CHAPTER 1

Introduction and overview

Our goal in this book is to illustrate, in concrete ways that you will be able to replicate on your own, properties of connectionist models which we believe are particularly relevant to developmental issues. Our emphasis on the principles and functional characteristics of these models is what sets this book apart from many of the other excellent introductions to neural networks (some of which the reader may wish to consult to get a broader view of architectures and techniques not covered in this volume).

In Chapters 3 through 12 we explore a set of simulations which focus on various aspects of connectionist models. However, we are aware that our readers will vary widely with regard to the knowledge and experience they bring with them. Before leaping into the simulations, therefore, there are several things we think it will be useful to do. The first three chapters therefore provide an overview of some of the technical aspects of doing simulations. In this chapter we introduce some of the terminological and notational conventions which will be used in this book, and provide a brief overview of network dynamics and learning. Our intent is modest here; we want to give the reader enough of an understanding of network mechanics so that he or she will understand what actions are being done by the simulator that is used in the subsequent exercises. Our goal in Chapter 2 is to make explicit the assumptions which underlie the simulation methodology we will be using. It is easy to do simulations; it's not as easy to do them well and to good purpose! In Chapter 3 we describe the software which will be used in this book. These first three chapters thus contain introductory material, some of which the experienced reader might wish to skip (although we urge that it at least be skimmed to ensure nothing vital is missed).

Nodes and connections

Neural networks are actually quite simple. They are made up of a few basic building blocks: *nodes* and *connections.* Figure 1.1 shows sev-

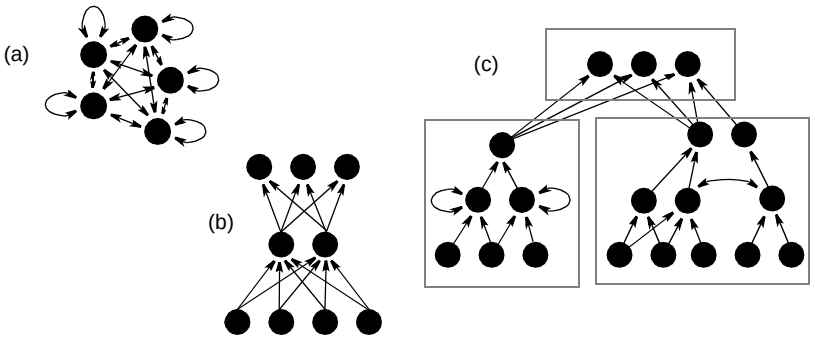


FIGURE 1.1 Various types of connectionist architectures. (a) A fully recurrent network; (b) a three-layer feedforward network; (c) a complex network consisting of several modules. Arrows indicate direction of flow of excitation or inhibition.

eral sample networks, where nodes are shown as filled circles and connections as lines between them.

Nodes are simple processing units. They are often likened to neurons. Like neurons, they receive inputs from other sources. These inputs may be excitatory or inhibitory. In the case of the neuron, excitatory inputs tend to increase the neuron's rate of firing, while inhibitory inputs decrease the firing rate. This notion of firing rate is captured in nodes by giving them a real-valued number which is called their *activation*. (We might think of higher activation values as corresponding to greater firing rates, and lower activation values to lower firing rates.)

The input to a given node comes either from other nodes or from some external source, and travels along connection lines. In most connectionist models, it is useful to allow connections between different nodes to have different potency, or connection strengths. The strength of a connection may also be represented by a real-valued number, and is usually called the *connection weight.* The input which flows from one node to another is multiplied by the connection weight. If the con-

nection weight from one node to another is a negative number, then the input from the first to the second node may be thought of as being inhibitory; if positive, it is excitatory.

