CHAPTER 4

The Multi-layer Perceptron

In the last chapter we saw that while linear models are easy to understand and use, they come with the inherent cost that is implied by the word 'linear'; that is, they can only identify straight lines, planes, or hyperplanes. And this is not usually enough, because the majority of interesting problems are not linearly separable. In Section 3.4 we saw that problems can be made linearly separable if we can work out how to transform the features suitably. We will come back to this idea in Chapter 8, but in this chapter we will instead consider making more complicated networks.

We have pretty much decided that the learning in the neural network happens in the weights. So, to perform more computation it seems sensible to add more weights. There are two things that we can do: add some backwards connections, so that the output neurons connect to the inputs again, or add more neurons. The first approach leads into recurrent networks. These have been studied, but are not that commonly used. We will instead consider the second approach. We can add neurons between the input nodes and the outputs, and this will make more complex neural networks, such as the one shown in Figure 4.1.

We will think about why adding extra layers of nodes makes a neural network more powerful in Section 4.3.2, but for now, to persuade ourselves that it is true, we can check that a prepared network can solve the two-dimensional XOR problem, something that we have seen is not possible for a linear model like the Perceptron. A suitable network is shown in Figure 4.2. To check that it gives the correct answers, all that is required is to put in each input and work through the network, treating it as two different Perceptrons, first computing the activations of the neurons in the middle layer (labelled as C and D in Figure 4.2) and then using those activations as the inputs to the single neuron at the output. As an example, I'll work out what happens when you put in $(1, 0)$ as an input; the job of checking the rest is up to you.

Input $(1, 0)$ corresponds to node A being 1 and B being 0. The input to neuron C is therefore $-1 \times 0.5 + 1 \times 1 + 0 \times 1 = -0.5 + 1 = 0.5$. This is above the threshold of 0, and so neuron C fires, giving output 1. For neuron D the input is $-1 \times 1 + 1 \times 1 + 0 \times 1 = -1 + 1 = 0$, and so it does not fire, giving output 0. Therefore the input to neuron E is $-1 \times 0.5 + 1 \times 1 + 0 \times -1 = 0.5$, so neuron E fires. Checking the result of the inputs should persuade you that neuron E fires when inputs A and B are different to each other, but does not fire when they are the same, which is exactly the XOR function (it doesn't matter that the fire and not fire have been reversed).

So far, so good. Since this network can solve a problem that the Perceptron cannot, it seems worth looking into further. However, now we've got a much more interesting problem to solve, namely how can we train this network so that the weights are adapted to generate the correct (target) answers? If we try the method that we used for the Perceptron we need

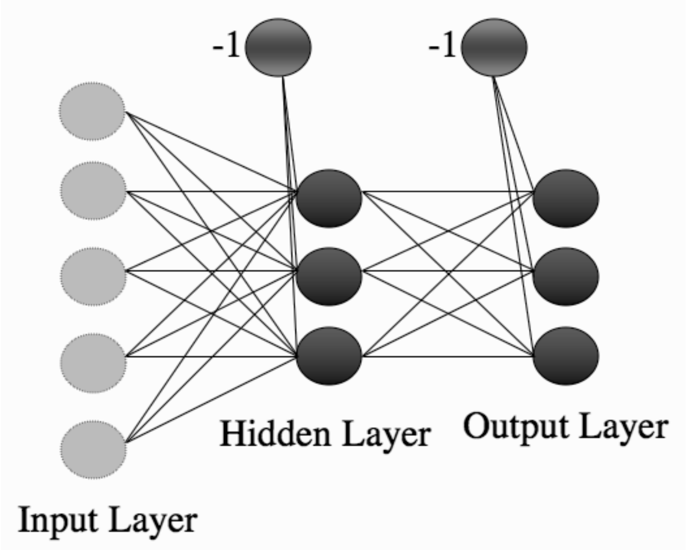

FIGURE 4.1 The Multi-layer Perceptron network, consisting of multiple layers of connected neurons.

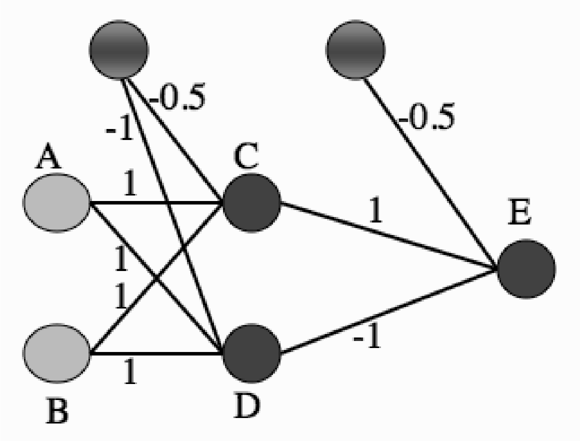

FIGURE 4.2 A Multi-layer Perceptron network showing a set of weights that solve the XOR problem.

to compute the error at the output. That's fine, since we know the targets there, so we can compute the difference between the targets and the outputs. But now we don't know which weights were wrong: those in the first layer, or the second? Worse, we don't know what the correct activations are for the neurons in the middle of the network. This fact gives the neurons in the middle of the network their name; they are called the hidden layer (or layers), because it isn't possible to examine and correct their values directly.

It took a long time for people who studied neural networks to work out how to solve this problem. In fact, it wasn't until 1986 that Rumelhart, Hinton, and McClelland managed it. However, a solution to the problem was already known by statisticians and engineers—they just didn't know that it was a problem in neural networks! In this chapter we are going to look at the neural network solution proposed by Rumelhart, Hinton, and McClelland, the Multi-layer Perceptron (MLP), which is still one of the most commonly used machine learning methods around. The MLP is one of the most common neural networks in use. It is often treated as a 'black box', in that people use it without understanding how it works, which often results in fairly poor results. Getting to the stage where we understand how it works and what we can do with it is going to take us into lots of different areas of statistics, mathematics, and computer science, so we'd better get started.

4.1 GOING FORWARDS

Just as it did for the Perceptron, training the MLP consists of two parts: working out what the outputs are for the given inputs and the current weights, and then updating the weights according to the error, which is a function of the difference between the outputs and the targets. These are generally known as going forwards and backwards through the network. We've already seen how to go forwards for the MLP when we saw the XOR example above, which was effectively the recall phase of the algorithm. It is pretty much just the same as the Perceptron, except that we have to do it twice, once for each set of neurons, and we need to do it layer by layer, because otherwise the input values to the second layer don't exist. In fact, having made an MLP with two layers of nodes, there is no reason why we can't make one with 3, or 4, or 20 layers of nodes (we'll discuss whether or not you might want to in Section 4.3.2). This won't even change our recall (forward) algorithm much, since we just work forwards through the network computing the activations of one layer of neurons and using those as the inputs for the next layer.

So looking at Figure 4.1, we start at the left by filling in the values for the inputs. We then use these inputs and the first level of weights to calculate the activations of the hidden layer, and then we use those activations and the next set of weights to calculate the activations of the output layer. Now that we've got the outputs of the network, we can compare them to the targets and compute the error.

4.1.1 Biases

We need to include a bias input to each neuron. We do this in the same way as we did for the Perceptron in Section 3.3.2, by having an extra input that is permanently set to -1, and adjusting the weights to each neuron as part of the training. Thus, each neuron in the network (whether it is a hidden layer or the output) has 1 extra input, with fixed value.

4.2 GOING BACKWARDS: BACK-PROPAGATION OF ERROR

It is in the backwards part of the algorithm that things get tricky. Computing the errors at the output is no more difficult than it was for the Perceptron, but working out what to do with those errors is more difficult. The method that we are going to look at is called back-propagation of error, which makes it clear that the errors are sent backwards through the network. It is a form of gradient descent (which is described briefly below, and also given its own section in Chapter 9; in that chapter, in Section 9.3.2, we will see how to use the general gradient descent algorithms for the MLP).

The best way to describe back-propagation properly is mathematically, but this can be intimidating and difficult to get a handle on at first. I've therefore tried to compromise by using words and pictures in the main text, but putting all of the mathematical details into Section 4.6. While you should look at that section and try to understand it, it can be skipped if you really don't have the background. Although it looks complicated, there are actually just three things that you need to know, all of which are from differential calculus: the derivative of $\frac{1}{2}x^2$, the fact that if you differentiate a function of x with respect to some other variable t, then the answer is 0, and the chain rule, which tells you how to differentiate composite functions.

When we talked about the Perceptron, we changed the weights so that the neurons fired when the targets said they should, and didn't fire when the targets said they shouldn't. What we did was to choose an error function for each neuron k: $E_k = y_k - t_k$, and tried to make it as small as possible. Since there was only one set of weights in the network, this was sufficient to train the network.

We still want to do the same thing—minimise the error, so that neurons fire only when they should—but, with the addition of extra layers of weights, this is harder to arrange. The problem is that when we try to adapt the weights of the Multi-layer Perceptron, we have to work out which weights caused the error. This could be the weights connecting the inputs to the hidden layer, or the weights connecting the hidden layer to the output layer. (For more complex networks, there could be extra weights between nodes in hidden layers. This isn't a problem—the same method works—but it is more confusing to talk about, so I'm only going to worry about one hidden layer here.)

The error function that we used for the Perceptron was $\sum_{k=1}^{N} E_k = \sum_{k=1}^{N} y_k - t_k$, where N is the number of output nodes. However, suppose that we make two errors. In the first, the target is bigger than the output, while in the second the output is bigger than the target. If these two errors are the same size, then if we add them up we could get 0, which means that the error value suggests that no error was made. To get around this we need to make all errors have the same sign. We can do this in a few different ways, but the one that will turn out to be best is the sum-of-squares error function, which calculates the difference between y and t for each node, squares them, and adds them all together:

$$E(\mathbf{t}, \mathbf{y}) = \frac{1}{2} \sum_{k=1}^{N} (y_k - t_k)^2. \tag{4.1}$$

You might have noticed the $\frac{1}{2}$ at the front of that equation. It doesn't matter that much, but it makes it easier when we differentiate the function, and that is the name of the game here: if we differentiate a function, then it tells us the gradient of that function, which is the direction along which it increases and decreases the most. So if we differentiate an error function, we get the gradient of the error. Since the purpose of learning is to minimise the error, following the error function downhill (in other words, in the direction of the negative gradient) will give us what we want. Imagine a ball rolling around on a surface that looks

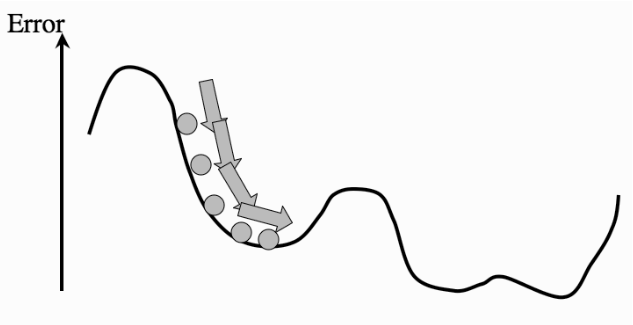

FIGURE 4.3 The weights of the network are trained so that the error goes downhill until it reaches a local minimum, just like a ball rolling under gravity.

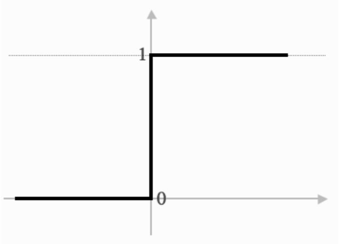

FIGURE 4.4 The threshold function that we used for the Perceptron. Note the discontinuity where the value changes from 0 to 1.

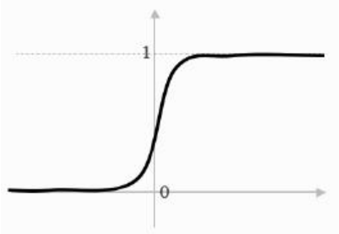

FIGURE 4.5 The sigmoid function, which looks qualitatively fairly similar, but varies smoothly and differentiably.

like the line in Figure 4.3. Gravity will make the ball roll downhill (follow the downhill gradient) until it ends up in the bottom of one of the hollows. These are places where the error is small, so that is exactly what we want. This is why the algorithm is called gradient descent. So what should we differentiate with respect to? There are only three things in the network that change: the inputs, the activation function that decides whether or not the node fires, and the weights. The first and second are out of our control when the algorithm is running, so only the weights matter, and therefore they are what we differentiate with respect to.

