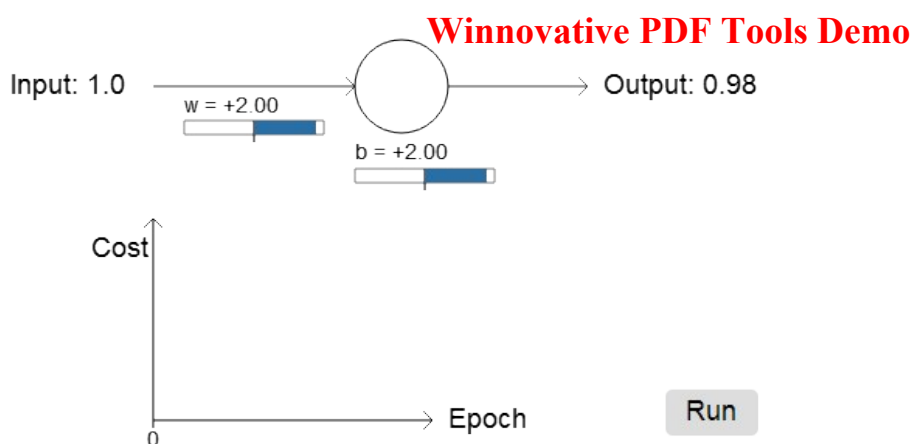
. That's not quite the desired output, 0.0

, but it is pretty good. Suppose, however, that we instead choose both the starting weight and the starting bias to be 2.0

. In this case the initial output is 0.98

, which is very badly wrong. Let's look at how the neuron learns to output 0

in this case. Click on "Run" again:



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

. This behaviour is strange when contrasted to human learning. As I said at the beginning of this section, we often learn fastest when we're badly wrong about something. But we've just seen that our artificial neuron has a lot of difficulty learning when it's badly

wrong - far more difficulty than when it's just a little wrong.

What's more, it turns out that this behaviour occurs not just in this toy model, but in more general networks. Why is learning so slow? And can we find a way of avoiding this slowdown?

To understand the origin of the problem, consider that our neuron learns by changing the weight and bias at a rate determined by the partial derivatives of the cost function, $\partial C / \partial w$ and $\partial C / \partial b$

. So saying "learning is slow" is really the same as saying that those partial derivatives are small. The challenge is to understand why they are small. To understand that, let's compute the partial derivatives. Recall that we're using the quadratic cost function, which, from Equation (6), is given by

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

where a

is the neuron's output when the training input $x = 1$

is used, and $y = 0$

is the corresponding desired output. To write this more explicitly

in terms of the weight and bias, recall that $a = \sigma(z)$

, where $z = wx + b$

Winnovative PDF Tools Demo

. Using the chain rule to differentiate with respect to the weight and bias we get

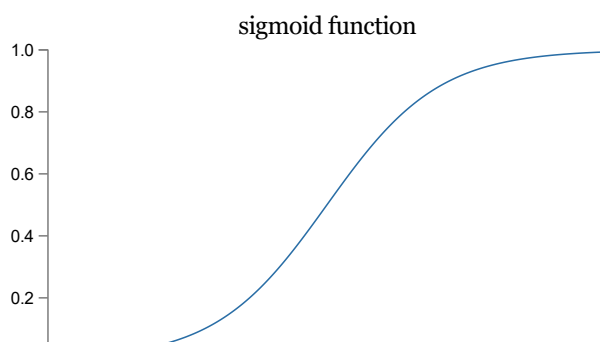
$$\begin{aligned}\frac{\partial C}{\partial w} &= (a - y)\sigma'(z)x = a\sigma'(z) \\ \frac{\partial C}{\partial b} &= (a - y)\sigma'(z) = a\sigma'(z),\end{aligned}$$

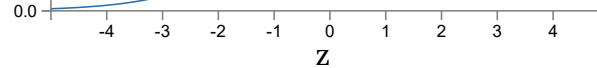
where I have substituted $x = 1$

and $y = 0$

. To understand the behaviour of these expressions, let's look more closely at the $\sigma'(z)$

term on the right-hand side. Recall the shape of the σ function:



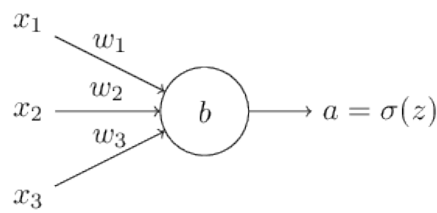


We can see from this graph that when the neuron's output is close to 1, the curve gets very flat, and so $\sigma'(z)$ gets very small. Equations (55) and (56) then tell us that $\partial C / \partial w$ and $\partial C / \partial b$ get very small. This is the origin of the learning slowdown. What's more, as we shall see a little later, the learning slowdown occurs for essentially the same reason in more general neural networks, not just the toy example we've been playing with.

Introducing the cross-entropy cost function

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

Winnovative PDF Tools Demo



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

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

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

It's not obvious that the expression (57) fixes the learning slowdown problem. In fact, frankly, it's not even obvious that it

makes sense to call this a cost function! Before addressing the learning slowdown, let's see in what sense the cross-entropy can be interpreted as a cost function.

Two properties in particular make it reasonable to interpret the cross-entropy as a cost function. First, it's non-negative, that is, $C > 0$

. To see this, notice that: (a) all the individual terms in the sum in (57) are negative, since both logarithms are of numbers in the range 0 to 1; and (b) there is a minus sign out the front of the sum.

Second, if the neuron's actual output is close to the desired output for all training inputs, x , then the cross-entropy will be close to zero*. To see this, suppose for example that $y = 0$ and $a \approx 0$ for some input x . This is a case when the neuron is doing a good job on that input. We see that the first term in the expression (57) for the cost vanishes, since $y = 0$

, while the second term is just $-\ln(1 - a) \approx 0$.

A similar analysis holds when $y = 1$ and $a \approx 1$

. And so the contribution to the cost will be low provided the actual output is close to the desired output.

Summing up, the cross-entropy is positive, and tends toward zero as the neuron gets better at computing the desired output, y , for all training inputs, x

. These are both properties we'd intuitively expect for a cost function. Indeed, both properties are also satisfied by the quadratic cost. So that's good news for the cross-entropy. But the cross-entropy cost function has the benefit that, unlike the quadratic cost, it avoids the problem of learning slowing down. To see this, let's compute the partial derivative of the cross-entropy cost with respect to the weights. We substitute $a = \sigma(z)$ into (57), and apply the chain rule twice, obtaining:

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

*To prove this I will need to assume that the desired outputs y are all either 0 or 1

. This is usually the case when solving classification problems, for example, or when computing Boolean functions. To understand what happens when we don't make this assumption, see the exercises at the end of this section.

Innovative PDF Tools Demo

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

Putting everything over a common denominator and simplifying this becomes:

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

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

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

This is a beautiful expression. It tells us that the rate at which the weight learns is controlled by $\sigma(z) - y$, i.e., by the error in the output. The larger the error, the faster the neuron will learn. This is just what we'd intuitively expect. In particular, it avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost, Equation (55). When we use the cross-entropy, the $\sigma'(z)$ term gets canceled out, and we no longer need worry about it being small. This cancellation is the special miracle ensured by the cross-entropy cost function. Actually, it's not really a miracle. As we'll see later, the cross-entropy was specially chosen to have just this property.