If we looked in more detail at a node, we might wish to represent it as in Figure 1.2. This shows the node as a circle, with input connec-

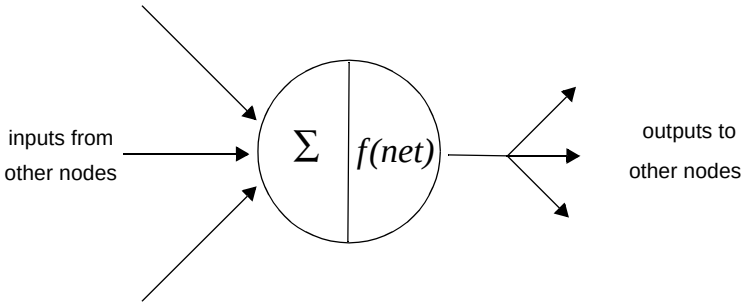


FIGURE 1.2 Detailed look at a single node. Inputs to the node are typically summed (indicated by the symbol Σ on the left); the net input is then passed through an *activation function* (shown as *f(net)*) which yields the node's activation. This value is then sent on to other nodes.

tions feeding into it, and output connections leading from it. Each input line or connection represents the flow of activity perhaps from some other neuron or from some external source (such as light falling on some photosensitive retinal cell).

For most of the nodes that you will meet in this book, the cell body performs two operations. The first is the simple adding together of the net inputs to the unit. Each input (from different nodes) is itself a number, which can be calculated by multiplying the activation value of the sending node by the weight on the connection from the sending to receiving node (note that connections between nodes may be asymmetric). If we use the letter i to index the receiving node, a_j to index the activation of those nodes which send to node i, and w_{ij} to refer to the weights on the connections from nodes j to node i, then we may calculate the net input to node i as

$$\text{net}_i = \sum w_{ij} a_j \qquad \textbf{(EQ 1.1)}$$

A word on notation

What does Σ mean?

When you see this symbol (called "sigma") it means that something is going to be added. Here we use it by itself to indicate that all the inputs will be summed together. Another example might be

$$\sum a_i$$

This means that we have some number of a's to be summed. We use i as a counter, beginning with i equal to 0 (by convention). We sum the first a (a_0), the second, (a_1), etc., till we have counted through all "i" of them.

What does f (net) mean?

This is read "f of net." It means we have some operation or set of operations which we want to carry out on the quantity contained in the variable named "net" (which we use to denote the net input to a node). We call these operations a function, and say that we are applying the function "f" to the quantity "net." (Note that so far, we have left unspecified just what that function is.)

What a node actually *does* with that net input is another matter. In the simplest case the node's activation is the same as its input. In this case the activation function (*f(net)*) is just the identity function. But one can easily imagine cases where the activation of a node (its output) might require a certain amount of "juice" before it actually starts to

Unpacking Equation 1.1

This equation tells us how to calculate the total input coming into some node. We call that node i so that the procedure can be general. Let's assume here we are dealing with node 5, so i equals 5. We have already said that the Σ means to add some things together; the subscript under the sigma (j) tells us how many things need to be added. The things to be added are indicated by the letters that follow—the w_{ij} and the a_j. By convention, two adjacent variables mean the numbers they represent are to be multiplied first. So, how do we calculate all of this?

We begin by setting our counter (j) to 0 (again, by convention). That means a_j is a_0, or the activation of the 0th node (whatever it happens to be). w_{ij} becomes the $w_{5,0}$ or the weight going to node 5 from node 0. We multiply these two numbers together and save the result. Then we set the counter j to 1, and calculate the product of a_1 (the activation of node 1) times $w_{5,1}$ (the weight to node 5 from node 1). We save that result. We continue till we have gone through all of the j's. Finally, we add them up (the sigma). This operation is often called a "sum of products."

fire. This is in fact typical of real neurons: In order to begin firing, the input must exceed a certain threshold.