Having mentioned the activation function, this is a good time to point out a little problem with the threshold function that we have been using for our neurons so far, which is that it is discontinuous (see Figure 4.4; it has a sudden jump in the middle) and so differentiating it at that point isn't possible. The problem is that we need that jump between firing and not firing to make it act like a neuron. We can solve the problem if we can find an activation function that looks like a threshold function, but is differentiable so that we can compute the gradient. If you squint at a graph of the threshold function (for example, Figure 4.4) then it looks kind of S-shaped. There is a mathematical form of S-shaped functions, called sigmoid functions (see Figure 4.5). They have another nice property, which is that their derivative also has a nice form, as is shown in Section 4.6.3 for those who know some mathematics. The most commonly used form of this function (where β is some positive parameter) is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}. \tag{4.2}$$

In some texts you will see the activation function given a different form, as:

$$a = g(h) = \tanh(h) = \frac{\exp(h) - \exp(-h)}{\exp(h) + \exp^(-h)}, \tag{4.3}$$

which is the hyperbolic tangent function. This is a different but similar function; it is still a sigmoid function, but it saturates (reaches its constant values) at ±1 instead of 0 and 1, which is sometimes useful. It also has a relatively simple derivative: $\frac{d}{dx} \tanh x = (1 - \tanh^2(x))$. We can convert between the two easily, because if the saturation points are (± 1), then we can convert to $(0, 1)$ by using $0.5 \times (x + 1)$.

So now we've got a new form of error computation and a new activation function that decides whether or not a neuron should fire. We can differentiate it, so that when we change the weights, we do it in the direction that is downhill for the error, which means that we know we are improving the error function of the network. As far as an algorithm goes, we've fed our inputs forward through the network and worked out which nodes are firing. Now, at the output, we've computed the errors as the sum-squared difference between the outputs and the targets (Equation (4.1) above). What we want to do next is to compute the gradient of these errors and use them to decide how much to update each weight in the network. We will do that first for the nodes connected to the output layer, and after we have updated those, we will work *backwards* through the network until we get back to the inputs again. There are just two problems:

- for the output neurons, we don't know the inputs.
- for the hidden neurons, we don't know the targets; for extra hidden layers, we know neither the inputs nor the targets, but even this won't matter for the algorithm we derive.

So we can compute the error at the output, but since we don't know what the inputs were that caused it, we can't update those second layer weights the way we did for the Perceptron. If we use the chain rule of differentiation that you all (possibly) remember from high school then we can get around this problem. Here, the chain rule tells us that if we want to know how the error changes as we vary the weights, we can think about how the error changes as we vary the inputs to the weights, and multiply this by how those input values change as we vary the weights. This is useful because it lets us calculate all of the derivatives that we want to: we can write the activations of the output nodes in terms of the activations of the hidden nodes and the output weights, and then we can send the error calculations back through the network to the hidden layer to decide what the target outputs were for those neurons. Note that we can do exactly the same computations if the network has extra hidden layers between the inputs and the outputs. It gets harder to keep track of which functions we should be differentiating, but there are no new tricks needed.

All of the relevant equations are derived in Section 4.6, and you should read that section carefully, since it is quite difficult to describe exactly what is going on here in words. The important thing to understand is that we compute the gradients of the errors with respect to the weights, so that we change the weights so that we go downhill, which makes the errors get smaller. We do this by differentiating the error function with respect to the weights, but we can't do this directly, so we have to apply the chain rule and differentiate with respect to things that we know. This leads to two different update functions, one for each of the

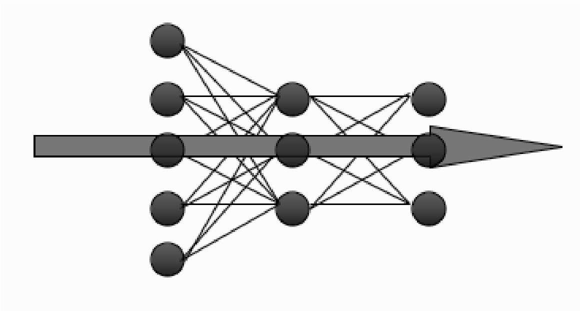

FIGURE 4.6 The forward direction in a Multi-layer Perceptron.

sets of weights, and we just apply these backwards through the network, starting at the outputs and ending up back at the inputs.

4.2.1 The Multi-layer Perceptron Algorithm

We'll get into the details of the basic algorithm here, and then, in the next section, have a look at some practical issues, such as how much training data is needed, how much training time is needed, and how to choose the correct size of network. We will assume that there are L input nodes, plus the bias, M hidden nodes, also plus a bias, and N output nodes, so that there are $(L+1) \times M$ weights between the input and the hidden layer and $(M+1) \times N$ between the hidden layer and the output. The sums that we write will start from 0 if they include the bias nodes and 1 otherwise, and run up to L, M, or N, so that $x_0 = -1$ is the bias input, and $a_0 = -1$ is the bias hidden node. The algorithm that is described could have any number of hidden layers, in which case there might be several values for M, and extra sets of weights between the hidden layers. We will also use i, j, k to index the nodes in each layer in the sums, and the corresponding Greek letters (ι, ζ, κ) for fixed indices.

Here is a quick summary of how the algorithm works, and then the full MLP training algorithm using back-propagation of error is described.

1. an input vector is put into the input nodes
2. the inputs are fed *forward* through the network (Figure 4.6)
   - the inputs and the first-layer weights (here labelled as v) are used to decide whether the hidden nodes fire or not. The activation function $g(\cdot)$ is the sigmoid function given in Equation (4.2) above
   - the outputs of these neurons and the second-layer weights (labelled as w) are used to decide if the output neurons fire or not
3. the *error* is computed as the sum-of-squares difference between the network outputs and the targets
4. this error is fed *backwards* through the network in order to
   - first update the second-layer weights
   - and then afterwards, the first-layer weights

The Multi-layer Perceptron Algorithm

```
• Initialisation
    – initialise all weights to small (positive and negative) random values
• Training
    – repeat:
        ∗ for each input vector:
          Forwards phase:
            · compute the activation of each neuron $j$ in the hidden layer(s) using:
                  $h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta}$                                  (4.4)
                  $a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)}$                 (4.5)
            · work through the network until you get to the output layer neurons,
              which have activations (although see also Section 4.2.3):
                  $h_\kappa = \sum_{j} a_j w_{j\kappa}$                                    (4.6)
                  $y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)}$              (4.7)
          Backwards phase:
            · compute the error at the output using:
                  $\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa(1-y_\kappa)$         (4.8)
            · compute the error in the hidden layer(s) using:
                  $\delta_h(\zeta) = a_\zeta(1-a_\zeta)\sum_{k=1}^{N} w_\zeta \delta_o(k)$  (4.9)
            · update the output layer weights using:
                  $w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta\delta_o(\kappa) a_\zeta^{\text{hidden}}$   (4.10)
            · update the hidden layer weights using:
                  $v_\iota \leftarrow v_\iota - \eta\delta_h(\kappa) x_\iota$               (4.11)
        ∗ (if using sequential updating) randomise the order of the input vectors so
          that you don't train in exactly the same order each iteration
    – until learning stops (see Section 4.3.3)
• Recall
    – use the Forwards phase in the training section above
```

This provides a description of the basic algorithm. As with the Perceptron, a NumPy implementation can take advantage of various matrix multiplications, which makes things easy to read and faster to compute. The implementation on the website is a `batch` version of the algorithm, so that weight updates are made after all of the input vectors have been presented (as is described in Section 4.2.4). The central weight update computations for the algorithm can be implemented as:

```
deltao = (targets-self.outputs)*self.outputs*(1.0-self.outputs)
deltah = self.hidden*(1.0-self.hidden)*(np.dot(deltao,np.transpose(self.
weights2)))

updatew1 = np.zeros((np.shape(self.weights1)))
updatew2 = np.zeros((np.shape(self.weights2)))

updatew1 = eta*(np.dot(np.transpose(inputs),deltah[:,:-1]))
updatew2 = eta*(np.dot(np.transpose(self.hidden),deltao))
self.weights1 += updatew1
self.weights2 += updatew2
```

There are a few improvements that can be made to the algorithm, and there are some important things that need to be considered, such as how many training datapoints are needed, how many hidden nodes should be used, and how much training the network needs. We will look at the improvements first, and then move on to practical considerations in Section 4.3. There are lots of details that are given in this section because it is one of the early examples in the book; later on things will be skipped over more quickly.

The first thing that we can do is check that this MLP can indeed learn the logic functions, especially the XOR. We can do that with this code (which is function `logic` on the website):

```
import numpy as np
import mlp

anddata = np.array([[0,0,0],[0,1,0],[1,0,0],[1,1,1]])
xordata = np.array([[0,0,0],[0,1,1],[1,0,1],[1,1,0]])

p = mlp.mlp(anddata[:,0:2],anddata[:,2:3],2)
p.mlptrain(anddata[:,0:2],anddata[:,2:3],0.25,1001)
p.confmat(anddata[:,0:2],anddata[:,2:3])

q = mlp.mlp(xordata[:,0:2],xordata[:,2:3],2)
q.mlptrain(xordata[:,0:2],xordata[:,2:3],0.25,5001)
q.confmat(xordata[:,0:2],xordata[:,2:3])
```

The outputs that this produces is something like:

```
Iteration:  0  Error:  0.367917569871
Iteration:  1000  Error:  0.0204860723612
Confusion matrix is:
[[ 3.  0.]
 [ 0.  1.]]
Percentage Correct:  100.0
Iteration:  0  Error:  0.515798627074
Iteration:  1000  Error:  0.499568173798
```

```
Iteration:  2000  Error:  0.498271692284
Iteration:  3000  Error:  0.480839047738
Iteration:  4000  Error:  0.382706753191
Iteration:  5000  Error:  0.0537169253359
Confusion matrix is:
[[ 2.  0.]
 [ 0.  2.]]
Percentage Correct:  100.0
```

There are a few things to notice about this. One is that it does work, producing the correct answers, but the other is that even for the AND we need significantly more iterations than we did for the Perceptron. So the benefits of a more complex network come at a cost, because it takes substantially more computational time to fit those weights to solve the problem, even for linear examples. Sometimes, even 5000 iterations are not enough for the XOR function, and more have to be added.

4.2.2 Initialising the Weights

The MLP algorithm suggests that the weights are initialised to small random numbers, both positive and negative. The question is how small is small, and does it matter? One way to get a feeling for this would be to experiment with the code, setting all of the weights to 0, and seeing how well the network learns, then setting them all to large numbers and comparing the results. However, to understand why they should be small we can look at the shape of the sigmoid. If the initial weight values are close to 1 or -1 (which is what we mean by large here) then the inputs to the sigmoid are also likely to be close to ±1 and so the output of the neuron is either 0 or 1 (the sigmoid has saturated, reached its maximum or minimum value). If the weights are very small (close to zero) then the input is still close to 0 and so the output of the neuron is just linear, so we get a linear model. Both of these things can be useful for the final network, but if we start off with values that are inbetween it can decide for itself.

Choosing the size of the initial values needs a little more thought, then. Each neuron is getting input from n different places (either input nodes if the neuron is in the hidden layer, or hidden neurons if it is in the output layer). If we view the values of these inputs as having uniform variance, then the typical input to the neuron will be $w\sqrt{n}$, where w is the initialisation value of the weights. So a common trick is to set the weights in the range $-1/\sqrt{n} < w < 1/\sqrt{n}$, where n is the number of nodes in the input layer to those weights. This makes the total input to a neuron have a maximum size of about 1. Further, if the weights are large, then the activation of a neuron is likely to be at, or close to, 0 or 1 already, which means that the gradients are small, and so the learning is very slow. There is an interplay here with the value of β in the logistic function, which means that small values of β (say $\beta = 3.0$ or less) are more effective. We use random values for the initialisation so that the learning starts off from different places for each run, and we keep them all about the same size because we want all of the weights to reach their final values at about the same time. This is known as uniform learning and it is important because otherwise the network will do better on some inputs than others.