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

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

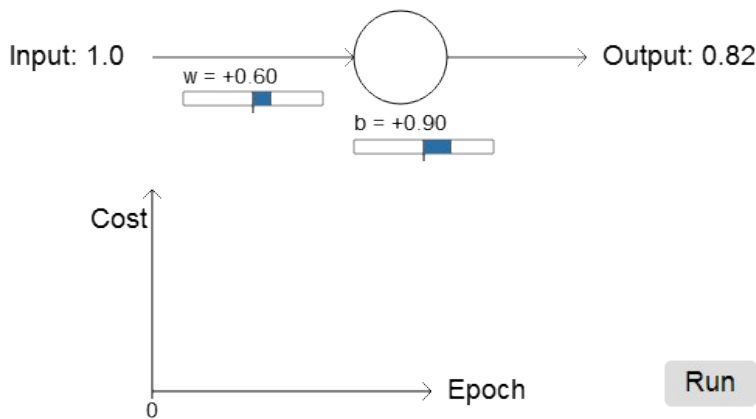
Again, this avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost, Equation (56).

Exercise

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

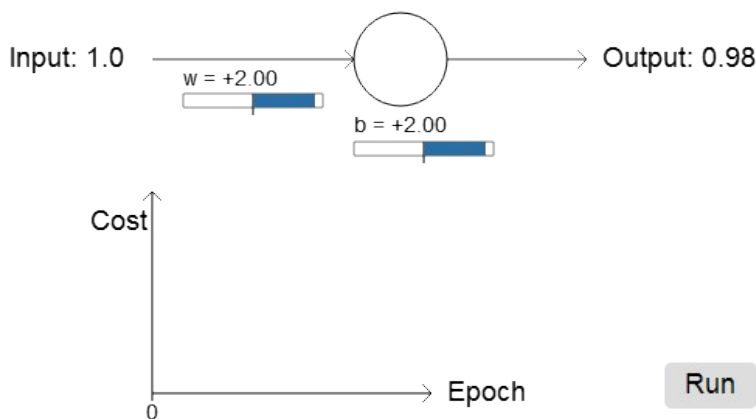
Let's return to the toy example we played with earlier, and explore what happens when we use the cross-entropy instead of the quadratic cost. To re-orient ourselves, we'll begin with the case where the quadratic cost did just fine, with starting weight 0.6 and starting bias 0.9

. Press "Run" to see what happens when we replace the quadratic cost by the cross-entropy:



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

:



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

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

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

Winnovative PDF Tools Demo

We've been studying the cross-entropy for a single neuron. However, it's easy to generalize the cross-entropy to many-neuron multi-layer networks. In particular, suppose $y = y_1, y_2, \dots$ are the desired values at the output neurons, i.e., the neurons in the final layer, while a_1^L, a_2^L, \dots are the actual output values. Then we define the cross-entropy by

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right].$$

This is the same as our earlier expression, Equation (57), except now we've got the \sum_j summing over all the output neurons. I won't explicitly work through a derivation, but it should be plausible that using the expression (63) avoids a learning slowdown in many-neuron networks. If you're interested, you can work through the derivation in the problem below.

Incidentally, I'm using the term "cross-entropy" in a way that has confused some early readers, since it superficially appears to conflict with other sources. In particular, it's common to define the

cross-entropy for two probability distributions, p_j and q_j , as $\sum_j p_j \ln q_j$. This definition may be connected to (57), if we treat a single sigmoid neuron as outputting a probability distribution consisting of the neuron's activation a and its complement $1 - a$.

However, when we have many sigmoid neurons in the final layer, the vector a_j^L of activations don't usually form a probability distribution. As a result, a definition like $\sum_j p_j \ln q_j$ doesn't even make sense, since we're not working with probability distributions. Instead, you can think of (63) as a summed set of per-neuron cross-entropies, with the activation of each neuron being interpreted as part of a two-element probability distribution*. In this sense, (63) is a generalization of the cross-entropy for probability distributions.

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

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

Exercises

- One gotcha with the cross-entropy is that it can be difficult at first to remember the respective roles of the y s and the a s. It's easy to get confused about whether the right form is $-[y \ln a + (1 - y) \ln(1 - a)]$ or $-[a \ln y + (1 - a) \ln(1 - y)]$. What happens to the second of these expressions when $y = 0$ or 1

? Does this problem afflict the first expression? Why or why not?

- In the single-neuron discussion at the start of this section, I argued that the cross-entropy is small if $\sigma(z) \approx y$ for all training inputs. The argument relied on y being equal to either 0 or 1.
• This is usually true in classification problems, but for other problems (e.g., regression problems) y can sometimes take values intermediate between 0 and 1.
• Show that the cross-entropy is still minimized when $\sigma(z) = y$ for all training inputs. When this is the case the cross-entropy has the value:

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

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

Problems

Winnovative PDF Tools Demo

- **Many-layer multi-neuron networks** In the notation introduced in the [last chapter](#), show that for the quadratic cost the partial derivative with respect to weights in the output layer is

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n} \sum_x a_k^{L-1} (a_j^L - y_j) \sigma'(z_j^L).$$

The term $\sigma'(z_j^L)$ causes a learning slowdown whenever an output neuron saturates on the wrong value. Show that for the cross-entropy cost the output error δ^L for a single training example x is given by

$$\delta^L = a^L - y.$$

Use this expression to show that the partial derivative with respect to the weights in the output layer is given by

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n} \sum_x a_k^{L-1} (a_j^L - y_j).$$

The $\sigma'(z_j^L)$

term has vanished, and so the cross-entropy avoids the problem of learning slowdown, not just when used with a single neuron, as we saw earlier, but also in many-layer multi-neuron networks. A simple variation on this analysis holds also for the biases. If this is not obvious to you, then you should work through that analysis as well.

- **Using the quadratic cost when we have linear neurons**

in the output layer Suppose that we have a many-layer multi-neuron network. Suppose all the neurons in the final layer are *linear neurons*, meaning that the sigmoid activation function is not applied, and the outputs are simply $a_j^L = z_j^L$. Show that if we use the quadratic cost function then the output error δ^L for a single training example x is given by

$$\delta^L = a^L - y.$$

Similarly to the previous problem, use this expression to show that the partial derivatives with respect to the weights and biases in the output layer are given by

Innovative PDF Tools Demo

$$\begin{aligned}\frac{\partial C}{\partial w_{jk}^L} &= \frac{1}{n_x} \sum_x a_k^{L-1} (a_j^L - y_j) \\ \frac{\partial C}{\partial b_j^L} &= \frac{1}{n_x} \sum_x (a_j^L - y_j).\end{aligned}$$

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

Using the cross-entropy to classify MNIST digits

The cross-entropy is easy to implement as part of a program which learns using gradient descent and backpropagation. We'll do that [later in the chapter](#), developing an improved version of our [earlier program](#) for classifying the MNIST handwritten digits, `network.py`. The new program is called `network2.py`, and incorporates not just the cross-entropy, but also several other techniques developed in this chapter*. For now, let's look at how well our new program classifies MNIST digits. As was the case in Chapter 1, we'll use a

*The code is available on [GitHub](#).

network with 30

hidden neurons, and we'll use a mini-batch size of 10

. We set the learning rate to $\eta = 0.5$

* and we train for 30

epochs. The interface to `network2.py` is slightly different than

`network.py`, but it should still be clear what is going on. You can, by

the way, get documentation about `network2.py`'s interface by using

commands such as `help(network2.Network.SGD)` in a Python shell.

