3.2: 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*

*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.

"I remember my friend Johnny von Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk."

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

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

Let's sharpen this problem up by constructing a situation where our network does a bad job generalizing to new situations. We'll use our 30 hidden neuron network, with its 23,860 parameters. But we won't train the network using all
50,000 MNIST training images. Instead, we'll use just the first 1,000 training images. Using that restricted set will make the problem with generalization much more evident. We'll train in a similar way to before, using the cross-entropy cost function, with a learning rate of $\eta=0.5$ and a mini-batch size of 10. However, we'll train for 400 epochs, a somewhat larger number than before, because we're not using as many training examples. Let's use network2 to look at the way the cost function changes:

```python
>>> 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.*

![Cost on the training data](https://eng.libretexts.org/Bookshelves/Computer_Science/Book%3A_Neural_Networks_and_Deep_Learning_(Nielsen)/03%3…)

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:

![Graph: 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:

```python
>>> 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*

*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.*

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 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:

\[
C = -\frac{1}{n} \sum_x [y_j \ln a^L_j + (1 - y_j) \ln (1 - a^L_j)] + \frac{\lambda}{2n} \sum_w w^2. \tag{85}
\]

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 \(\frac{\lambda}{2n}\), 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 \(\lambda\) 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:

\[
C = \frac{1}{2n} \sum_x \| y - a^L \|^2 + \frac{\lambda}{2n} \sum_w w^2. \quad \tag{86}
\]

In both cases we can write the regularized cost function as

\[
C = C_0 + \frac{\lambda}{2n} \sum_w w^2, \quad \tag{87}
\]

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 \( \lambda \): when \( \lambda \) is small we prefer to minimize the original cost function, but when \( \lambda \) 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 \( \frac{\partial C}{\partial w} \) and \( \frac{\partial C}{\partial b} \) for all the weights and biases in the network. Taking the partial derivatives of Equation \( \ref{87} \) gives

\[
\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \quad \tag{88}
\]

\[
\frac{\partial C_0}{\partial b} = \frac{\partial C_0}{\partial b}. \quad \tag{89}
\]

The \( \frac{\partial C_0}{\partial w} \) and \( \frac{\partial C_0}{\partial b} \) 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 \( \lambda n w \) 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:

\[
\begin{align}
\frac{\partial C}{\partial b} &= \frac{\partial C_0}{\partial b}. \quad \tag{90}
\end{align}
\]

The learning rule for the weights becomes:

\[
\begin{align}
\textbf{align} \ w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w, \quad \textbf{align} \ \tag{91}
\end{align}
\]

This is exactly the same as the usual gradient descent learning rule, except we first rescale the weight \( \eta w \) by a factor \( \left(1 - \frac{\eta \lambda}{n}\right) \). 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 (ref{20})))