4.2.3 Different Output Activation Functions

In the algorithm described above, we used sigmoid neurons in the hidden layer and the output layer. This is fine for classification problems, since there we can make the classes be 0 and 1. However, we might also want to perform regression problems, where the output needs to be from a continuous range, not just 0 or 1. The sigmoid neurons at the output are not very useful in that case. We can replace the output neurons with linear nodes that just sum the inputs and give that as their activation (so $g(h) = h$ in the notation of Equation (4.2)). This does not mean that we change the hidden layer neurons; they stay exactly the same, and we only modify the output nodes. They are not models of neurons anymore, since they don't have the characteristic fire/don't fire pattern. Even so, they enable us to solve regression problems, where we want a real number out, not just a 0/1 decision.

There is a third type of output neuron that is also used, which is the soft-max activation function. This is most commonly used for classification problems where the 1-of-N output encoding is used, as is described in Section 4.4.2. The soft-max function rescales the outputs by calculating the exponential of the inputs to that neuron, and dividing by the total sum of the inputs to all of the neurons, so that the activations sum to 1 and all lie between 0 and 1. As an activation function it can be written as:

$$y_\kappa = g(h_\kappa) = \frac{\exp(h_\kappa)}{\sum_{k=1}^{N} \exp(h_k)}. \tag{4.12}$$

Of course, if we change the activation function, then the derivative of the activation function will also change, and so the learning rule will be different. The changes that need to be made to the algorithm are in Equations (4.7) and (4.8), and are derived in Section 4.6.5. For the linear activation function the first is replaced by:

$$y_\kappa = g(h_\kappa) = h_\kappa, \tag{4.13}$$

while the second is replaced by:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\,. \tag{4.14}$$

For the soft-max activation, the update equation that replaces (4.8) is

$$\delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (\delta_{\kappa K} - y_K), \tag{4.15}$$

where $\delta_{\kappa K} = 1$ if $\kappa = K$ and 0 otherwise; see Section 4.6.5 for further details. However, if we modify the error function as well, to have the cross-entropy form (where ln is the natural logarithm):

$$E_{\text{ce}} = -\sum_{k=1}^{N} t_k \ln(y_k), \tag{4.16}$$

then the delta term is Equation (4.14), just as for the linear output; for more details, see Section 4.6.6. Computing these update equations requires computing the error function that is being optimised, and then differentiating it. These additions can be added into the code by allowing the user to specify the type of output activation, which has to be done twice, once in the `mlpfwd` function, and once in the `mlptrain` function. In the former, the new piece of code can be written as:

```
# Different types of output neurons
if self.outtype == 'linear':
        return outputs
elif self.outtype == 'logistic':
    return 1.0/(1.0+np.exp(-self.beta*outputs))
elif self.outtype == 'softmax':
    normalisers = np.sum(np.exp(outputs),axis=1)*np.ones((1,np.shape(outputs)
    [0]))
    return np.transpose(np.transpose(np.exp(outputs))/normalisers)
else:
    print "error"
```

4.2.4 Sequential and Batch Training

The MLP is designed to be a batch algorithm. All of the training examples are presented to the neural network, the average sum-of-squares error is then computed, and this is used to update the weights. Thus there is only one set of weight updates for each epoch (pass through all the training examples). This means that we only update the weights once for each iteration of the algorithm, which means that the weights are moved in the direction that most of the inputs want them to move, rather than being pulled around by each input individually. The batch method performs a more accurate estimate of the error gradient, and will thus converge to the local minimum more quickly.

The algorithm that was described earlier was the sequential version, where the errors are computed and the weights updated after each input. This is not guaranteed to be as efficient in learning, but it is simpler to program when using loops, and it is therefore much more common. Since it does not converge as well, it can also sometimes avoid local minima, thus potentially reaching better solutions. While the description of the algorithm is sequential, the NumPy implementation on the book website is a batch version, because the matrix manipulation methods of NumPy make that easy. It is, however, relatively simple to modify it to use sequential update (making this change to the code is suggested as an exercise at the end of the chapter). In a sequential version, the order of the weight updates can matter, which is why the pseudocode version of the algorithm include a suggestion about randomising the order of the input vectors at each iteration. This can significantly improve the speed with which the algorithm learns. NumPy has a useful function that assists with this, `np.random.shuffle()`, which takes a list of numbers and reorders them. It can be used like this:

```
np.random.shuffle(change)
inputs = inputs[change,:]
targets = targets[change,:]
```

4.2.5 Local Minima

The driving force behind the learning rule is the minimisation of the network error by gradient descent (using the derivative of the error function to make the error smaller). This

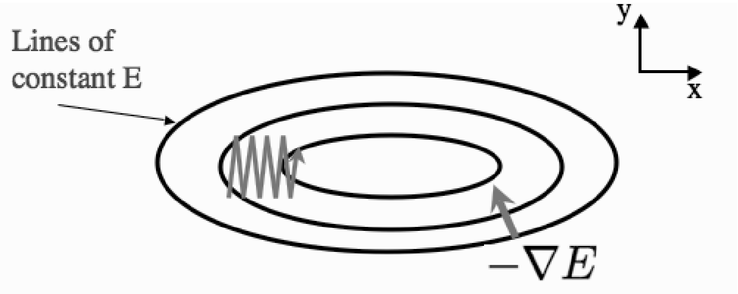

FIGURE 4.7 In 2D, downhill means at right angles to the lines of constant contour. Imagine walking down a hill with your eyes closed. If you find a direction that stays flat, then it is quite likely that perpendicular to that the ground goes uphill or downhill. However, this is not the direction that takes you directly towards the local minimum.

means that we are performing an optimisation: we are adapting the values of the weights in order to minimise the error function. As should be clear by now, the way that we are doing this is by approximating the gradient of the error and following it downhill so that we end up at the bottom of the slope. However, following the slope downhill only guarantees that we end up at a local minimum, a point that is lower than those close to it. If we imagine a ball rolling down a hill, it will settle at the bottom of a dip. However, there is no guarantee that it will have stopped at the lowest point—only the lowest point locally. There may be a much lower point over the next hill, but the ball can't see that, and it doesn't have enough energy to climb over the hill and find the global minimum (have another look at Figure 4.3 to see a picture of this).

Gradient descent works in the same way in two or more dimensions, and has similar (and worse) problems. The problem is that efficient downhill directions in two dimensions and higher are harder to compute locally. Standard contour maps provide beautiful images of gradients in our three-dimensional world, and if you imagine that you are walking in a hilly area aiming to get to the bottom of the nearest valley then you can get some idea of what is going on. Now suppose that you close your eyes, so that you can only feel which direction to go by moving one step and checking if you are higher up or lower down than you were. There will be places where going downwards as steeply as possible at the current point will not take you much closer to the valley bottom. There can be two reasons for this. The first is that you find a nearby local minimum, while the second is that sometimes the steepest direction is effectively across the valley, not towards the global minimum. This is shown in Figure 4.7.

All of these things are true for most of our optimisation problems, including the MLP. We don't know where the global minimum is because we don't know what the error landscape looks like; we can only compute local features of it for the place we are in at the moment. Which minimum we end up in depends on where we start. If we begin near the global minimum, then we are very likely to end up in it, but if we start near a local minimum we will probably end up there. In addition, how long it will take to get to the minimum that we do find depends upon the exact appearance of the landscape at the current point.

We can make it more likely that we find the global minimum by trying out several different starting points by training several different networks, and this is commonly done. However, we can also try to make it less likely that the algorithm will get stuck in local minima. There is a moderately effective way of doing this, which is discussed next.

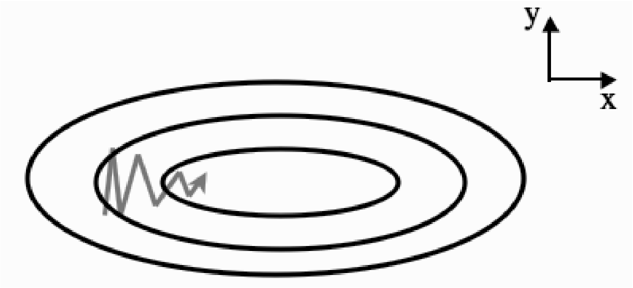

FIGURE 4.8 Adding momentum can help to avoid local minima, and also makes the dynamics of the optimisation more stable, improving convergence.

4.2.6 Picking Up Momentum

Let's go back to the analogy of the ball rolling down the hill. The reason that the ball stops rolling is because it runs out of energy at the bottom of the dip. If we give the ball some weight, then it will generate momentum as it rolls, and so it is more likely to overcome a small hill on the other side of the local minimum, and so more likely to find the global minimum. We can implement this idea in our neural network learning by adding in some contribution from the previous weight change that we made to the current one. In two dimensions it will mean that the ball rolls more directly towards the valley bottom, since on average that will be the correct direction, rather than being controlled by the local changes. This is shown in Figure 4.8.

There is another benefit to momentum. It makes it possible to use a smaller learning rate, which means that the learning is more stable. The only change that we need to make to the MLP algorithm is in Equations (4.10) and (4.11), where we need to add a second term to the weight updates so that they have the form:

$$w^{t}_{\zeta\kappa} \leftarrow w^{t-1}_{\zeta\kappa} + \eta\delta_o(\kappa)a^{\mathrm{hidden}}_{\zeta} + \alpha\Delta w^{t-1}_{\zeta\kappa}, \tag{4.17}$$

where t is used to indicate the current update and $t-1$ is the previous one. $\Delta w^{t-1}_{\zeta\kappa}$ is the previous update that we made to the weights (so $\Delta w^{t}_{\zeta\kappa} = \eta\delta_o(\kappa)a^{\mathrm{hidden}}_{\zeta} + \alpha\Delta w^{t-1}_{\zeta\kappa}$) and $0 < \alpha < 1$ is the momentum constant. Typically a value of $\alpha = 0.9$ is used. This is a very easy addition to the code, and can improve the speed of learning a lot.

```
updatew1 = eta*(np.dot(np.transpose(inputs),deltah[:,:-1])) + ⤸
momentum*updatew1
updatew2 = eta*(np.dot(np.transpose(hidden),deltao)) + momentum*updatew2
```

Another thing that can be added is known as weight decay. This reduces the size of the weights as the number of iterations increases. The argument goes that small weights are better since they lead to a network that is closer to linear (since they are close to zero, they are in the region where the sigmoid is increasing linearly), and only those weights that are essential to the non-linear learning should be large. After each learning iteration through all of the input patterns, every weight is multiplied by some constant $0 < \epsilon < 1$. This makes the network simpler and can often produce improved results, but unfortunately, it isn't fail-safe: occasionally it can make the learning significantly worse, so it should be used with care. Setting the value of ϵ is typically done experimentally.

4.2.7 Minibatches and Stochastic Gradient Descent

In Section 4.2.4 it was stated that the batch algorithm converges to a local minimum faster than the sequential algorithm, which computes the error for each input individually and then does a weight update, but that the latter is sometimes less likely to get stuck in local minima. The reason for both of these observations is that the batch algorithm makes a better estimate of the steepest descent direction, so that the direction it chooses to go is a good one, but this just leads to a local minimum.

The idea of a minibatch method is to find some happy middle ground between the two, by splitting the training set into random batches, estimating the gradient based on one of the subsets of the training set, performing a weight update, and then using the next subset to estimate a new gradient and using that for the weight update, until all of the training set have been used. The training set are then randomly shuffled into new batches and the next iteration takes place. If the batches are small, then there is often a reasonable degree of error in the gradient estimate, and so the optimisation has the chance to escape from local minima, albeit at the cost of heading in the wrong direction.

A more extreme version of the minibatch idea is to use just one piece of data to estimate the gradient at each iteration of the algorithm, and to pick that piece of data uniformly at random from the training set. So a single input vector is chosen from the training set, and the output and hence the error for that one vector computed, and this is used to estimate the gradient and so update the weights. A new random input vector (which could be the same as the previous one) is then chosen and the process repeated. This is known as stochastic gradient descent, and can be used for any gradient descent problem, not just the MLP. It is often used if the training set is very large, since it would be very expensive to use the whole dataset to estimate the gradient in that case.

4.2.8 Other Improvements

There are a few other things that can be done to improve the convergence and behaviour of the back-propagation algorithm. One is to reduce the learning rate as the algorithm progresses. The reasoning behind this is that the network should only be making large-scale changes to the weights at the beginning, when the weights are random; if it is still making large weight changes later on, then something is wrong.