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

Note, by the way, that the `net.large_weight_initializer()`

command is used to initialize the weights and biases in the same

way as described in Chapter 1. We need to run this command

because later in this chapter we'll change the default weight

initialization in our networks. The result from running the above

sequence of commands is a network with 95.49

percent accuracy. This is pretty close to the result we obtained in

Chapter 1, 95.42

percent, using the quadratic cost.

Winnovative PDF Tools Demo

Let's look also at the case where we use 100

hidden neurons, the cross-entropy, and otherwise keep the

parameters the same. In this case we obtain an accuracy of 96.82

percent. That's a substantial improvement over the results from

Chapter 1, where we obtained a classification accuracy of 96.59

percent, using the quadratic cost. That may look like a small

change, but consider that the error rate has dropped from 3.41

percent to 3.18

percent. That is, we've eliminated about one in fourteen of the

original errors. That's quite a handy improvement.

It's encouraging that the cross-entropy cost gives us similar or

better results than the quadratic cost. However, these results don't

conclusively prove that the cross-entropy is a better choice. The

reason is that I've put only a little effort into choosing hyper-

parameters such as learning rate, mini-batch size, and so on. For

the improvement to be really convincing we'd need to do a

thorough job optimizing such hyper-parameters. Still, the results

are encouraging, and reinforce our earlier theoretical argument

*In Chapter 1 we used the quadratic cost and a learning rate of $\eta = 3.0$

. As discussed above, it's not possible to say precisely what it means to use the "same" learning rate when the cost function is changed. For both cost functions I experimented to find a learning rate that provides near-optimal performance, given the other hyper-parameter choices.

There is, incidentally, a very rough general heuristic for relating the learning rate for the cross-entropy and the quadratic cost. As we saw earlier, the gradient terms for the quadratic cost have an extra $\sigma' = \sigma(1 - \sigma)$ term in them. Suppose we average this over values for σ

$$, \int_0^1 d\sigma \sigma(1 - \sigma) = 1/6$$

. We see that (very roughly) the quadratic cost learns an average of 6

times slower, for the same learning rate.

This suggests that a reasonable starting point is to divide the learning rate for the quadratic cost by 6

. Of course, this argument is far from rigorous, and shouldn't be taken too seriously. Still, it can sometimes be a useful starting point.

that the cross-entropy is a better choice than the quadratic cost.

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

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

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

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

Let's begin with the last of these questions: what could have motivated us to think up the cross-entropy in the first place? Suppose we'd discovered the learning slowdown described earlier, and understood that the origin was the $\sigma'(z)$ terms in Equations (55) and (56). After staring at those equations for a bit, we might wonder if it's possible to choose a cost function

so that the $\sigma'(z)$

term disappeared. In that case, the cost $C = C_x$

for a single training example x

would satisfy

$$\begin{aligned}\frac{\partial C}{\partial w_j} &= x_j(a - y) \\ \frac{\partial C}{\partial b} &= (a - y).\end{aligned}$$

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

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

Using $\sigma'(z) = \sigma(z)(1 - \sigma(z)) = a(1 - a)$

the last equation becomes

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

Comparing to Equation (72) we obtain

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

Integrating this expression with respect to a gives

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

for some constant of integration. This is the contribution to the cost from a single training example, x

. To get the full cost function we must average over training examples, obtaining

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

where the constant here is the average of the individual constants for each training example. And so we see that Equations (71) and (72) uniquely determine the form of the cross-entropy, up to an overall constant term. The cross-entropy isn't something that was

miraculously pulled out of thin air. Rather, it's something that we could have discovered in a simple and natural way.

What about the intuitive meaning of the cross-entropy? How should we think about it? Explaining this in depth would take us further afield than I want to go. However, it is worth mentioning that there is a standard way of interpreting the cross-entropy that comes from the field of information theory. Roughly speaking, the idea is that the cross-entropy is a measure of surprise. In particular, our neuron is trying to compute the function $x \rightarrow y = y(x)$

. But instead it computes the function $x \rightarrow a = a(x)$

. Suppose we think of a

as our neuron's estimated probability that y is 1

, and $1 - a$

is the estimated probability that the right value for y is 0

. Then the cross-entropy measures how "surprised" we are, on average, when we learn the true value for y

. We get low surprise if the output is what we expect, and high surprise if the output is unexpected. Of course, I haven't said exactly what "surprise" means, and so this perhaps seems like empty verbiage. But in fact there is a precise information-theoretic way of saying what is meant by surprise. Unfortunately, I don't know of a good, short, self-contained discussion of this subject that's available online. But if you want to dig deeper, then Wikipedia contains a [brief summary](#) that will get you started down the right track. And the details can be filled in by working through the materials about the Kraft inequality in chapter 5 of the book about information theory by [Cover and Thomas](#).

Problem

- We've discussed at length the learning slowdown that can occur when output neurons saturate, in networks using the quadratic cost to train. Another factor that may inhibit learning is the presence of the x_j term in Equation (61). Because of this term, when an input x_j is near to zero, the corresponding weight w_j will learn slowly. Explain why it is not possible to eliminate the x_j term through a clever choice of cost function.

Softmax

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

The idea of softmax is to define a new type of output layer for our neural networks. It begins in the same way as with a sigmoid layer, by forming the weighted inputs* $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$. However, we don't apply the sigmoid function to get the output. Instead, in a softmax layer we apply the so-called *softmax function* to the z_j^L . According to this function, the activation a_j^L of the j th output neuron is

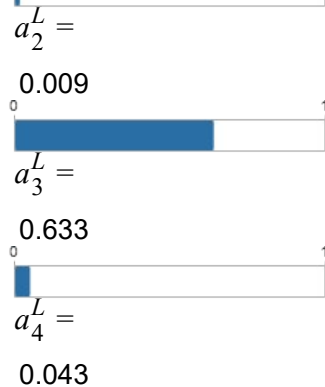
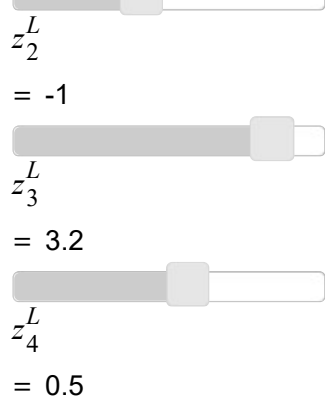
$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}},$$

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

If you're not familiar with the softmax function, Equation (78) may look pretty opaque. It's certainly not obvious why we'd want to use this function. And it's also not obvious that this will help us address the learning slowdown problem. To better understand Equation (78), suppose we have a network with four output neurons, and four corresponding weighted inputs, which we'll denote z_1^L, z_2^L, z_3^L , and z_4^L . Shown below are adjustable sliders showing possible values for the weighted inputs, and a graph of the corresponding output activations. A good place to start exploration is by using the bottom slider to increase z_4^L :



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



As you increase z_4^L

, you'll see an increase in the corresponding output activation, a_4^L

, and a decrease in the other output activations. Similarly, if you

decrease z_4^L

then a_4^L

will decrease, and all the other output activations will increase. In

fact, if you look closely, you'll see that in both cases the total

change in the other activations exactly compensates for the change

in a_4^L

. The reason is that the output activations are guaranteed to always

sum up to 1

, as we can prove using Equation (78) and a little algebra:

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

As a result, if a_4^L

increases, then the other output activations must decrease by the

same total amount, to ensure the sum over all activations remains

1

. And, of course, similar statements hold for all the other

activations.