In many neural networks, the activation function is nonlinear function of the input, resembling a sigmoid. In the networks we will

be using here, nodes' activations are given by the logistic function shown in

$$a_i = \frac{1}{1 + e^{-net_i}} \qquad \textbf{(EQ 1.2)}$$

(where a_i refers to the activation (output) of node$_i$, net_i is the net activation flowing into node$_i$, and e is the exponential). This equation tells us what the output of a node will be, for any given net input. If we graph this relationship, as we have done in Figure 1.3, we get a

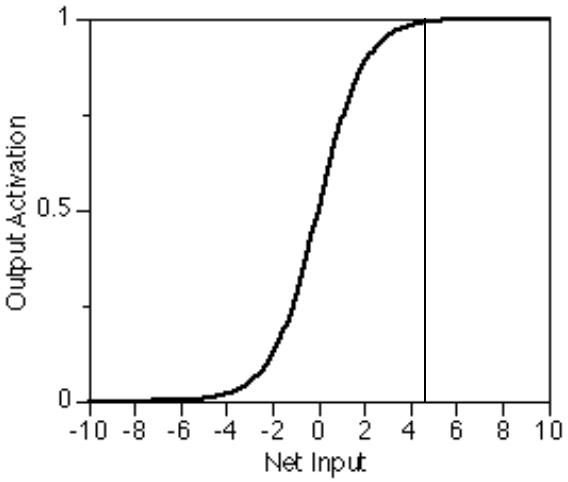


FIGURE 1.3 The sigmoid activation function often used for nodes in neural networks.

better idea of how a node's output is related to its input.

We see that over a wide range of inputs (roughly, inputs greater than 4.0, or less than -4.0), such nodes exhibit an all-or-nothing response—they are either fully "on" (output their maximum values of 1.0) or "off" (output their minimum values of 0.0). Within the range of -4.0 to 4.0, on the other hand, the nodes show a greater sensitivity and their output is capable of making fine discriminations between different inputs. ***This nonlinear response lies at the heart of much of what makes such networks interesting.***

A concrete example

Although the dynamics of node activations are fairly straightforward, it is easy to be confused between the *input* which a node receives, and its *output*. Calculating these quantities is one of the things a simulator does, but these can also be calculated by hand and it is useful to do this a few times to be sure you understand what is going on.

To place things in context, let us first assemble a simple network. A neural network consists of a collection of nodes of the sort that we discussed in the previous section. When we talk about the *architecture* of a network we are referring to the particular way in which that network is assembled, or its pattern of connectivity. There are many types of architectures, and we shall consider a number of them in this book.

A very common architecture is one in which nodes are connected to each other in a layered fashion. For example, consider the neural network depicted in Figure 1.4. This network consists of four nodes

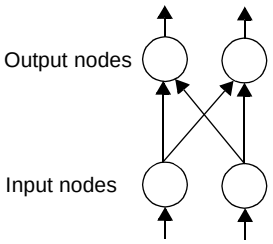


FIGURE 1.4 A two-layered feedforward network.

organized into two layers: an input layer and an output layer. Within the input layer, all the nodes have connections which project to the output layer. There are no connections between nodes within a layer (no *intra*-level connections). Furthermore, in this architecture the nodes do not possess *recurrent* connections, i.e., they do not have connections which project back to themselves or to lower levels. Thus, in this network, the flow of activity is in one direction only, from the input layer to the output layer. We call these types of networks "feedforward networks." In contrast, "recurrent networks" may possess both intra- and inter-level connections as well as feedback connections from one level to an earlier level.

Notice that the input nodes in Figure 1.4 have only a single connection projecting into them. Similarly, the output nodes have only a single connection projecting out from them. Again, this portrayal is a gross simplification in comparison to biological neural networks. Real neural networks are likely to receive inputs from multiple sources and send outputs to multiple destinations. Of course, there will be some biological neurons that receive only a single input. For example, retinal photoreceptors might be thought as neurons with just a single input—in this case, the light source that fires the neuron. More generally, though, it is appropriate to think of the single input to an input neuron in Figure 1.4 as summarizing the input from multiple sources, and the single output from an output neuron as summarizing the output to multiple destinations.