Something that results in much larger performance gains is to include information about the second derivatives of the error with respect to the weights. In the back-propagation algorithm we use the first derivatives to drive the learning. However, if we have knowledge of the second derivatives as well, we can use them as well to improve the network. This will be described in more detail in Section 9.1.

4.3 THE MULTI-LAYER PERCEPTRON IN PRACTICE

The previous section looked at the design and implementation of the MLP network itself. In this section, we are going to look more at choices that can be made about the network in order to use it for solving real problems. We will then apply these ideas to using the MLP to find solutions to four different types of problem: regression, classification, time-series prediction, and data compression.

4.3.1 Amount of Training Data

For the MLP with one hidden layer there are $(L + 1) \times M + (M + 1) \times N$ weights, where L, M, N are the number of nodes in the input, hidden, and output layers, respectively. The extra +1s come from the bias nodes, which also have adjustable weights. This is a potentially huge number of adjustable parameters that we need to set during the training phase. Setting the values of these weights is the job of the back-propagation algorithm, which is driven by the errors coming from the training data. Clearly, the more training data there is, the better for learning, although the time that the algorithm takes to learn increases. Unfortunately, there is no way to compute what the minimum amount of data required is, since it depends on the problem. A rule of thumb that has been around for almost as long as the MLP itself is that you should use a number of training examples that is at least 10 times the number of weights. This is probably going to be a very large number of examples, so neural network training is a fairly computationally expensive operation, because we need to show the network all of these inputs lots of times.

4.3.2 Number of Hidden Layers

There are two other considerations concerning the number of weights that are inherent in the calculation above, which is the choice of the number of hidden nodes, and the number of hidden layers. Making these choices is obviously fundamental to the successful application of the algorithm. We will shortly see a pictorial demonstration of the fact that two hidden layers is the most that you ever need for normal MLP learning. In fact, this result can be strengthened: it is possible to show mathematically that one hidden layer with lots of hidden nodes is sufficient. This is known as the Universal Approximation Theorem; see the Further Reading section for more details. However, the bad news is that there is no theory to guide the choice of the number of hidden nodes. You just have to experiment by training networks with different numbers of hidden nodes and then choosing the one that gives the best results, as we will see in Section 4.4.

We can use the back-propagation algorithm for a network with as many layers as we like, although it gets progressively harder to keep track of which weights are being updated at any given time. Fortunately, as was mentioned above, we will never normally need more than two layers (that is, one hidden layer and the output layer). This is because we can approximate any smooth functional mapping using a linear combination of localised sigmoidal functions. There is a sketchy demonstration that two hidden layers are sufficient using pictures in Figure 4.9. The basic idea is that by combining sigmoid functions we can generate ridge-like functions, and by combining ridge-like functions we can generate functions with a unique maximum. By combining these and transforming them using another layer of neurons, we obtain a localised response (a 'bump' function), and any functional mapping can be approximated to arbitrary accuracy using a linear combination of such bumps. The way that the MLP does this is shown in Figure 4.10. We will use this idea again when we look at approximating functions, for example using radial basis functions in Chapter 5. Note that Figure 4.9 shows that two hidden layers are sufficient. In fact, they aren't necessary: one hidden layer will do, although it may require an arbitrarily large number of hidden nodes. This is known as the Universal Approximation Theorem, and the (mathematical) paper that shows this is provided in the references at the end of the chapter.

Two hidden layers are sufficient to compute these bump functions for different inputs, and so if the function that we want to learn (approximate) is continuous, the network can compute it. It can therefore approximate any decision boundary, not just the linear one that the Perceptron computed.

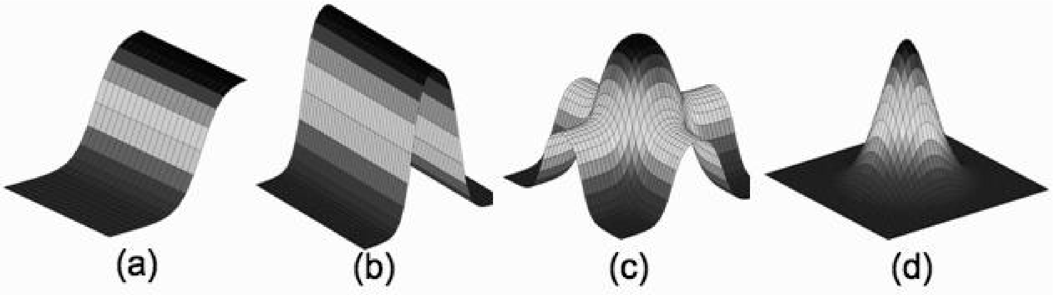

FIGURE 4.9 The learning of the MLP can be shown as the output of a single sigmoidal neuron (a), which can be added to others, including reversed ones, to get a hill shape (b). Adding another hill at 90° produces a bump (c) , which can be sharpened to any extent we want (d), with the bumps added together in the output layer. Thus the MLP learns a local representation of individual inputs.

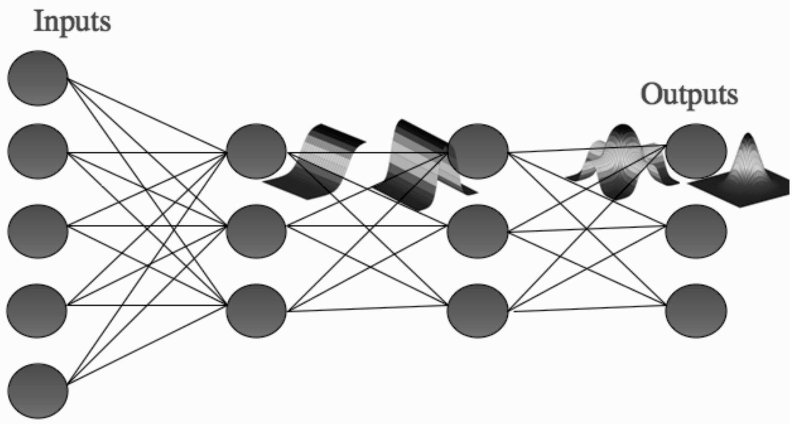

FIGURE 4.10 Schematic of the effective learning shape at each stage of the MLP.

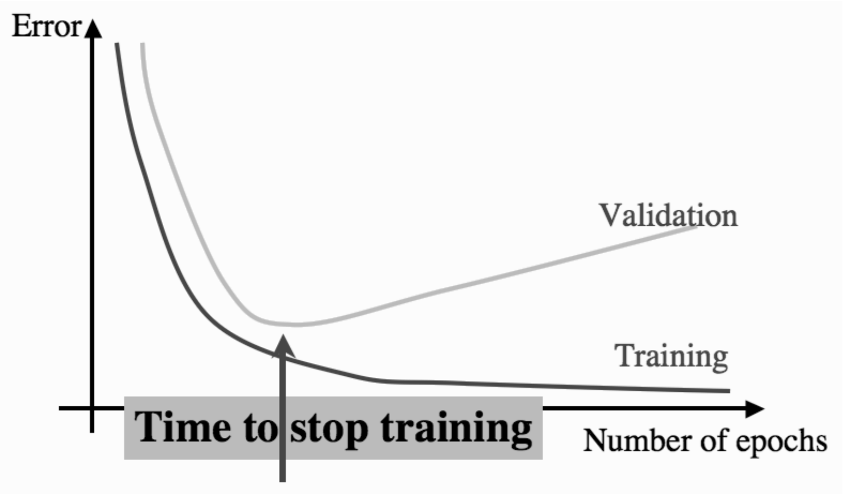

FIGURE 4.11 The effect of overfitting on the training and validation error curves, with the point at which early stopping will stop the learning marked.

4.3.3 When to Stop Learning

The training of the MLP requires that the algorithm runs over the entire dataset many times, with the weights changing as the network makes errors in each iteration. The question is how to decide when to stop learning, and this is a question that we are now ready to answer. It is unfortunate that the most obvious options are not sufficient: setting some predefined number N of iterations, and running until that is reached runs the risk that the network has overfitted by then, or not learnt sufficiently, and only stopping when some predefined minimum error is reached might mean the algorithm never terminates, or that it overfits. Using both of these options together can help, as can terminating the learning once the error stops decreasing.

However, the validation set gives us something rather more useful, since we can use it to monitor the generalisation ability of the network at its current stage of learning. If we plot the sum-of-squares error during training, it typically reduces fairly quickly during the first few training iterations, and then the reduction slows down as the learning algorithm performs small changes to find the exact local minimum. We don't want to stop training until the local minimum has been found, but, as we've just discussed, keeping on training too long leads to overfitting of the network. This is where the validation set comes in useful. We train the network for some predetermined amount of time, and then use the validation set to estimate how well the network is generalising. We then carry on training for a few more iterations, and repeat the whole process. At some stage the error on the validation set will start increasing again, because the network has stopped learning about the function that generated the data, and started to learn about the noise that is in the data itself (shown in Figure 4.11). At this stage we stop the training. This technique is called early stopping.

4.4 EXAMPLES OF USING THE MLP

This section is intended to be practical, so you should follow the examples at a computer, and add to them as you wish. The MLP is rather too complicated to enable us to work through the weight changes as we did with the Perceptron.

Instead, we shall look at some demonstrations of how to make the network learn about some data. As was mentioned above, we shall look at the four types of problems that are generally solved using an MLP: regression, classification, time-series prediction, and data compression/data denoising.

4.4.1 A Regression Problem

The regression problem we will look at is a very simple one. We will take a set of samples generated by a simple mathematical function, and try to learn the generating function (that describes how the data was made) so that we can find the values of any inputs, not just the ones we have training data for.

The function that we will use is a very simple one, just a bit of a sine wave. We'll make the data in the following way (make sure that you have NumPy imported as `np` first):

```
x = np.ones((1,40))*np.linspace(0,1,40)
t = np.sin(2*np.pi*x) + np.cos(4*np.pi*x) + np.random.randn(40)*0.2
x = x.T
t = t.T
```

The reason why we have to use the `reshape()` method is that NumPy defaults to lists for arrays that are $N \times 1$; compare the results of the `np.shape()` calls below, and the effect of the transpose operator `.T` on the array:

```
>>> x = np.linspace(0,1,40)
>>> np.shape(x)
(40,)
>>> np.shape(x.T)
(40,)
>>>
>>> x = np.linspace(0,1,40).reshape((1,40))
>>> np.shape(x)
(1, 40)
>>> np.shape(x.T)
(40, 1)
```

You can plot this data to see what it looks like (the results of which are shown in Figure 4.12) using:

```
>>> import pylab as pl
>>> pl.plot(x,t,'.')
```

We can now train an MLP on the data. There is one input value, `x` and one output value `t`, so the neural network will have one input and one output. Also, because we want the output to be the value of the function, rather than 0 or 1, we will use linear neurons at the output. We don't know how many hidden neurons we will need yet, so we'll have to experiment to see what works.

Before getting started, we need to normalise the data using the method shown in Section 3.4.5, and then separate the data into training, testing, and validation sets. For this example there are only 40 datapoints, and we'll use half of them as the training set, although that isn't very many and might not be enough for the algorithm to learn effectively. We can split the data in the ratio 50:25:25 by using the odd-numbered elements as training data, the even-numbered ones that do not divide by 4 for testing, and the rest for validation:

```
train = x[0::2,:]
test = x[1::4,:]
valid = x[3::4,:]
traintarget = t[0::2,:]
testtarget = t[1::4,:]
validtarget = t[3::4,:]
```

With that done, it is just a case of making and training the MLP. To start with, we will construct a network with three nodes in the hidden layer, and run it for 101 iterations with a learning rate of 0.25, just to see that it works:

```
>>> import mlp
>>> net = mlp.mlp(train,traintarget,3,outtype='linear')
>>> net.mlptrain(train,traintarget,0.25,101)
```

The output from this will look something like:

```
Iteration:  0  Error:  12.3704163654
Iteration:  100  Error:  8.2075961385
```

so we can see that the network is learning, since the error is decreasing. We now need to do two things: work out how many hidden nodes we need, and decide how long to train the network for. In order to solve the first problem, we need to test out different networks and see which get lower errors, but to do that properly we need to know when to stop training. So we'll solve the second problem first, which is to implement early stopping.