Equation (78) also implies that the output activations are all

positive, since the exponential function is positive. Combining this

with the observation in the last paragraph, we see that the output

from the softmax layer is a set of positive numbers which sum up

to 1

. In other words, the output from the softmax layer can be thought

of as a probability distribution.

The fact that a softmax layer outputs a probability distribution is

rather pleasing. In many problems it's convenient to be able to

interpret the output activation a_j^L

as the network's estimate of the probability that the correct output

is j

. So, for instance, in the MNIST classification problem, we can interpret a_j^L as the network's estimated probability that the correct digit classification is j

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

Exercise

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

We're starting to build up some feel for the softmax function and the way softmax layers behave. Just to review where we're at: the exponentials in Equation (78) ensure that all the output activations are positive. And the sum in the denominator of Equation (78) ensures that the softmax outputs sum to 1. So that particular form no longer appears so mysterious: rather, it is a natural way to ensure that the output activations form a probability distribution. You can think of softmax as a way of rescaling the z_j^L , and then squishing them together to form a probability distribution.

Exercises

- **Monotonicity of softmax** Show that $\partial a_j^L / \partial z_k^L$ is positive if $j = k$ and negative if $j \neq k$. As a consequence, increasing z_j^L is guaranteed to increase the corresponding output activation, a_j^L , and will decrease all the other output activations. We already saw this empirically with the sliders, but this is a rigorous proof.

- **Non-locality of softmax** A nice thing about sigmoid layers is that the output a_j^L is a function of the corresponding weighted input, $a_j^L = \sigma(z_j^L)$. Explain why this is not the case for a softmax layer: any particular output activation a_j^L depends on *all* the weighted inputs.

Problem

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

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

$$C \equiv -\ln a_y^L.$$

So, for instance, if we're training with MNIST images, and input an image of a 7

, then the log-likelihood cost is $-\ln a_7^L$

. To see that this makes intuitive sense, consider the case when the network is doing a good job, that is, it is confident the input is a 7

. In that case it will estimate a value for the corresponding probability a_7^L

which is close to 1

, and so the cost $-\ln a_7^L$

will be small. By contrast, when the network isn't doing such a good job, the probability a_7^L

will be smaller, and the cost $-\ln a_7^L$

will be larger. So the log-likelihood cost behaves as we'd expect a cost function to behave.

What about the learning slowdown problem? To analyze that,

recall that the key to the learning slowdown is the behaviour of the quantities $\partial C / \partial w_{jk}^L$ and $\partial C / \partial b_j^L$

. I won't go through the derivation explicitly - I'll ask you to do in the problems, below - but with a little algebra you can show that*

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

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

These equations are the same as the analogous expressions obtained in our earlier analysis of the cross-entropy. Compare, for example, Equation (82) to Equation (67). It's the same equation, albeit in the latter I've averaged over training instances. And, just as in the earlier analysis, these expressions ensure that we will not encounter a learning slowdown. In fact, it's useful to think of a softmax output layer with log-likelihood cost as being quite similar to a sigmoid output layer with cross-entropy cost.

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

Problems

- Derive Equations (81) and (82).
- **Where does the "softmax" name come from?** Suppose we change the softmax function so the output activations are given by

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

where c

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

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

- **Backpropagation with softmax and the log-likelihood**

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

$$\delta_j^L = a_j^L - y_j.$$

Winnovative PDF Tools Demo

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

Overfitting and regularization

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

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

The point, of course, is that models with a large number of free parameters can describe an amazingly wide range of phenomena. Even if such a model agrees well with the available data, that doesn't make it a good model. It may just mean there's enough freedom in the model that it can describe almost any data set of

the given size, without capturing any genuine insights into the underlying phenomenon. When that happens the model will work well for the existing data, but will fail to generalize to new situations. The true test of a model is its ability to make predictions in situations it hasn't been exposed to before.

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

Let's sharpen this problem up by constructing a situation where our network does a bad job generalizing to new situations. We'll use our 30 hidden neuron network, with its 23,860 parameters. But we won't train the network using all 50,000 MNIST training images. Instead, we'll use just the first 1,000 training images. Using that restricted set will make the problem with generalization much more evident. We'll train in a similar way to before, using the cross-entropy cost function, with a learning rate of $\eta = 0.5$ and a mini-batch size of 10

. However, we'll train for 400 epochs, a somewhat longer number than before, because we're not using as many training examples.

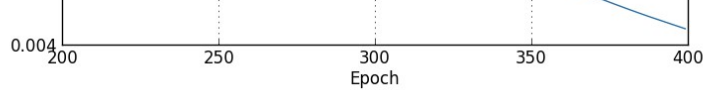
Let's use `network2` to look at the way the cost function changes:

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

Using the results we can plot the way the cost changes as the network learns* :

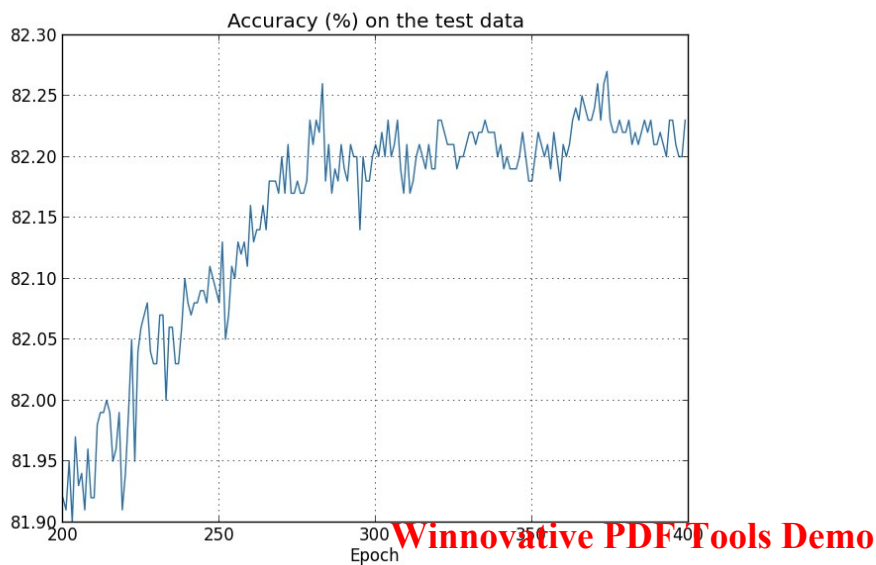
*This and the next four graphs were generated by the program [overfitting.py](#).