We can now begin to consider just how the neural network performs its task. First, let's assume that each input node has a certain level of activity associated with it. Our goal is to determine how the activity of the input neurons influence the output nodes. To simplify the explanation, we shall consider the process from the point of view of just one output unit, the left-hand output in Figure 1.4. This is highlighted in Figure 1.5. We refer the two input nodes as node$_0$ and

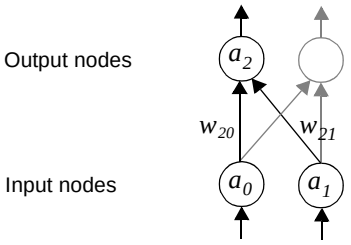


FIGURE 1.5 The activation of the left-hand output unit from Figure 1.4.

node$_1$, and to the two output nodes as node$_2$ and node$_3$. The activation values of the input nodes are denoted a_0 and a_1, respectively. Our goal is to calculate the activation of the left-most output node, a_2.

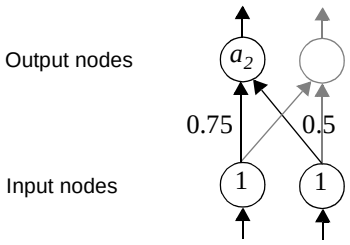
From Figure 1.2 we see that one of the computations that the neuron performs is to calculate its net input from other neurons. The output neuron in Figure 1.5 receives input from two input neurons, namely a_0 and a_1. These two input neurons communicate with the output neurons via independent connections. We also said earlier that

exactly how much input was received along a given connection depended on the activation values of the sending units (in this example, a_0 and a_1), but also the weights on the connections. These weights serve as multipliers. In Figure 1.5 we have denoted the weight from input node$_0$ to output node$_2$ with the symbol w_{20}, using the convention that a weight labeled w_{ij} refers to the connection *to* node$_i$ *from* node$_j$. Note that since activity flows in only one direction along the connections, the value of the weight w_{20} is not the same as w_{02}. In fact, the connection w_{02} does not exist in the network depicted in Figure 1.5.

In the example above, the only inputs to node$_2$ come from the two input nodes. Each input is the product of the activation of the sender unit times the weight; the total input to node$_2$ is simply given by the sum of these two products, i.e., $w_{20}a_0 + w_{21}a_1$.

Exercise 1.1 [a]

To make the example concrete, assume our network has the weights shown, and the input nodes have the activations shown.



1. What will be the *input* which is received by node$_2$?

The net input by itself does not determine the activity of the output node. We also need to know the activation function of the node. Let us assume our nodes have activation functions as given in Equation 1.2 (and shown graphically in Figure 1.3). In the table below we give sample inputs and the activations they produce, assuming a logistic activation function.

Exercise 1.1 [a]

INPUT	ACTIVATION
-2.00	0.119
-1.75	0.148
-1.50	0.182
-1.25	0.223
-1.00	0.269
-0.75	0.321
-0.50	0.378
-0.25	0.438
0.00	0.500
0.25	0.562
0.50	0.622
0.75	0.679
1.00	0.731
1.25	0.777
1.50	0.818
1.75	0.852
2.00	0.881

2. What will be the *activation* of $node_2$, assuming the input you just calculated?

a. Answers to exercises are given at the end of each chapter.

In many networks, it is also useful to allow nodes to have what amounts to a default activation. Note that in the absence of any input (which means an input of 0.0), our nodes will have an output of 0.5 (see Exercise 1.1). Suppose we want a node to be "off"—have an output of 0.0—in the absence of input. Or we might wish its default state to be on.

We can accomplish this by adding one additional node to our network. This node receives no inputs, but is always fully activated and outputs a 1.0. The node can be connected to whatever other nodes in the network we wish; we often connect this node to all nodes except the input nodes. Finally, we allow the weights on the connections from this node to its receiving nodes to be different.