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

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 (ref{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 $\text{lambda}$ 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.

```python
>>> 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:
We obtain the results:

![Graph showing accuracy over epochs](image)

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.

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 \( \langle x \rangle \) and \( \langle y \rangle \) representing real-world data. Our goal is to build a model which lets us predict \( \langle y \rangle \) as a function of \( \langle x \rangle \). 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 \( \langle y \rangle \) as a polynomial in \( \langle x \rangle \). 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 9th-order polynomial \( \langle y=a_0x^9+a_1x^8+\ldots+a_9 \rangle \) 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 true that on the data provided there's only a small difference between the two models. But suppose we want to predict the value of $y$ corresponding to some large value of $x$, much larger than any shown on the graph above. If we try to do that there will be a dramatic difference between the predictions of the two models, as the 9th order polynomial model comes to be dominated by the $x^9$ term, while the linear model remains, well, linear.

One point of view is to say that in science we should go with the simpler explanation, unless compelled not to. When we

find a simple model that seems to explain many data points we are tempted to shout "Eureka!" After all, it seems unlikely that a simple explanation should occur merely by coincidence. Rather, we suspect that the model must be expressing some underlying truth about the phenomenon. In the case at hand, the model \( y = 2x + \text{noise} \) seems much simpler than \( y = a_0x^9 + a_1x^8 + \ldots \). It would be surprising if that simplicity had occurred by chance, and so we suspect that \( y = 2x + \text{noise} \) expresses some underlying truth. In this point of view, the 9th order model is really just learning the effects of local noise. And so while the 9th order model works perfectly for these particular data points, the model will fail to generalize to other data points, and the noisy linear model will have greater predictive power.

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

With that said, this idea of preferring simpler explanation should make you nervous. People sometimes refer to this idea as "Occam's Razor", and will zealously apply it as though it has the status of some general scientific principle. But, of course, it's not a general scientific principle. There is no \textit{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
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*

*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).

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

In fact, our networks already generalize better than one might a priori expect. A network with 100 hidden neurons has
nearly 80,000 parameters. We have only 50,000 images in our training data. It's like trying to fit an 80,000th degree
polynomial to 50,000 data points. By all rights, our network should overfit terribly. And yet, as we saw earlier, such a
network actually does a pretty good job generalizing. Why is that the case? It's not well understood. It has been
conjectured*
In Gradient-Based Learning Applied to Document Recognition, by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998). 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 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 $C_0$ 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

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

where, as per usual, we can estimate $\frac{\partial C_0}{\partial w}$ using a mini-batch average, if we wish. Compare that to the update rule for L2 regularization (c.f. Equation $\ref{93}$).
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 \(\frac{\partial C}{\partial w}\) 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 \(\ref{96}\) and \(\ref{97}\) with the convention that \(|sgn(0)=0\|\). 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:
following lines. Note that the dropout neurons, i.e., the neurons which have been temporarily deleted, are still ghosted in:

![Diagram showing a neural network with dropout neurons]

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 that point using such an architecture was \(98.4\) percent classification accuracy on the test set. They improved that to \(98.7\) percent accuracy using a combination of dropout and a modified form of L2 regularization. Similarly impressive results have been obtained for many other tasks, including problems in image and speech recognition, and natural language processing. Dropout has been especially useful in training large, deep networks, where the problem of overfitting is often acute.

*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.

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

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

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

Obtaining more training data is a great idea. Unfortunately, it can be expensive, and so is not always possible in practice. However, there’s another idea which can work nearly as well, and that’s to artificially expand the training data. Suppose, for example, that we take an MNIST training image of a five,
and rotate it by a small amount, let's say 15 degrees:

![Image of rotated digit]

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

This idea is very powerful and has been widely used. Let's look at some of the results from a paper*

*Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis, by Patrice Simard, Dave Steinkraus, and John Platt (2003). which applied several variations of the idea to MNIST. One of the neural network architectures they considered was along similar lines to what we've been using, a feedforward network with 800 hidden neurons and using the cross-entropy cost function. Running the network with the standard MNIST training data they achieved a classification accuracy of $98.4\%$ percent on their test set. But then they expanded the training data, using not just rotations, as I described above, but also translating and skewing the images. By training on the expanded data set they increased their network's accuracy to $98.9\%$ percent. They also experimented with what they called "elastic distortions", a special type of image distortion intended to emulate the random oscillations found in hand muscles. By using the elastic distortions to expand the data they achieved an even higher accuracy, $99.3\%$ percent. Effectively, they were broadening the experience of their network by exposing it to the sort of variations that are found in real handwriting.

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

**Exercise**

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

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

![Graph showing accuracy (%) on the validation data vs training set size](https://eng.libretexts.org/Bookshelves/Computer_Science/Book%3A_Neural_Networks_and_Deep_Learning_(Nielsen)/03%3D...)

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

*This graph was produced with the program more_data.py (as were the last few graphs):
Probably the first thing that strikes you about this graph is that our neural network outperforms the SVM for every training set size. That's nice, although you shouldn't read too much into it, since I just used the out-of-the-box settings from scikit-learn's SVM, while we've done a fair bit of work improving our neural network. A more subtle but more interesting fact about the graph is that if we train our SVM using 50,000 images then it actually has better performance (94.48 percent accuracy) than our neural network does when trained using 5,000 images (93.24 percent accuracy). In other words, more training data can sometimes compensate for differences in the machine learning algorithm used.

Something even more interesting can occur. Suppose we're trying to solve a problem using two machine learning algorithms, algorithm A and algorithm B. It sometimes happens that algorithm A will outperform algorithm B with one set of training data, while algorithm B will outperform algorithm A with a different set of training data. We don't see that above - it would require the two graphs to cross - but it does happen*

*Striking examples may be found in Scaling to very very large corpora for natural language disambiguation, by Michele Banko and Eric Brill (2001). The correct response to the question "Is algorithm A better than algorithm B?" is really: "What training data set are you using?"

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

Problem

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

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