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

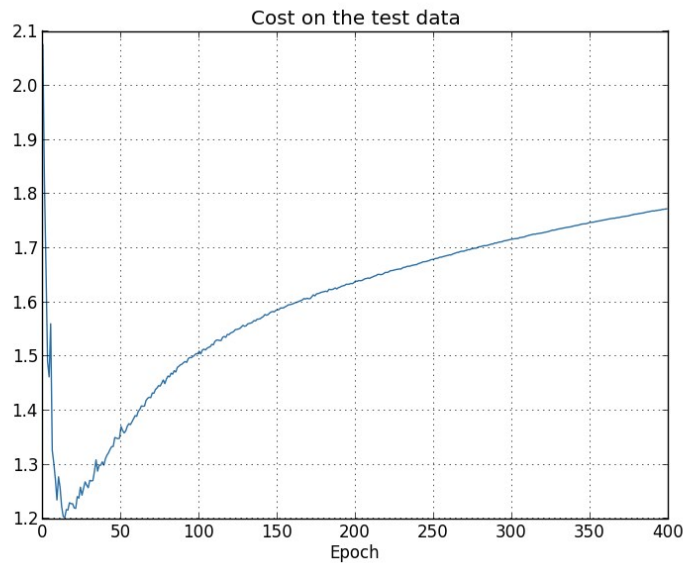
Let's now look at how the classification accuracy on the test data changes over time:



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

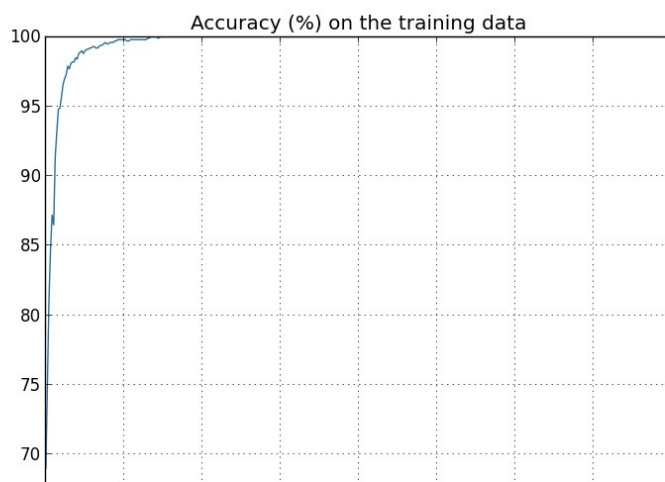
You might wonder if the problem here is that I'm looking at the *cost* on the training data, as opposed to the *classification accuracy* on the test data. In other words, maybe the problem is that we're making an apples and oranges comparison. What would happen if we compared the cost on the training data with the cost on the test

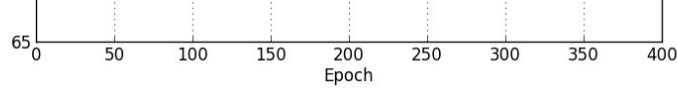
data, so we're comparing similar measures? Or perhaps we could compare the classification accuracy on both the training data and the test data? In fact, essentially the same phenomenon shows up no matter how we do the comparison. The details do change, however. For instance, let's look at the cost on the test data:



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

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





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

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

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

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

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

Up to now we've been using the `training_data` and `test_data`, and ignoring the `validation_data`. The `validation_data` contains 10,000 images of digits, images which are different from the 50,000 images in the MNIST training set, and the 10,000 images in the MNIST test set. Instead of using the `test_data` to prevent overfitting, we will use the `validation_data`. To do this, we'll use much the same strategy as was described above for the `test_data`. That is, we'll compute the classification accuracy on the `validation_data` at the end of each epoch. Once the classification accuracy on the `validation_data` has saturated, we stop training.

This strategy is called *early stopping*. Of course, in practice we won't immediately know when the accuracy has saturated. Instead, we continue training until we're confident that the accuracy has saturated*.

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

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

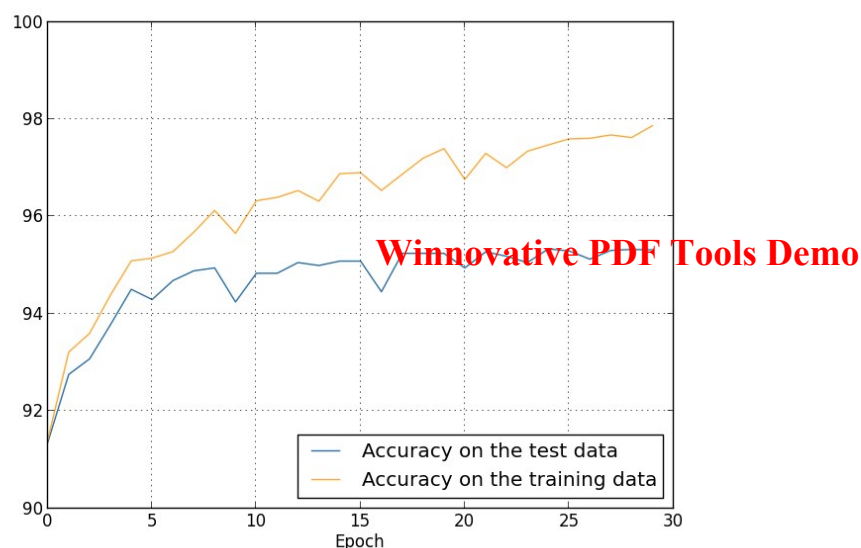
Now, in practice, even after evaluating performance on the `test_data` we may change our minds and want to try another approach - perhaps a different network architecture - which will involve finding a new set of hyper-parameters. If we do this, isn't there a danger we'll end up overfitting to the `test_data` as well? Do we need a potentially infinite regress of data sets, so we can be

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

Innovative PDF Tools Demo

confident our results will generalize? Addressing this concern fully is a deep and difficult problem. But for our practical purposes, we're not going to worry too much about this question. Instead, we'll plunge ahead, using the basic hold out method, based on the `training_data`, `validation_data`, and `test_data`, as described above.

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



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

Regularization

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

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

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

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

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

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

In both cases we can write the regularized cost function as

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

where C_0 is the original, unregularized cost function.

Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowed if they considerably improve the first

part of the cost function. Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function. The relative importance of the two elements of the compromise depends on the value of λ : when λ is small we prefer to minimize the original cost function, but when λ is large we prefer small weights.

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

To construct such an example, we first need to figure out how to apply our stochastic gradient descent learning algorithm in a regularized neural network. In particular, we need to know how to compute the partial derivatives $\partial C / \partial w$ and $\partial C / \partial b$ for all the weights and biases in the network. Taking the partial derivatives of Equation (87) gives

$$\begin{aligned}\frac{\partial C}{\partial w} &= \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \\ \frac{\partial C}{\partial b} &= \frac{\partial C_0}{\partial b}.\end{aligned}$$

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

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

The learning rule for the weights becomes:

$$w \rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w$$

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

This is exactly the same as the usual gradient descent learning rule, except we first rescale the weight w

by a factor $1 - \frac{\eta\lambda}{n}$

. This rescaling is sometimes referred to as *weight decay*, since it makes the weights smaller. At first glance it looks as though this means the weights are being driven unstoppably toward zero. But that's not right, since the other term may lead the weights to increase, if so doing causes a decrease in the unregularized cost function.