This effectively guarantees that all the receiving nodes will have some input, even if all the other nodes are off. Since the extra node's

output is always 1.0, the input it sends to any other node is just $1.0 \times w_{ij}$—or the value of the weight itself.

Because of what it does, this extra node is called the *bias* node (only one is needed per network). What it does is similar to giving each node a variable threshold. A large negative bias means that the node will be off (have activations close to 0.0) unless it receives sufficient positive input from other sources to compensate. Conversely, if the bias is very positive, then the receiving node will by default be on and will require negative input from other nodes to turn it off. Allowing individual nodes to have different defaults turns out to be very useful.

Learning

So far we have discussed simple networks that have been pre-wired. In Exercise 1.1 we gave as an example a network whose weights were determined by us. For some other problem, we might wish the network to learn what those weights should be.

In this book we will be using a learning algorithm called "backpropagation of error" (Rumelhart, Hinton, & Williams, 1986; see also Le Cun, 1985; Werbos, 1974). Backpropagation is also referred to as the 'generalized delta rule'. (This algorithm is described fully in the paper by Rumelhart et al. (1986) and the reader is urged to consult that paper for a more detailed explanation.)

The basic strategy employed by "backprop" is to begin with a network which has been assigned initial weights drawn at random, usually from a uniform distribution with a mean of 0.0 and some user-defined upper and lower bounds (frequently ± 1.0). The user also has a set of training data, which come in the form of *input/output* pairs. The goal of training is to learn a single set of weights such that any input pattern will produce the correct output pattern. Often it is also desired that those weights will allow the network to generalize to novel data not encountered during training.

The training regime involves several steps. First, an input/output pattern is selected, usually at random. The input pattern is used to activate the network, and activation values for output nodes are calculated. (Note that in the example in Figure 1.4 our network has only

input nodes and output nodes. We could just as easily have additional nodes between these two layers, and in fact there are good reasons to

What is $f'(\text{net}_{ip})$ *?*

$f'(net_{ip})$ (pronounced "f prime of net_{ip}") is the first derivative of the node's activation function. This is just the slope of the activation function. The activation function, the sigmoid, is defined mathematically in **(EQ 1.2)** *and depicted graphically in Figure 1.3. Notice that the slope is steepest around the middle of the function (where the net input is close to zero). In fact, the slope of the sigmoid activation function is given precisely by the expression $o_{ip}(1 - o_{ip})$. This is plotted in Figure 1.6.*

The error term δ_{ip} is just the product of the actual error on the output node and the derivative of the node's activation function. For large values of net input to the node (both positive and negative) the derivative is small. Consequently, δ_{ip} will be small. Net input to a node tends to be large when the connections feeding into the node are strong. Conversely, weak connections tend to yield a small input to a node. With small values of net input, the derivative of the activation function is large (see Figure 1.6) and δ_{ip} can be large—provided the output error is large.

wish to have such "hidden nodes"; see the companion volume *Rethinking Innateness*, Chapter 1 and Chapter 3, this volume.)

Because the weights of the network have been chosen at random, the outputs that are generated at the outset of training will typically not be those that go with the input pattern we have chosen; the outputs are more likely to be garbage than anything else. In the second step of training, we compare the network's output with the desired output (which we call the *teacher* pattern). These two patterns are compared on a node-by-node basis so that for each output node we can calculate its error. This error is simply the difference in value between the target for node$_i$ on training pattern p (we will call this target t_{ip}) and the actual output for that node on that pattern (o_{ip}), multiplied by the derivative of the output node's activation function given its input. We'll call that error δ_{ip} (δ is pronounced "delta"):

$$\delta_{ip} = (t_{ip} - o_{ip}) f'(net_{ip}) = (t_{ip} - o_{ip}) o_{ip}(1 - o_{ip}) \qquad \textbf{(EQ 1.3)}$$

So the problem now is how to apportion credit or blame to each of the connections in the network. We know, for each output node, how far off the target value it is. What we need to do is to adjust the weights on the connections which feed into it in such a way as to reduce that error. That is, we want to change the weight on the connections from every node$_j$ coming into our current node$_i$ in such a way that we will

reduce the error on this pattern. This change in weight is calculated as:

$$\Delta w_{ij} = -\eta \frac{\partial E_p}{\partial w_{ij}} \quad \textbf{(EQ 1.4)}$$

What does $\frac{\partial E_p}{\partial w_{ij}}$ mean?