We train the network for a few iterations (let's make it 10 for now), then evaluate the validation set error by running the network forward (i.e., the recall phase). Learning should stop when the validation set error starts to increase. We'll write a Python program that does all the work for us. The important point is that we keep track of the validation error and stop when it starts to increase. The following code is a function within the MLP on the book website. It keeps track of the last two changes in validation error to ensure that small fluctuations in the learning don't change it from early stopping to premature stopping:

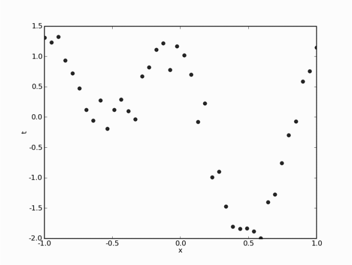

FIGURE 4.12 The data that we will learn using an MLP, consisting of some samples from a sine wave with Gaussian noise added.

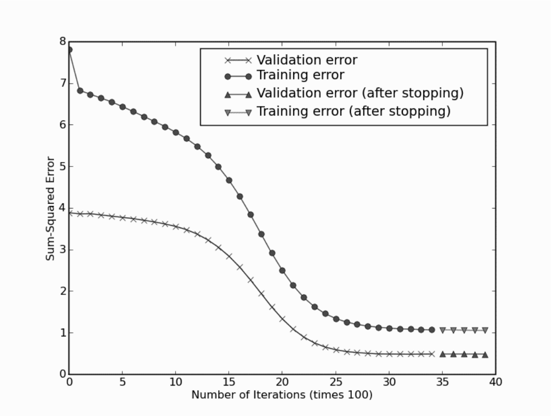

FIGURE 4.13 Plot of the error as the MLP learns (top line is total error on the training set; bottom line is on the validation set; it is larger on the training set because there are more datapoints in this set). Early-stopping halts the learning at the point where there is no line, where the crosses become triangles. The learning was continued to show that the error got slightly worse afterwards.

```
old_val_error1 = 100002
old_val_error2 = 100001
new_val_error = 100000

count = 0
while (((old_val_error1 - new_val_error) > 0.001) or ((old_val_error2 - ↩
old_val_error1)>0.001)):
    count+=1
    self.mlptrain(inputs,targets,0.25,100)
    old_val_error2 = old_val_error1
    old_val_error1 = new_val_error
    validout = self.mlpfwd(valid)
    new_val_error = 0.5*np.sum((validtargets-validout)**2)

print "Stopped", new_val_error,old_val_error1, old_val_error2
```

Figure 4.13 gives an example of the output of running the function. It plots the training and validation errors. The point at which early stopping makes the learning finish is the point where there is a missing validation datapoint. I ran it on after that so you could see that the validation error did not improve after that, and so early stopping found the correct point.

We can now return to the problem of finding the right size of network. There is one important thing to remember, which is that the weights are initialised randomly, and so the fact that a particular size of network gets a good solution once does not mean it is the right size, it could have been a lucky starting point. So each network size is run 10 times, and the average is monitored. The following table shows the results of doing this, reporting the sum-of-squares validation error, for a few different sizes of network:

No. of hidden nodes	1	2	3	5	10	25	50
Mean error	2.21	0.52	0.52	0.52	0.55	1.35	2.56
Standard deviation	0.17	0.00	0.00	0.02	0.00	1.20	1.27
Max error	2.31	0.53	0.54	0.54	0.60	3.230	3.66
Min error	2.10	0.51	0.50	0.50	0.47	0.42	0.52

Based on these numbers, we would select a network with a small number of hidden nodes, certainly between 2 and 10 (and the smaller the better, in general), since their maximum error is much smaller than a network with just 1 hidden node. Note also that the error increases once too many hidden nodes are used, since the network has too much variation for the problem. You can also do the same kind of experimentation with more hidden layers.

4.4.2 Classification with the MLP

Using the MLP for classification problems is not radically different once the output encoding has been worked out. The inputs are easy: they are just the values of the feature measurements (suitably normalised). There are a couple of choices for the outputs. The first is to use a single linear node for the output, y, and put some thresholds on the activation value of that node. For example, for a four-class problem, we could use:

$$\text{Class is:} \begin{cases} C_1 & \text{if } y \leq -0.5 \\ C_2 & \text{if } -0.5 < y \leq 0 \\ C_3 & \text{if } 0 < y \leq 0.5 \\ C_4 & \text{if } y > 0.5 \end{cases} \tag{4.18}$$

However, this gets impractical as the number of classes gets large, and the boundaries are artificial; what about an example that is very close to a boundary, say $y = 0.5$? We arbitrarily guess that it belongs to class C_3, but the neural network doesn't give us any information about how close it was to the boundary in the output, so we don't know that this was a difficult example to classify. A more suitable output encoding is called 1-of-N encoding. A separate node is used to represent each possible class, and the target vectors consist of zeros everywhere except for in the one element that corresponds to the correct class, e.g., $(0, 0, 0, 1, 0, 0)$ means that the correct result is the 4th class out of 6. We are therefore using binary output values (we want each output to be either 0 or 1).

Once the network has been trained, performing the classification is easy: simply choose the element y_k of the output vector that is the largest element of $\mathbf{y}$ (in mathematical notation, pick the y_k for which $y_k > y_j \forall j \neq k$; $\forall$ means for all, so this statement says pick the y_k that is bigger than all other possible values y_j). This generates an unambiguous decision, since it is very unlikely that two output neurons will have identical largest output values. This is known as the hard-max activation function (since the neuron with the highest activation is chosen to fire and the rest are ignored). An alternative is the soft-max function, which we saw in Section 4.2.3, and which has the effect of scaling the output of each neuron according to how large it is in comparison to the others, and making the total output sum to 1. So if there is one clear winner, it will have a value near 1, while if there are several values that are close to each other, they will each have a value of about $\frac{1}{p}$, where p is the number of output neurons that have similar values.

There is one other thing that we need to be aware of when performing classification, which is true for all classifiers. Suppose that we are doing two-class classification, and 90% of our data belongs to class 1. (This can happen: for example in medical data, most tests are negative in general.) In that case, the algorithm can learn to always return the negative class, since it will be right 90% of the time, but still a completely useless classifier! So you should generally make sure that you have approximately the same number of each class in your training set. This can mean discarding a lot of data from the over-represented class, which may seem rather wasteful. There is an alternative solution, known as novelty detection, which is to train the data on the data in the negative class only, and to assume that anything that looks different to that is a positive example. There is a reference about novelty detection in the readings at the end of the chapter.

4.4.3 A Classification Example: The Iris Dataset

As an example we are going to look at another example from the UCI Machine Learning repository. This one is concerned with classifying examples of three types of iris (flower) by the length and width of the sepals and petals and is called `iris`. It was originally worked on by R.A. Fisher, a famous statistician and biologist, who analysed it in the 1930s.

Unfortunately we can't currently load this into NumPy using `loadtxt()` because the class (which is the last column) is text rather than a number, and the `txt` in the function name doesn't mean that it reads text, only numbers in plaintext format. There are two alternatives. One is to edit the data in a text editor using search and replace, and the other is to use some Python code, such as this function:

```
def preprocessIris(infile,outfile):

    stext1 = 'Iris-setosa'
    stext2 = 'Iris-versicolor'
    stext3 = 'Iris-virginica'
    rtext1 = '0'
    rtext2 = '1'
    rtext3 = '2'

    fid = open(infile,"r")
    oid = open(outfile,"w")

    for s in fid:
        if s.find(stext1)>-1:
            oid.write(s.replace(stext1, rtext1))
        elif s.find(stext2)>-1:
            oid.write(s.replace(stext2, rtext2))
        elif s.find(stext3)>-1:
            oid.write(s.replace(stext3, rtext3))
    fid.close()
    oid.close()
```

You can then load it from the new file using `loadtxt()`. In the dataset, the last column is the class ID, and the others are the four measurements. We'll start by normalising the inputs, which we'll do in the same way as in Section 3.4.5, but using the maximum rather than the variance, and leaving the class IDs alone for now:

```
iris = np.loadtxt('iris_proc.data',delimiter=',')
iris[:,:4] = iris[:,:4]-iris[:,:4].mean(axis=0)
imax = np.concatenate((iris.max(axis=0)*np.ones((1,5)),np.abs(iris.min(
axis=0))*np.ones((1,5))),axis=0).max(axis=0)
iris[:,:4] = iris[:,:4]/imax[:4]
```

The first few datapoints will then look like:

```
>>> print iris[0:5,:]
[[-0.36142626  0.33135215 -0.7508489  -0.76741803  0. ]
 [-0.45867099 -0.04011887 -0.7508489  -0.76741803  0. ]
 [-0.55591572  0.10846954 -0.78268251 -0.76741803  0. ]
 [-0.60453809  0.03417533 -0.71901528 -0.76741803  0. ]
 [-0.41004862  0.40564636 -0.7508489  -0.76741803  0. ]]
```

We now need to convert the targets into 1-of-N encoding, from their current encoding as class 1, 2, or 3. This is pretty easy if we make a new matrix that is initially all zeroes, and simply set one of the entries to be 1:

```
# Split into training, validation, and test sets
target = np.zeros((np.shape(iris)[0],3));
indices = np.where(iris[:,4]==0)
target[indices,0] = 1
indices = np.where(iris[:,4]==1)
target[indices,1] = 1
indices = np.where(iris[:,4]==2)
target[indices,2] = 1
```

We now need to separate the data into training, testing, and validation sets. There are 150 examples in the dataset, and they are split evenly amongst the three classes, so the three classes are the same size and we don't need to worry about discarding any datapoints. We'll split them into half training, and one quarter each testing and validation. If you look at the file, you will notice that the first 50 are class 1, the second 50 class 2, etc. We therefore need to randomise the order before we split them into sets, to ensure that there are not too many of one class in one of the sets:

```
# Randomly order the data
order = range(np.shape(iris)[0])
```

```
np.random.shuffle(order)
iris = iris[order,:]
target = target[order,:]

train = iris[::2,0:4]
traint = target[::2]
valid = iris[1::4,0:4]
validt = target[1::4]
test = iris[3::4,0:4]
testt = target[3::4]
```

We're now finally ready to set up and train the network. The commands should all be familiar from earlier:

```
>>> import mlp
>>> net = mlp.mlp(train,traint,5,outtype='softmax')
>>> net.earlystopping(train,traint,valid,validt,0.1)
>>> net.confmat(test,testt)
Confusion matrix is:
[[ 16.   0.   0.]
 [  0.  12.   2.]
 [  0.   1.   6.]]
Percentage Correct:  91.8918918919
```

This tells us that the algorithm got nearly all of the test data correct, misclassifying just two examples of class 2 and one of class 3.

4.4.4 Time-Series Prediction

There is a common data analysis task known as time-series prediction, where we have a set of data that show how something varies over time, and we want to predict how the data will vary in the future. It is quite a difficult task, but a fairly important one. It is useful in any field where there is data that appears over time, which is to say almost any field. Most notable (if often unsuccessful) uses have been in trying to predict stock markets and disease patterns. The problem is that even if there is some regularity in the time-series, it can appear over many different scales. For example, there is often seasonal variation—if we plotted average temperature over several years, we would notice that it got hotter in the summer and colder in the winter, but we might not notice if there was a overall upward or downward trend to the summer temperatures, because the summer peaks are spread too far apart in the data.

The other problems with the data are practical. How many datapoints should we look at to make the prediction (i.e., how many inputs should there be to the neural network) and how far apart in time should we space those inputs (i.e., should we use every second datapoint, every 10th, or all of them)? We can write this as an equation, where we are predicting y using a neural network that is written as a function $f(\cdot)$:

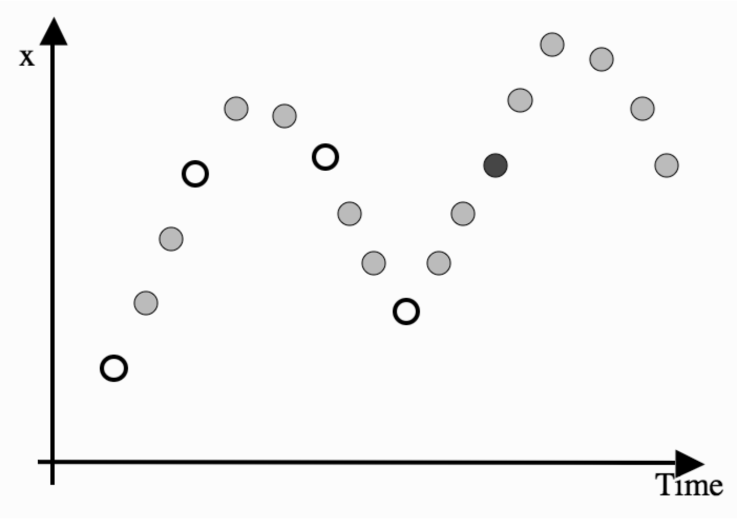

FIGURE 4.14 Part of a time-series plot, showing the datapoints and the meanings of τ and k.

$$y = x(t+\tau) = f(x(t), x(t-\tau), \ldots, x(t-k\tau)), \tag{4.19}$$

where the two questions about how many datapoints and how far apart they should be come down to choices about τ and k.