Okay, that's how gradient descent works. What about stochastic gradient descent? Well, just as in unregularized stochastic gradient descent, we can estimate $\partial C_0 / \partial w$ by averaging over a mini-batch of m training examples. Thus the regularized learning rule for stochastic gradient descent becomes (c.f. Equation (20))

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

Winnovative PDF Tools Demo

where the sum is over training examples x

in the mini-batch, and C_x

is the (unregularized) cost for each training example. This is

exactly the same as the usual rule for stochastic gradient descent,

except for the $1 - \frac{\eta\lambda}{n}$

weight decay factor. Finally, and for completeness, let me state the

regularized learning rule for the biases. This is, of course, exactly

the same as in the unregularized case (c.f. Equation (21)),

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

where the sum is over training examples x

in the mini-batch.

Let's see how regularization changes the performance of our

neural network. We'll use a network with 30

hidden neurons, a mini-batch size of 10

, a learning rate of 0.5

, and the cross-entropy cost function. However, this time we'll use

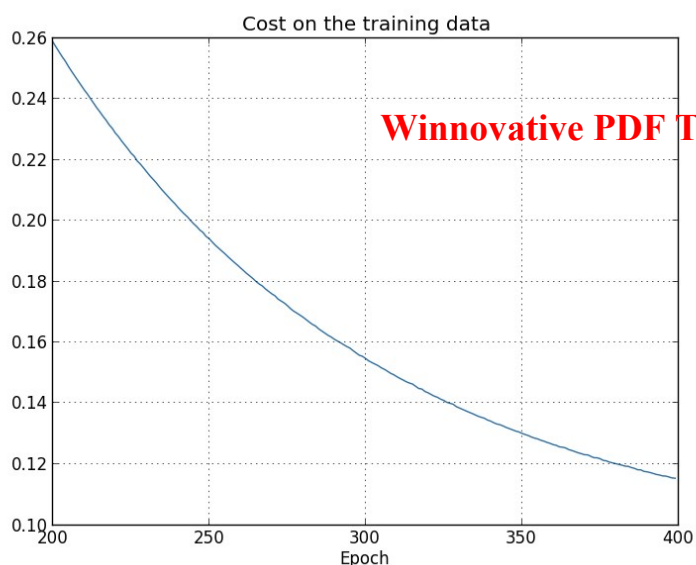
a regularization parameter of $\lambda = 0.1$

. Note that in the code, we use the variable name `lmbda`, because `lambda` is a reserved word in Python, with an unrelated meaning. I've also used the `test_data` again, not the `validation_data`. Strictly speaking, we should use the `validation_data`, for all the reasons we discussed earlier. But I decided to use the `test_data` because it makes the results more directly comparable with our earlier, unregularized results. You can easily change the code to use the `validation_data` instead, and you'll find that it gives similar results.