*This is what is called a **partial derivative.** What it expresses is actually very straightforward and can be understood intuitively without knowing calculus. Basically, this term measures how the quantity on the top changes when the quantity on the bottom is changed. In this particular case, we want to know how the error (E) is affected by changing the weights (w).*

If we knew this, then we would know how to change the weight (the Δ symbol—also pronounced "delta"—on the left of Equation 1.4 means "change") in order to decrease the error, where error will mean the discrepancy between what the network is outputting, compared with what we want it to be outputting.

That is, we want to know how changes in error are related to changes in weights. (The η—pronounced "eta"—is known as the learning rate, and is a small constant. Since our goal is to find a set of weights which will work for *all* input/output patterns, we should be cautious in changing the weights too much on any given pattern.)

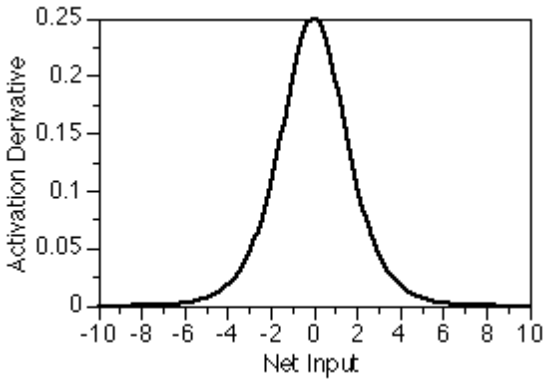


FIGURE 1.6 The derivative of the activation function

Of course, the real question is how we compute the expression on the right, practically speaking. It turns out that this quantity can be calculated as:

$$\Delta w_{ij} = \eta \delta_{ip} o_j \qquad \textbf{(EQ 1.5)}$$

This is often called the "delta rule." We won't explain the math behind the derivation here; if you are interested, consult Rumelhart et al., 1986; or Hertz, Krøgh & Palmer, 1991.

We do three things with this equation. First, we make our changes small, so η is often set to some value less than 1.0 (e.g., 0.1 or 0.3). We do this because, if we are updating weights after every pattern, we don't want to have changes be too drastic. There are other patterns yet to be encountered, and we wish to proceed cautiously so that we can find a set of weights which will work for all the patterns, not just the current one. Second, the change in weight depends on the error we have for this unit, δ_{ip}, as calculated in Equation 1.3. Finally, we also take into account the output we have received from the sending node, o_j. This makes sense because the node's error is related to how much (mis)information it has received from another node; if the other node is highly active and has contributed a great deal to our current activation, then it bears a large share of the responsibility for our error.

We proceed in this manner, calculating errors on all output nodes, and weight changes on the connections coming into them (but we do not yet actually make the changes). Then we move down to the hidden layer(s) (if there are any). We use the same equation, Equation 1.5, for changing weights that lead into the hidden units from below. However, we cannot use Equation 1.3 to compute the hidden nodes' errors, since there is no given target against which they can be compared. Instead, we make the hidden nodes "inherit" the errors of all the nodes they have activated, using the same principle of credit/blame. If the nodes activated by a hidden node have large errors, then the hidden unit shares responsibility. So we calculate its error by simply summing up the errors of the nodes which it activates (multiplied by the weight between the nodes, since obviously if the weight is very small the hidden node has much less responsibility). This procedure is summarized in Equation 1.6 where the subscript *i* indicates the hidden node, *p* indicates the current pattern and *k* indexes the output node feeding error back to the hidden node:

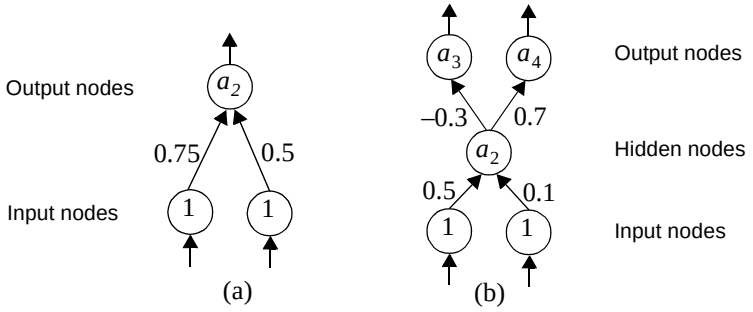
$$\delta_{ip} = f'(net_{ip}) \sum \delta_{kp} w_{ki} \qquad \textbf{(EQ 1.6)}$$

(As in Equation 1.3, the derivative of the hidden unit's activation function is also multiplied in.)

This procedure continues iteratively down through the network, hence the name "backpropagation of error." When we get to the layer above the input layer (inputs have no incoming weights), we take the third and final step of actually imposing the weight changes.

Exercise 1.2

1. Why do we wait until we have calculated all the δs before making the weight changes, rather than change weights as we go down the network?



2. Imagine we are training the single-layered network shown above on the left. The network is shown with a set of activation values for the input nodes and weights connecting the input nodes to the output node. Assume that the output node has a sigmoid activation function (see Exercise 1.1), that the desired output is 1.0 and that the learning rate $\eta = 0.1$. Calculate the changes that will be made to the two weights in the network. *Hint: You will also need to know the value of the derivative in Equation 1.3. These are tabulated for different activation values below. Don't forget to calculate net inputs to determine the activation value from the table in Exercise 1.1.*

Exercise 1.2

ACTIVATION	DERIVATIVE
0.0	0.00
0.1	0.09
0.2	0.16
0.3	0.21
0.4	0.24
0.5	0.25
0.6	0.24
0.7	0.21
0.8	0.16
0.9	0.09
1.0	0.00

3. Repeat the calculations for the multi-layered network with hidden nodes using the same learning rate parameter and target activations of 1.0 on both output nodes. *Hint: Calculate δs for the output nodes in just the same way as you did in Exercise 1.2.1. However, you will also need to consult Equation 1.6 to determine δs for the hidden node.*

4. What training conditions promote maximum weight changes in a network? *Hint: Consult Equation 1.3 and Equation 1.5 before attempting to answer this question.*

Answers to exercises

Exercise 1.1

1. If the activation values of the two input nodes were 1.0 and 1.0, respectively and the connection weights were 0.75 and 0.5 then the net input to node$_2$ would simply be the sum of the products of each weight by the input coming in along that weight:

   $$\text{netinput}_2 = (1.0 \times 0.75) + (1.0 \times 0.5) = 1.25.$$

2. If node$_2$ receives this input, we can read off its corresponding activation by consulting the activation table in Exercise 1.1. We see that an input of 1.25 leads to an activation of 0.777. Note that the activation function scales inputs into the range from 0.0 to 1.0. When the net input is 0.0, the node's output is exactly in the midrange of its possible activation range: 0.5. Positive inputs result in activations that are greater than 0.5; negative inputs result in activations that are less than 0.5.

Exercise 1.2

1. After we compute the weight changes for a layer, we hold off actually changing weights until at least after we have calculated the errors on the nodes *below* us. This is because the errors for those nodes are calculated using Equation 1.6. In that equation, the nodes below a layer inherit some part of the error of the layer above; how much error is based on the error itself, multiplied by the weights from the lower to the upper layers. So we do not want to change those weights until after we have apportioned blame, using the weights that were in effect at the time the output pattern was generated.