The target data for training the neural network is simple, because it comes from further up the time-series, and so training is easy. Suppose that $\tau = 2$ and $k = 3$. Then the first input data are elements $1, 3, 5$ of the dataset, and the target is element 7. The next input vector is elements $2, 4, 6$, with target 8, and then $3, 5, 7$ with target 9. You train the network by passing through the time-series (remembering to save some data for testing), and then press on into the future making predictions. Figure 4.14 shows an example of a time-series with $\tau = 3$ and $k = 4$, with a set of datapoints that make up an input vector marked as white circles, and the target coloured black.

The dataset I am going to use is available on the book website. It provides the daily measurement of the thickness of the ozone layer above Palmerston North in New Zealand (where I live) between 1996 and 2004. Ozone thickness is measured in Dobson Units, which are 0.01 mm thickness at 0 degrees Celsius and 1 atmosphere of pressure. I'm sure that I don't need to tell you that the reduction in stratospheric ozone is partly responsible for global warming and the increased incidence of skin cancer, and that in New Zealand we are fairly close to the large hole over Antarctica. What you might not know is that the thickness of the ozone layer varies naturally over the year. This should be obvious in the plot shown in Figure 4.15. A typical time-series problem is to predict the ozone levels into the future and see if you can detect an overall drop in the mean ozone level.

You can load the data using `PNoz = loadtxt('PNOz.dat')` (once you've downloaded it from the website), which will load the data and stick it into an array called PNoz. There are 4 elements to each vector: the year, the day of the year, and the ozone level and sulphur dioxide level, and there are 2855 readings. To just plot the ozone data so that you can see what it looks like, use `plot(arange(shape(PNoz)[0]),PNoz[:,2],'.')`.

The difficult bit is assembling the input vector from the time-series data. The first thing

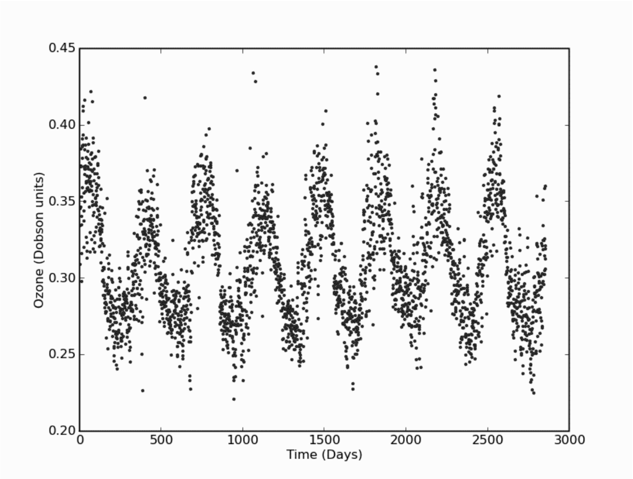

FIGURE 4.15 Plot of the ozone layer thickness above Palmerston North in New Zealand between 1996 and 2004.

is to choose values of τ and k. Then it is just a question of picking k values out of the array with spacing τ, which is a good use for the slice operator, as in this code:

```
test = inputs[-800:,:]
testtargets = targets[-800,:]
train = inputs[:-800:2,:]
traintargets = targets[:-800:2]
valid = inputs[1:-800:2,:]
validtargets = targets[1:-800:2]
```

You then need to assemble training, testing, and validation sets. However, some care is needed here since you need to ensure that they are not picked systematically into each group, (for example, if the inputs are the even-indexed datapoints, but some feature is only seen at odd datapoint times, then it will be completely missed). This can be averted by randomising the order of the datapoints first. However, it is also common to use the datapoints near the end as part of the test set; some possible results from using the MLP in this way are shown in Figure 4.16.

From here you can treat time-series as regression problems: the output nodes need to have linear activations, and you aim to minimise the sum-of-squares error. Since there are no classes, the confusion matrix is not useful. The only extra work is that in addition to testing MLPs with different numbers of input nodes and hidden nodes, you also need to consider different values of τ and k.

4.4.5 Data Compression: The Auto-Associative Network

We are now going to consider an interesting variation of the MLP. Suppose that we train the network to reproduce the inputs at the output layer (called auto-associative learning; sometimes the network is known as an autoencoder). The network is trained so that whatever

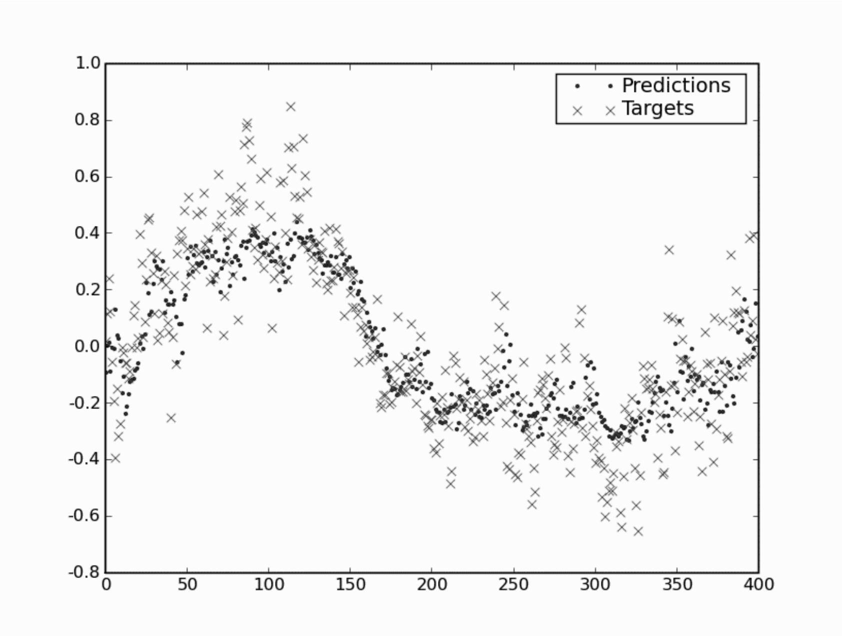

FIGURE 4.16 Plot of 400 predicted and actual output values of the ozone data using the MLP as a time-series predictor with $k = 3$ and $\tau = 2$.

you show it at the input is reproduced at the output, which doesn't seem very useful at first, but suppose that we use a hidden layer that has fewer neurons than the input layer (see Figure 4.17). This bottleneck hidden layer has to represent all of the information in the input, so that it can be reproduced at the output. It therefore performs some compression of the data, representing it using fewer dimensions than were used in the input. This gives us some idea of what the hidden layers of the MLP are doing: they are finding a different (often lower dimensional) representation of the input data that extracts important components of the data, and ignores the noise.

This auto-associative network can be used to compress images and other data. A schematic of this is shown in Figure 4.18: the 2D image is turned into a 1D vector of inputs by cutting the image into strips and sticking the strips into a long line. The values of this vector are the intensity (colour) values of the image, and these are the input values. The network learns to reproduce the same image at the output, and the activations of the hidden nodes are recorded for each image. After training, we can throw away the input nodes and first set of weights of the network. If we insert some values in the hidden nodes (their activations for a particular image; see Figure 4.19), then by feeding these activations forward through the second set of weights, the correct image will be reproduced on the output. So all we need to store are the set of second-layer weights and the activations of the hidden nodes for each image, which is the compressed version.

Auto-associative networks can also be used to denoise images, since, after training, the network will reproduce the trained image that best matches the current (noisy) input. We don't throw away the first set of weights this time, but if we feed a noisy version of the image into the inputs, then the network will produce the image that is closest to the noisy version at the outputs, which will be the version it learnt on, which is uncorrupted by noise.

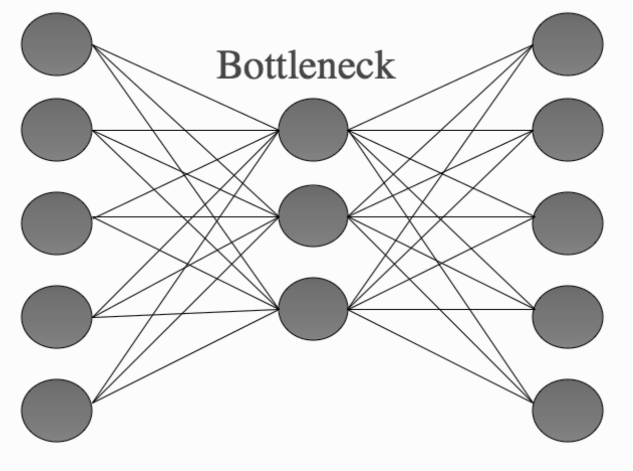

FIGURE 4.17 The auto-associative network. The network is trained to reproduce the inputs at the outputs, passing them through the bottleneck hidden layer that compresses the data.

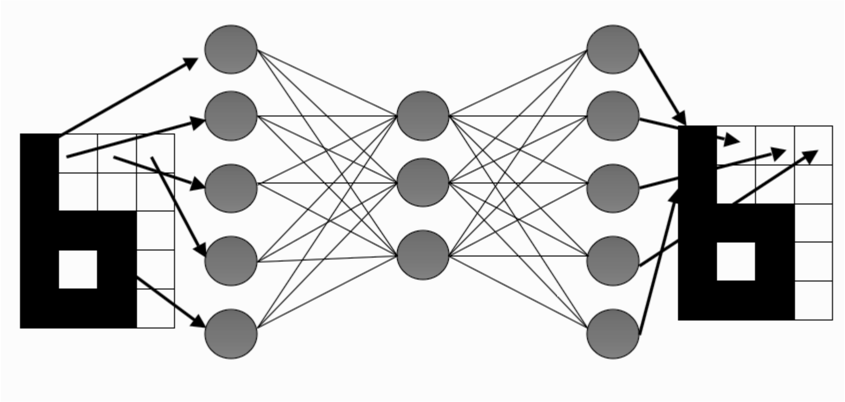

FIGURE 4.18 Schematic showing how images are fed into the auto-associative network for compression.

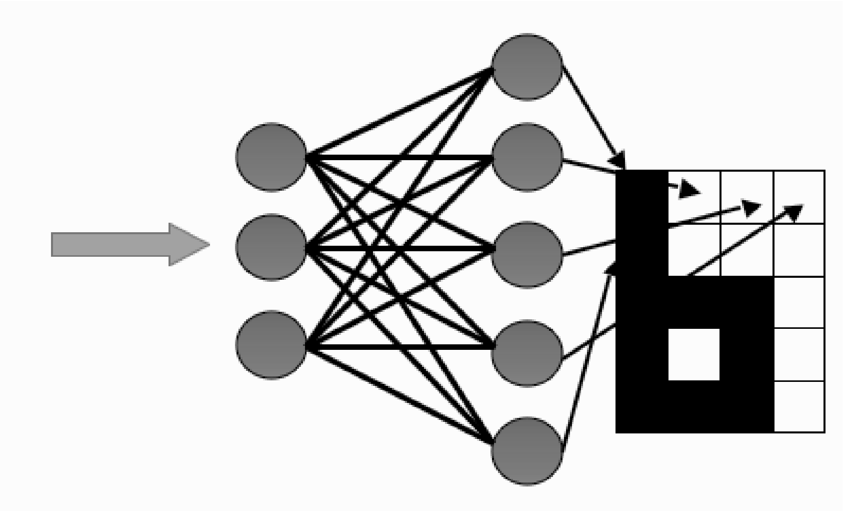

FIGURE 4.19 Schematic showing how the hidden nodes and second layer of weights can be used to regain the compressed images after the network has been trained.

You might be wondering what this representation in the hidden nodes looks like. In fact, if the nodes all have linear activation, then what the network learns to compute are the Principal Components of the input data. Principal Components Analysis (PCA) is a useful dimensionality reduction technique, and is described in Section 6.2.