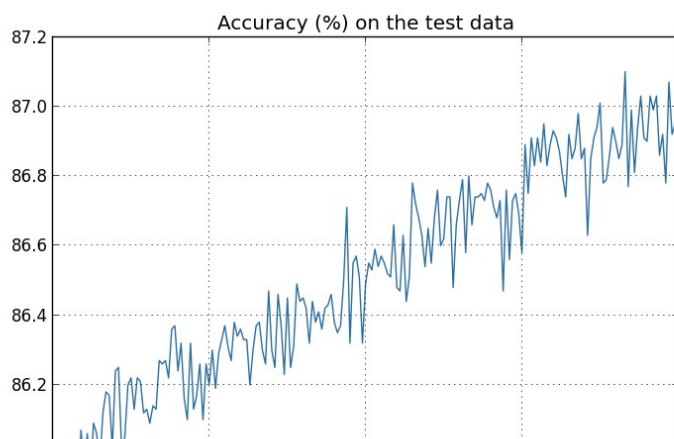
```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5,
... evaluation_data=test_data,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

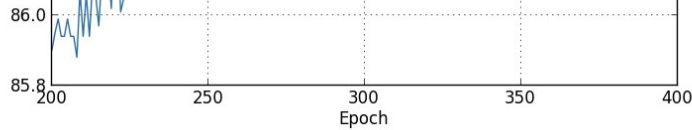
The cost on the training data decreases over the whole time, much as it did in the earlier, unregularized case*:

*This and the next two graphs were produced with the program [overfitting.py](#).



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





Clearly, the use of regularization has suppressed overfitting.

What's more, the accuracy is considerably higher, with a peak classification accuracy of 87.1

percent, compared to the peak of 82.27

percent obtained in the unregularized case. Indeed, we could

almost certainly get considerably better results by continuing to

train past 400 epochs. It seems that, empirically, regularization is

causing our network to generalize better, and considerably

reducing the effects of overfitting.

What happens if we move out of the artificial environment of just

having 1,000 training images, and return to the full 50,000 image

training set? Of course, we've seen already that overfitting is much

less of a problem with the full 50,000 images. Does regularization

help any further? Let's keep the hyper-parameters the same as

before - 30

epochs, learning rate 0.5

, mini-batch size of 10

. However, we need to modify the regularization parameter. The

reason is because the size n

of the training set has changed from $n = 1,000$

to $n = 50,000$

, and this changes the weight decay factor $1 - \frac{\eta\lambda}{n}$

. If we continued to use $\lambda = 0.1$

that would mean much less weight decay, and thus much less of a

regularization effect. We compensate by changing to $\lambda = 5.0$

.

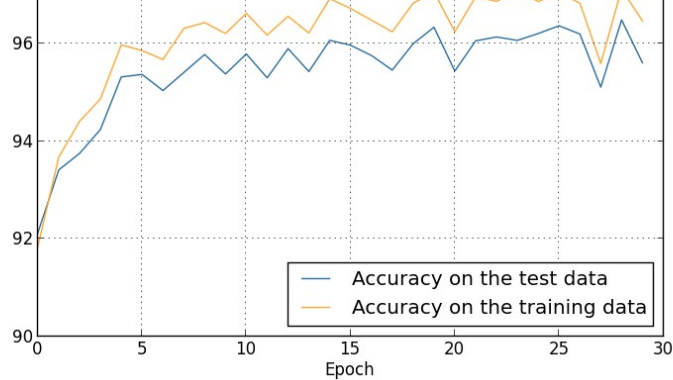
Okay, let's train our network, stopping first to re-initialize the

weights:

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

We obtain the results:





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

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

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

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

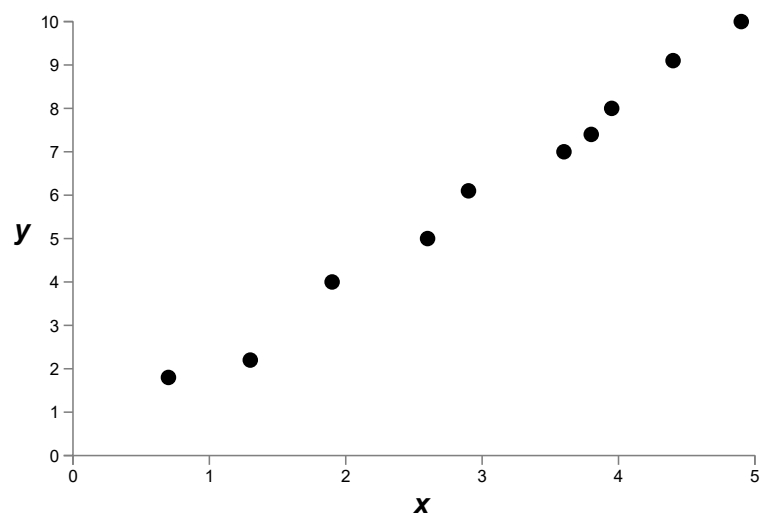
I've described regularization as a way to reduce overfitting and to increase classification accuracies. In fact, that's not the only benefit. Empirically, when doing multiple runs of our MNIST networks, but with different (random) weight initializations, I've found that the unregularized runs will occasionally get "stuck",

apparently caught in local minima of the cost function. The result is that different runs sometimes provide quite different results. By contrast, the regularized runs have provided much more easily replicable results.

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

Why does regularization help reduce overfitting?

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



Implicitly, we're studying some real-world phenomenon here, with

x

and y

representing real-world data. Our goal is to build a model which lets us predict y as a function of x

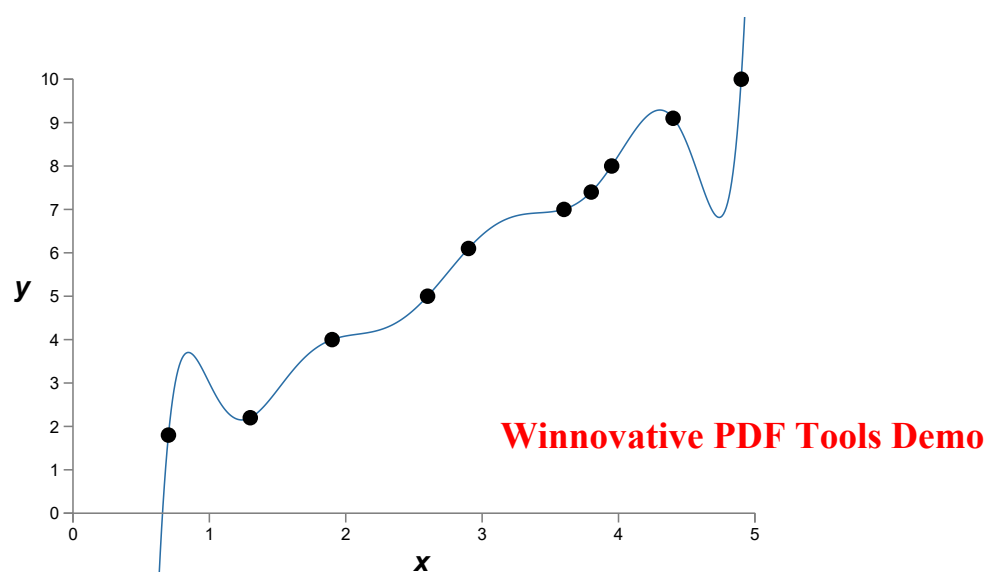
. We could try using neural networks to build such a model, but I'm going to do something even simpler: I'll try to model y as a polynomial in x

. I'm doing this instead of using neural nets because using polynomials will make things particularly transparent. Once we've understood the polynomial case, we'll translate to neural networks.

Now, there are ten points in the graph above, which means we can find a unique 9

th-order polynomial $y = a_0x^9 + a_1x^8 + \dots + a_9$

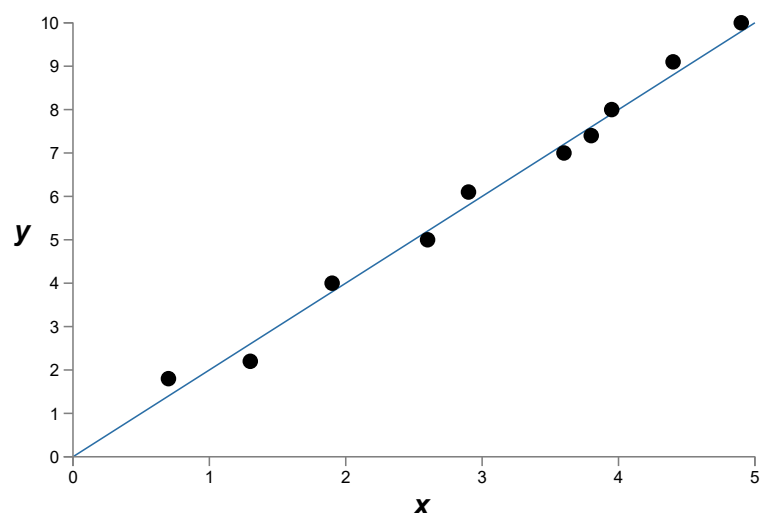
which fits the data exactly. Here's the graph of that polynomial*:



*I won't show the coefficients explicitly, although they are easy to find using a routine such as Numpy's `polyfit`. You can view the exact form of the polynomial in the [source code for the graph](#) if you're curious. It's the function `p(x)` defined starting on line 14 of the program which produces the graph.

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

:



Which of these is the better model? Which is more likely to be true? And which model is more likely to generalize well to other

examples of the same underlying real-world phenomenon?

These are difficult questions. In fact, we can't determine with certainty the answer to any of the above questions, without much more information about the underlying real-world phenomenon.

But let's consider two possibilities: (1) the 9th order polynomial is, in fact, the model which truly describes the real-world phenomenon, and the model will therefore generalize perfectly; (2) the correct model is $y = 2x$, but there's a little additional noise due to, say, measurement error, and that's why the model isn't an exact fit.

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

Innovative PDF Tools Demo

One point of view is to say that in science we should go with the simpler explanation, unless compelled not to. When we find a simple model that seems to explain many data points we are tempted to shout "Eureka!" After all, it seems unlikely that a simple explanation should occur merely by coincidence. Rather, we suspect that the model must be expressing some underlying truth about the phenomenon. In the case at hand, the model

$$y = 2x + \text{noise}$$

seems much simpler than $y = a_0x^9 + a_1x^8 + \dots$.

It would be surprising if that simplicity had occurred by chance,

and so we suspect that $y = 2x + \text{noise}$

expresses some underlying truth. In this point of view, the 9th order model is really just learning the effects of local noise. And so while the 9th order model works perfectly for these particular data points, the model will fail to generalize to other data points, and the noisy linear model will have greater predictive power.

Let's see what this point of view means for neural networks.

Suppose our network mostly has small weights, as will tend to happen in a regularized network. The smallness of the weights

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

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

Let me describe two examples where more complex explanations have turned out to be correct. In the 1940s the physicist Marcel Schein announced the discovery of a new particle of nature. The company he worked for, General Electric, was ecstatic, and publicized the discovery widely. But the physicist Hans Bethe was skeptical. Bethe visited Schein, and looked at the plates showing the tracks of Schein's new particle. Schein showed Bethe plate after plate, but on each plate Bethe identified some problem that suggested the data should be discarded. Finally, Schein showed Bethe a plate that looked good. Bethe said it might just be a statistical fluke. Schein: "Yes, but the chance that this would be statistics, even according to your own formula, is one in five." Bethe: "But we have already looked at five plates." Finally, Schein said: "But on my plates, each one of the good plates, each one of the good pictures, you explain by a different theory, whereas I have one hypothesis that explains all the plates, that they are [the new particle]." Bethe replied: "The sole difference between your and my

explanations is that yours is wrong and all of mine are right. Your single explanation is wrong, and all of my multiple explanations are right." Subsequent work confirmed that Nature agreed with Bethe, and Schein's particle is no more*.