2. As we saw in Exercise 1.1 the activation of the output node is 0.77. This means that the discrepancy between the actual output and the desired output $t - o = 1.0 - 0.77 = 0.23$. The calculation of δ from Equation 1.3 requires we determine the derivative $f'(\text{net})$ for an activation of 0.77. From Exercise 1.2 we see that an activation value of 0.77 yields a derivative value of approximately 0.16. Substituting these values in Equation 1.3 we get:

$$\delta_2 = (t - o)f'(\text{net}_2) = 0.23 \times 0.16 = 0.037$$

We are now in a position to calculate the individual weight changes from Equation 1.5:

$$\Delta w_{20} = \eta \delta_2 o_0 = 0.1 \times 0.037 \times 1.0 = 0.0037$$

Recall that the constant η defines the learning rate and o_o stands for the activation of the input node 0. Similarly, the weight change for the connections from input node 1 to the output node is given by:

$$\Delta w_{21} = \eta \delta_2 o_1 = 0.1 \times 0.037 \times 1.0 = 0.0037$$

In this example, the changes to each of the weights are identical since the input nodes have the same activation. Note that the overall effect of this training trial is to change the connections only slightly. The process would have to be repeated many times before the error is reduced to zero. Keeping the weight changes small helps prevent the network making adjustments that might not be suitable for other input/output patterns.

3. First calculate the activations of all the receiving nodes:

$$a_2 = \text{logistic}((1.0 \times 0.5) + (1.0 \times 0.1)) \approx 0.65$$
$$a_3 = \text{logistic}(0.65 \times -0.3) \approx 0.45$$
$$a_4 = \text{logistic}(0.65 \times 0.7) \approx 0.6$$

Second calculate $\delta_{3,4}$ for the output nodes:

$$\delta_3 = (t_3 - o_3)f'(\text{net}_3) \approx (1.0 - 0.45) \times 0.24 = 0.13$$
$$\delta_4 = (t_4 - o_4)f'(\text{net}_4) \approx (1.0 - 0.6) \times 0.24 = 0.1$$

Now calculate δ_2 for the hidden unit using Equation 1.6:

$$\begin{aligned}\delta_2 &= f'(\text{net}_2) \sum_{k \in 3,4} \delta_k w_{k2} \\ &= 0.23((0.13 \times -0.3) + (0.1 \times 0.7)) \\ &= 0.007\end{aligned}$$

We can now determine all the weight changes using Equation 1.5:

$$\Delta w_{20} = \eta \delta_2 o_0 = 0.1 \times 0.007 \times 1.0 = 0.0007$$
$$\Delta w_{21} = \eta \delta_2 o_1 = 0.1 \times 0.007 \times 1.0 = 0.0007$$
$$\Delta w_{32} = \eta \delta_3 o_2 = 0.1 \times 0.13 \times 0.65 = 0.008$$
$$\Delta w_{42} = \eta \delta_4 o_2 = 0.1 \times 0.1 \times 0.65 = 0.007$$

Note how all the weight changes are made *after* the δs for output and hidden nodes have been computed. The learning rate $\eta = 0.1$ scales the size of the weight changes and thereby helps to ensure the network is not too influenced by errors for individual patterns.

4. There are several factors that determine how much the weights are changed on any learning trial. These are summarized in Equation 1.5. The learning rate influences the proportion of the error term δ that is used to change the weights. So large learning rates will tend to lead to large weight changes. The error term δ itself is defined as the product of the actual output error and the derivative of the sigmoid activation functions (see Equation 1.3). Large errors will also tend to lead to large weight changes. However, the effect of the error is modified by the derivative of the sigmoid. We saw in Figure 1.6 that the derivative is largest when the input to the node is small. Small inputs tend to go together with small weights. Networks tend to have small weights at the beginning of training. Hence, networks are most sensitive to learning early in their lives. As the network gets older, its weights get bigger leading to large net input values. Large net inputs give small derivatives, leading to small weight changes. Older networks find it more difficult to learn!