4.5 A RECIPE FOR USING THE MLP

We have covered a lot in this chapter, so I'm going to give you a 'recipe' for how to use the Multi-layer Perceptron when presented with a dataset. This is, by necessity, a simplification of the problem, but it should serve to remind you of many of the important features.

Select inputs and outputs for your problem Before anything else, you need to think about the problem you are trying to solve, and make sure that you have data for the problem, both input vectors and target outputs. At this stage you need to choose what features are suitable for the problem (something we'll talk about more in other chapters) and decide on the output encoding that you will use — standard neurons, or linear nodes. These things are often decided for you by the input features and targets that you have available to solve the problem. Later on in the learning it can also be useful to re-evaluate the choice by training networks with some input feature missing to see if it improves the results at all.

Normalise inputs Rescale the data by subtracting the mean value from each element of the input vector, and divide by the variance (or alternatively, either the maximum or minus the minimum, whichever is greater).

Split the data into training, testing, and validation sets You cannot test the learning ability of the network on the same data that you trained it on, since it will generally fit that data very well (often too well, overfitting and modelling the noise in the data as well as the generating function). We generally split the data into three sets, one for training, one for testing, and then a third set for validation, which is testing how well the network is learning during training. The ratio between the sizes of the three groups depends on how much data you have, but is often around 50:25:25. If you do not have enough data for this, use cross-validation instead.

Select a network architecture You already know how many input nodes there will be, and how many output neurons. You need to consider whether you will need a hidden layer at all, and if so how many neurons it should have in it. You might want to consider more than one hidden layer. The more complex the network, the more data it will need to be trained on, and the longer it will take. It might also be more subject to overfitting. The usual method of selecting a network architecture is to try several with different numbers of hidden nodes and see which works best.

Train a network The training of the neural network consists of applying the Multi-layer Perceptron algorithm to the training data. This is usually run in conjunction with early stopping, where after a few iterations of the algorithm through all of the training data, the generalisation ability of the network is tested by using the validation set. The neural network is very likely to have far too many degrees of freedom for the problem, and so after some amount of learning it will stop modelling the generating function of the data, and start to fit the noise and inaccuracies inherent in the training data. At this stage the error on the validation set will start to increase, and learning should be stopped.

Test the network Once you have a trained network that you are happy with, it is time to use the test data for the first (and only) time. This will enable you to see how well the network performs on some data that it has not seen before, and will tell you whether this network is likely to be usable for other data, for which you do not have targets.

4.6 DERIVING BACK-PROPAGATION

This section derives the back-propagation algorithm. This is important to understand how and why the algorithm works. There isn't actually that much mathematics involved except some slightly messy algebra. In fact, there are only three things that you really need to know. One is the derivative (with respect to x) of $\frac{1}{2}x^2$, which is x, and another is the chain rule, which says that $\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx}$. The third thing is very simple: $\frac{dy}{dx} = 0$ if y is not a function of x. With those three things clear in your mind, just follow through the algebra, and you'll be fine. We'll work in simple steps.

4.6.1 The Network Output and the Error

The output of the neural network (the end of the forward phase of the algorithm) is a function of three things:

- the current input ($\mathbf{x}$)
- the activation function $g(\cdot)$ of the nodes of the network
- the weights of the network ($\mathbf{v}$ for the first layer and $\mathbf{w}$ for the second)

We can't change the inputs, since they are what we are learning about, nor can we change the activation function as the algorithm learns. So the weights are the only things that we can vary to improve the performance of the network, i.e., to make it learn. However, we do need to think about the activation function, since the threshold function that we used for the Perceptron is not differentiable (it has a discontinuity at 0). We'll think about a better one in Section 4.6.3, but first we'll think about the error of the network. Remember that we have run the algorithm forwards, so that we have fed the inputs ($\mathbf{x}$) into the algorithm, used the first set of weights ($\mathbf{v}$) to compute the activations of the hidden neurons, then those activations and the second set of weights ($\mathbf{w}$) to compute the activations of the output neurons, which are the outputs of the network ($\mathbf{y}$). Note that I'm going to use i to be an index over the input nodes, j to be an index over the hidden layer neurons, and k to be an index over the output neurons.

4.6.2 The Error of the Network

When we discussed the Perceptron learning rule in the previous chapter we motivated it by minimising the error function $E = \sum_{k=1}^{N} y_k - t_k$. We then invented a learning rule that made this error smaller. We are going to do much better this time, because everything is computed from the principles of gradient descent.

To begin with, let's think about the error of the network. This is obviously going to have something to do with the difference between the outputs $\mathbf{y}$ and the targets $\mathbf{t}$, but I'm going to write it as $E(\mathbf{v}, \mathbf{w})$ to remind us that the only things that we can change are the weights $\mathbf{v}$ and $\mathbf{w}$, and that changing the weights changes the output, which in turn changes the error.

For the Perceptron we computed the error as $E = \sum_{k=1}^{N} y_k - t_k$, but there are some problems with this: if $t_k > y_k$, then the sign of the error is different to when $y_k > t_k$, so if we have lots of output nodes that are all wrong, but some have positive sign and some have negative sign, then they might cancel out. Instead, we'll choose the sum-of-squares error function, which calculates the difference between y_k and t_k for each node k, squares them, and adds them together (I've missed out the $\mathbf{v}$ in $E(\mathbf{w})$ because we don't use them here):

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2 && (4.20)\\ &= \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{j=0}^{M} w_{jk}a_j\right) - t_k\right]^2 && (4.21)\end{aligned}$$

The second line adds in the input from the hidden layer neurons and the second-layer weights to decide on the activations of the output neurons. For now we're going to think about the Perceptron and index the input nodes by i and the output nodes by k, so Equation (4.21) will be replaced by:

$$\frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{i=0}^{L} w_{ik}x_i\right) - t_k\right]^2. \qquad (4.22)$$

Now we can't differentiate the threshold function, which is what the Perceptron used for $g(\cdot)$, because it has a discontinuity (sudden jump) at the threshold value. So I'm going to miss it out completely for the moment. Also, for the Perceptron there are no hidden neurons, and so the activation of an output neuron is just $y_\kappa = \sum_{i=0}^{L} w_{i\kappa}x_i$ where x_i is the value of an input node, and the sum runs over the number of input nodes, including the bias node.

We are going to use a gradient descent algorithm that adjusts each weight $w_{\iota\kappa}$ for fixed values of ι and κ, in the direction of the negative gradient of $E(\mathbf{w})$. In what follows, the notation ∂ means the partial derivative, and is used because there are lots of different functions that we can differentiate E with respect to: all of the different weights. If you don't know what a partial derivative is, think of it as being the same as a normal derivative, but taking care that you differentiate in the correct direction. The gradient that we want to know is how the error function changes with respect to the different weights:

$$\begin{aligned}\frac{\partial E}{\partial w_{\iota\kappa}} &= \frac{\partial}{\partial w_{\iota\kappa}}\left(\frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2\right) \\ &= \frac{1}{2}\sum_{k=1}^{N} 2(y_k - t_k)\frac{\partial}{\partial w_{\iota\kappa}}\left(y_k - \sum_{i=0}^{L} w_{i\kappa}x_i\right) \qquad (4.23) \\ & \qquad (4.24)\end{aligned}$$

Now t_k is not a function of any of the weights, since it is a value given to the algorithm, so $\frac{\partial t_k}{\partial w_{\iota\kappa}} = 0$ for all values of k, ι, κ, and the only part of $\sum_{i=0}^{L} w_{i\kappa}x_i$ that is a function of $w_{\iota\kappa}$ is when $i = \iota$, that is $w_{\iota\kappa}$ itself, which has derivative 1. Hence:

$$\frac{\partial E}{\partial w_{\iota\kappa}} = \sum_{k=1}^{N}(t_k - y_k)(-x_\iota). \qquad (4.25)$$

Now the idea of the weight update rule is that we follow the gradient downhill, that is, in the direction $-\frac{\partial E}{\partial w_{\iota\kappa}}$. So the weight update rule (when we include the learning rate η) is:

$$w_{\iota\kappa} \leftarrow w_{\iota\kappa} + \eta(t_\kappa - y_\kappa)x_\iota, \qquad (4.26)$$

which hopefully looks familiar (see Equation (3.3)). Note that we are computing y_κ differently: for the Perceptron we used the threshold activation function, whereas in the work above we ignored the threshold function. This isn't very useful if we want units that act like neurons, because neurons either fire or do not fire, rather than varying continuously. However, if we want to be able to differentiate the output in order to use gradient descent, then we need a differentiable activation function, so that's what we'll talk about now.

4.6.3 Requirements of an Activation Function

In order to model a neuron we want an activation function that has the following properties:

- it must be differentiable so that we can compute the gradient
- it should saturate (become constant) at both ends of the range, so that the neuron either fires or does not fire
- it should change between the saturation values fairly quickly in the middle

There is a family of functions called sigmoid functions because they are S-shaped (see Figure 4.5) that satisfy all those criteria perfectly. The form in which it is generally used is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}, \qquad (4.27)$$

where β is some positive parameter. One happy feature of this function is that its derivative has an especially nice form:

$$\begin{aligned}
g'(h) = \frac{dg}{dh} &= \frac{d}{dh}(1+e^{-\beta h})^{-1} && (4.28)\\
&= -1(1+e^{-\beta h})^{-2}\frac{de^{-\beta h}}{dh} && (4.29)\\
&= -1(1+e^{-\beta h})^{-2}(-\beta e^{-\beta h}) && (4.30)\\
&= \frac{\beta e^{-\beta h}}{(1+e^{-\beta h})^{2}} && (4.31)\\
&= \beta g(h)(1-g(h)) && (4.32)\\
&= \beta a(1-a) && (4.33)
\end{aligned}$$

We'll be using this derivative later. So we've now got an error function and an activation function that we can compute derivatives of. We will consider some other possible activation functions for the output neurons in Section 4.6.5 and an alternative error function in Section 4.6.6. The next thing to do is work out how to use them in order to adjust the weights of the network.

4.6.4 Back-Propagation of Error

It is now that we'll need the chain rule that I reminded you of earlier. In the form that we want, it looks like this:

$$\frac{\partial E}{\partial w_{\zeta\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}}, \tag{4.34}$$

where $h_\kappa = \sum_{j=0}^{M} w_{j\kappa}a_\zeta$ is the input to output-layer neuron κ; that is, the sum of the activations of the hidden-layer neurons multiplied by the relevant (second-layer) weights. So what does Equation (4.34) say? It tells us that if we want to know how the error at the output changes as we vary the second-layer weights, we can think about how the error changes as we vary the input to the output neurons, and also about how those input values change as we vary the weights.

Let's think about the second term first (in the third line we use the fact that $\frac{\partial w_{j\kappa}}{\partial w_{\zeta\kappa}} = 0$ for all values of j except $j = \zeta$, when it is 1):

$$\begin{aligned}
\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}} &= \frac{\partial \sum_{j=0}^{M} w_{j\kappa}a_j}{\partial w_{\zeta\kappa}} && (4.35)\\
&= \sum_{j=0}^{M}\frac{\partial w_{j\kappa}a_j}{\partial w_{\zeta\kappa}} && (4.36)\\
&= a_\zeta. && (4.37)
\end{aligned}$$

Now we can worry about the $\frac{\partial E}{\partial h_\kappa}$ term. This term is important enough to get its own term, which is the error or delta term:

$$\delta_o(\kappa) = \frac{\partial E}{\partial h_\kappa}. \tag{4.38}$$

Let's start off by trying to compute this error for the output. We can't actually compute it directly, since we don't know much about the inputs to a neuron, we just know about its output. That's fine, because we can use the chain rule again:

$$\delta_o(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa}. \tag{4.39}$$

Now the output of output layer neuron κ is

$$y_\kappa = g(h_\kappa^{\text{output}}) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{\text{hidden}}\right), \tag{4.40}$$

where $g(\cdot)$ is the activation function. There are different possible choices for $g(\cdot)$ including the sigmoid function given in Equation (4.27), so for now I'm going to leave it as a function. I've also started labelling whether h refers to an output or hidden layer neuron, just to avoid any possible confusion. We don't need to worry about this for the activations, because we use y for the activations of output neurons and a for hidden neurons. In Equation (4.43) I've substituted in the expression for the error at the output, which we computed in Equation (4.21):

$$\begin{aligned}
\delta_o(\kappa) &= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)}\frac{\partial g\left(h_\kappa^{\text{output}}\right)}{\partial h_\kappa^{\text{output}}} &(4.41)\\
&= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)} g'\left(h_\kappa^{\text{output}}\right) &(4.42)\\
&= \frac{\partial}{\partial g\left(h_\kappa^{\text{output}}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g(h_k^{\text{output}}) - t_k\right)^2\right] g'\left(h_\kappa^{\text{output}}\right) &(4.43)\\
&= \left(g(h_\kappa^{\text{output}}) - t_\kappa\right) g'(h_\kappa^{\text{output}}) &(4.44)\\
&= (y_\kappa - t_\kappa) g'(h_\kappa^{\text{output}}), &(4.45)
\end{aligned}$$

where $g'(h_\kappa)$ denotes the derivative of g with respect to h_κ. This will change depending upon which activation function we use for the output neurons, so for now we will write the update equation for the output layer weights in a slightly general form and pick it up again at the end of the section:

$$\begin{aligned}
w_{\zeta\kappa} &\leftarrow w_{\zeta\kappa} - \eta\frac{\partial E}{\partial w_{\zeta\kappa}}\\
&= w_{\zeta\kappa} - \eta\delta_o(\kappa)a_\zeta. &(4.46)
\end{aligned}$$

where we are using the minus sign because we want to go downhill to minimise the error.

