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:

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

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:

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

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:

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

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:

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

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:

$$\frac{\partial E}{\partial w_{\zeta\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}}, \qquad (4.34)$$

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:

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

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):

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

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.