*The story is related by the physicist Richard Feynman in an [interview](#) with the historian Charles Weiner.

As a second example, in 1859 the astronomer Urbain Le Verrier observed that the orbit of the planet Mercury doesn't have quite the shape that Newton's theory of gravitation says it should have. It was a tiny, tiny deviation from Newton's theory, and several of the explanations proffered at the time boiled down to saying that Newton's theory was more or less right, but needed a tiny alteration. In 1916, Einstein showed that the deviation could be explained very well using his general theory of relativity, a theory radically different to Newtonian gravitation, and based on much more complex mathematics. Despite that additional complexity, today it's accepted that Einstein's explanation is correct, and Newtonian gravity, even in its modified forms, is wrong. This is in part because we now know that Einstein's theory explains many other phenomena which Newton's theory has difficulty with. Furthermore, and even more impressively, Einstein's theory accurately predicts several phenomena which aren't predicted by Newtonian gravity at all. But these impressive qualities weren't entirely obvious in the early days. If one had judged merely on the grounds of simplicity, then some modified form of Newton's theory would arguably have been more attractive.

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

With that said, and keeping the need for caution in mind, it's an empirical fact that regularized neural networks usually generalize better than unregularized networks. And so through the remainder of the book we will make frequent use of regularization. I've included the stories above merely to help convey why no-one has yet developed an entirely convincing theoretical explanation for why regularization helps networks generalize. Indeed, researchers continue to write papers where they try different approaches to regularization, compare them to see which works better, and attempt to understand why different approaches work better or worse. And so you can view regularization as something of a

Winnovative PDF Tools Demo

kludge. While it often helps, we don't have an entirely satisfactory systematic understanding of what's going on, merely incomplete heuristics and rules of thumb.

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

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

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

Let me conclude this section by returning to a detail which I left unexplained earlier: the fact that L2 regularization *doesn't* constrain the biases. Of course, it would be easy to modify the regularization procedure to regularize the biases. Empirically, doing this often doesn't change the results very much, so to some

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

Wininative PDF Tools Demo

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

extent it's merely a convention whether to regularize the biases or not. However, it's worth noting that having a large bias doesn't make a neuron sensitive to its inputs in the same way as having large weights. And so we don't need to worry about large biases enabling our network to learn the noise in our training data. At the same time, allowing large biases gives our networks more flexibility in behaviour - in particular, large biases make it easier for neurons to saturate, which is sometimes desirable. For these reasons we don't usually include bias terms when regularizing.

Other techniques for regularization

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

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

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

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

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

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

where $\text{sgn}(w)$ is the sign of w , that is, $+1$

if w

is positive, and -1

if w

is negative. Using this expression, we can easily modify

backpropagation to do stochastic gradient descent using L1

regularization. The resulting update rule for an L1 regularized

network is

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

where, as per usual, we can estimate $\partial C_0 / \partial w$

using a mini-batch average, if we wish. Compare that to the update

rule for L2 regularization (c.f. Equation (93)),

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

In both expressions the effect of regularization is to shrink the

weights. This accords with our intuition that both kinds of

regularization penalize large weights. But the way the weights

shrink is different. In L1 regularization, the weights shrink by a

constant amount toward 0

. In L2 regularization, the weights shrink by an amount which is proportional to w

. And so when a particular weight has a large magnitude, $|w|$

, L1 regularization shrinks the weight much less than L2

regularization does. By contrast, when $|w|$

is small, L1 regularization shrinks the weight much more than L2

regularization. The net result is that L1 regularization tends to

concentrate the weight of the network in a relatively small number

of high-importance connections, while the other weights are driven

toward zero.

I've glossed over an issue in the above discussion, which is that the

partial derivative $\partial C / \partial w$

isn't defined when $w = 0$

. The reason is that the function $|w|$

has a sharp "corner" at $w = 0$

, and so isn't differentiable at that point. That's okay, though. What

we'll do is just apply the usual (unregularized) rule for stochastic

gradient descent when $w = 0$

. That should be okay - intuitively, the effect of regularization is to

shrink weights, and obviously it can't shrink a weight which is

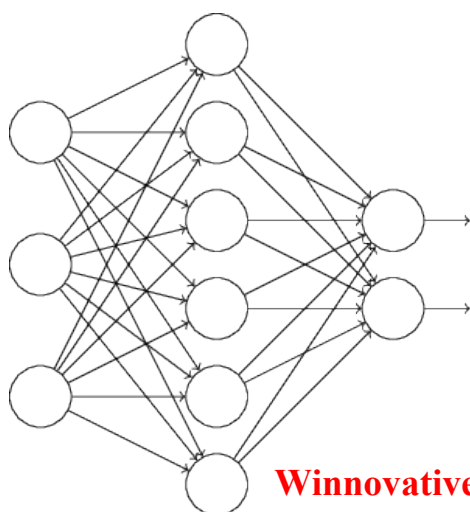
already 0

. To put it more precisely, we'll use Equations (96) and (97) with the convention that $\text{sgn}(0) = 0$

. That gives a nice, compact rule for doing stochastic gradient descent with L1 regularization.

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

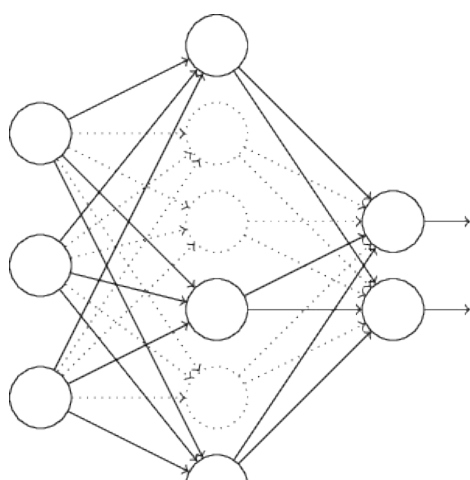
Suppose we're trying to train a network:



Winnovative PDF Tools Demo

In particular, suppose we have a training input x and corresponding desired output y

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



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

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

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

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

A related heuristic explanation for dropout is given in one of the

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

Of course, the true measure of dropout is that it has been very successful in improving the performance of neural networks. The original paper* introducing the technique applied it to many different tasks. For us, it's of particular interest that they applied dropout to MNIST digit classification, using a vanilla feedforward neural network along lines similar to those we've been considering. The paper noted that the best result anyone had achieved up to

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

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

Winnovative PDF Tools Demo