We don't actually need to do too much more work to get to the first layer weights, v_ι, which connects input ι to hidden node ζ. We need the chain rule (Equation (4.34)) one more time to get to these weights, remembering that we are working backwards through the network so that k runs over the output nodes. The way to think about this is that each hidden node contributes to the activation of all of the output nodes, and so we need to consider all of these contributions (with the relevant weights).

$$\begin{aligned}
\delta_h(\zeta) &= \sum_{k=1}^{N} \frac{\partial E}{\partial h_k^{\text{output}}} \frac{\partial h_k^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} && (4.47) \\
&= \sum_{k=1}^{N} \delta_o(k) \frac{\partial h_k^{\text{output}}}{\partial h_\zeta^{\text{hidden}}}, && (4.48)
\end{aligned}$$

where we obtain the second line by using Equation (4.38). We now need a nicer expression for that derivative. The important thing that we need to remember is that inputs to the output layer neurons come from the activations of the hidden layer neurons multiplied by the second layer weights:

$$h_\kappa^{\text{output}} = \sum_{j=0}^{M} w_{j\kappa} g\left(h_j^{\text{hidden}}\right), \tag{4.49}$$

which means that:

$$\frac{\partial h_\kappa^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} = \frac{\partial g\left(\sum_{j=0}^{M} w_{j\kappa} h_j^{\text{hidden}}\right)}{\partial h_j^{\text{hidden}}}. \tag{4.50}$$

We can now use a fact that we've used before, which is that $\frac{\partial h_\zeta}{\partial h_j} = 0$ unless $j = \zeta$, when it is 1. So:

$$\frac{\partial h_\kappa^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} = w_{\zeta\kappa} g'(a_\zeta). \tag{4.51}$$

The hidden nodes always have sigmoidal activation functions, so that we can use the derivative that we computed in Equation (4.33) to get that $g'(a_\zeta) = \beta a_\zeta(1 - a_\zeta)$, which allows us to compute:

$$\delta_h(\zeta) = \beta a_\zeta(1 - a_\zeta) \sum_{k=1}^{N} \delta_o(k) w_\zeta. \tag{4.52}$$

This means that the update rule for v_ι is:

$$\begin{aligned}
v_\iota &\leftarrow v_\iota - \eta \frac{\partial E}{\partial v_\iota} \\
&= v_\iota - \eta a_\zeta(1 - a_\zeta) \left(\sum_{k=1}^{N} \delta_o(k) w_\zeta\right) x_\iota. && (4.53)
\end{aligned}$$

Note that we can do exactly the same computations if the network has extra hidden layers between the inputs and the outputs. It gets harder to keep track of which functions we should be differentiating, but there are no new tricks needed.

4.6.5 The Output Activation Functions

The sigmoidal activation function that we have created is aimed at making the nodes act a bit like neurons, either firing or not firing. This is very important in the hidden layer, but earlier in the chapter we have observed two cases where it is not suitable for the output neurons. One was regression, where we want the output to be continuous, and one was multi-class classification, where we want only one of the output neurons to fire. We identified possible activation functions for these cases, and here we will derive the delta term δ_o for them. As a reminder, the three functions are:

Linear $y_\kappa = g(h_\kappa) = h_\kappa$

Sigmoidal $y_\kappa = g(h_\kappa) = 1/(1 + \exp(-\beta h_\kappa)$

Soft-max $y_\kappa = g(h_\kappa) = \exp(h_\kappa)/\sum_{k=1}^N \exp(h_k)$

For each of these we need the derivative with respect to each of the output weights so that we can use Equation (4.45).

This is easy for the first two cases, and tells us that for linear outputs $\delta_o(\kappa) = (y_\kappa - t(\kappa))y_\kappa$, while for sigmoidal outputs it is $\delta_o(\kappa) = \beta(y_\kappa - t(\kappa))y_\kappa(1 - y_\kappa)$.

However, we have to do some more work for the soft-max case, since we haven't differentiated it yet. If we write it as:

$$\frac{\partial}{\partial h_K} y_\kappa = \frac{\partial}{\partial h_K}\left(\exp(h_\kappa)\left(\sum_{k=1}^N \exp(h_k)\right)^{-1}\right) \tag{4.54}$$

then the problem becomes clear: we have a product of two things to differentiate, and three different indices to worry about. Further, the k index runs over all the output nodes, and so includes K and κ within it. There are two cases: either $K = \kappa$, or it does not. If they are the same, then we can write that $\frac{\partial \exp(h_\kappa)}{\partial h_{kappa}} = \exp(h_\kappa)$ to get (where the last term in the first line comes from the use of the chain rule):

$$\begin{aligned}
&\frac{\partial}{\partial h_\kappa}\left(\exp(h_\kappa)\left(\sum_{k=1}^N \exp(h_k)\right)^{-1}\right) \\
&= \exp(h_\kappa)\left(\sum_{k=1}^N \exp(h_k)\right)^{-1} - \exp(h_\kappa)\left(\sum_{k=1}^N \exp(h_k)\right)^{-2} \exp(h_\kappa) \\
&= y_\kappa(1 - y_\kappa).
\end{aligned} \tag{4.55}$$

For the case where $K \neq \kappa$ things are a little easier, and we get:

$$\begin{aligned}
\frac{\partial}{\partial h_K}\exp(h_\kappa)\left(\sum_{k=1}^N \exp(h_k)\right)^{-1} &= -\exp(h_\kappa)\exp(h_K)\left(\sum_{k=1}^N \exp(h_k)\right)^{-2} \\
&= -y_\kappa y_K.
\end{aligned} \tag{4.56}$$

Using the Kronecker delta function δ_{ij}, which is 1 if $i = j$ and 0 otherwise, we can write the two cases in one equation to get the delta term:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (\delta_{\kappa K} - y_K). \tag{4.57}$$

The very last thing to think about is whether or not the sum-of-squares error function is always the best one to use.

4.6.6 An Alternative Error Function

We have been using the sum-of-squares error function throughout this chapter. It is easy to compute and works well in general; we will see another benefit of it in Section 9.2. However, for classification tasks we are assuming that the outputs represent different, independent classes, and this means that we can think of the activations of the nodes as giving us a probability that each class is the correct one.

In this probabilistic interpretation of the outputs, we can ask how likely we are to see each target given the set of weights that we are using. This is known as the likelihood and the aim is to maximise it, so that we predict the targets as well as possible. If we have a 1 output node, taking values 0 or 1, then the likelihood is:

$$p(t|\mathbf{w}) = y_k^{t_k} \left(1 - y_k\right)^{1-t_k}. \tag{4.58}$$

In order to turn this into a minimisation function we put a minus sign in front, and it will turn out to be useful to take the logarithm of it as well, which produces the cross-entropy error function, which is (for N output nodes):

$$E_{\mathrm{ce}} = -\sum_{k=1}^{N} t_k \ln(y_k), \tag{4.59}$$

where ln is the natural logarithm. This error function has the nice property that when we use the soft-max function the derivatives are very easy because the exponential and logarithm are inverse functions, and so the delta term is simply $\delta_o(\kappa) = y_\kappa - t_\kappa$.

FURTHER READING

The original papers describing the back-propagation algorithm are listed here, along with a well-known introduction to neural networks:

- D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323(99):533–536, 1986a.
- D.E. Rumelhart, J.L. McClelland, and the PDP Research Group, editors. *Parallel Distributed Processing.* MIT Press, Cambridge, MA, 1986b.
- R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, 1987.

For more on the Universal Approximation Theorem, which shows that one hidden layer is sufficient, some references (which are not for the mathematically faint-hearted) are:

- G. Cybenko. Approximations by superpositions of sigmoidal functions. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.
- Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

If you are interested in novelty detection, then a review article is:

- S. Marsland. Novelty detection in learning systems. *Neural Computing Surveys*, 3: 157–195, 2003.

The topics in this chapter are covered in any book on machine learning and neural networks. Different treatments are given by:

- Sections 5.1–5.3 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.
- Section 5.4 of J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City, CA, USA, 1991.
- Sections 4.4–4.7 of T. Mitchell. *Machine Learning.* McGraw-Hill, New York, USA, 1997.

PRACTICE QUESTIONS

Problem 4.1 Work through the MLP shown in Figure 4.2 to ensure that it does solve the XOR problem.

Problem 4.2 Suppose that the local power company wants to predict electricity demand for the next 5 days. They have the data about daily demand for the last 5 years. Typically, the demand will be a number between 80 and 400.

1. Describe how you could use an MLP to make the prediction. What parameters would you have to choose, and what do you think would be sensible values for them?
2. If the weather forecast for the next day, being the estimated temperatures for daytime and nighttime, was available, how would you add that into your system?
3. Do you think that this system would work well for predicting power consumption? Are there demands that it would not be able to predict?

Problem 4.3 Design an MLP that would learn to hyphenate words correctly. You would have a dictionary that shows correct hyphenation examples for lots of words, and you need to choose methods of encoding the inputs and outputs that say whether a hyphen is allowed between each pair of letters. You should also describe how you would perform training and testing.

Problem 4.4 Would the previous system be better than just using the dictionary?

Problem 4.5 Modify the code on the book website to work sequentially rather than in batch mode. Compare the results on the iris dataset.

Problem 4.6 Modify the code so that it performs minibatch optimisation and then stochastic gradient descent (both described in Section 4.2.7) and compare the results with using the standard algorithm on the Pima Indian dataset that was described in Section 3.4.4. Experiment with different sizes for the minibatches.

Problem 4.7 Modify the code to allow another hidden layer to be used. You will have to work out the gradient as well in order to compute the weight updates for the extra layer of weights. Test this new network on the Pima Indian dataset that was described in Section 3.4.4.

Problem 4.8 Modify the code to use the alternative error term in Section 4.6.6 and see what difference it makes for classification problems.

Problem 4.9 A hospital manager wants to predict how many beds will be needed in the geriatric ward. He asks you to design a neural network method for making this prediction. He has data for the last 5 years that cover:

- The number of people in the geriatric ward each week.
- The weather (average day and night temperatures).
- The season of the year (spring, summer, autumn, winter).
- Whether or not there was an epidemic on (use a binary variable: yes or no).

Design a suitable MLP for this problem, considering how you would choose the number of hidden neurons, the inputs (and whether there are any other inputs you need) and the preprocessing, and whether or not you would expect the system to work.

Problem 4.10 Look into the MNIST dataset that is available via the book website. Implement an MLP to learn about them and test different numbers of hidden nodes.

Problem 4.11 A recurrent network has some of its outputs connected to its own inputs, so that the outputs at time t are fed back into the network at time $t+1$. This can be a different way to deal with time-series data. Modify the MLP code so that it acts as a recurrent network, and test it out on the Palmerston North ozone data on the book website.

Problem 4.12 The alternative activation function that can be used in $\tanh(h)$. Show that $\tanh(h) = 2g(2h) - 1$, where g is given by Equation (4.2). Use this to show that there is an exactly equivalent MLP using the tanh activation function. Modify the code to implement it.