

Learning to use the simulator

Defining the task

In this chapter, you will investigate how a neural network can be trained to map the Boolean functions AND, OR and EXCLUSIVE OR (XOR). Boolean functions just take some set of inputs, usually 1s and 0s, and decide whether a given input falls into a positive or a negative category. We can think of these Boolean functions as equivalent to the input and output activation values of the nodes in a network with 2 input units and 1 output unit. Table 3.1 summarizes the mapping con-

TABLE 3.1 The Boolean functions AND, OR and XOR

Input Activations		Output Activation		
Node 0	Node 1	Node 3		
		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

tingencies for each of these 3 functions. The first two columns of Table 3.1 specify the input activations. There are 4 possible input patterns made up from the 2^2 possible binary combinations of 0 and 1. Columns 3–5 specify the single output activation for the desired

Boolean function we require our network to compute. These input patterns and output activations define the training environment for the network we will build.

Exercise 3.1

-
- Do you notice anything peculiar about any of these functions? Which one would you say was the odd one out?
-

It may seem odd that we are asking you to perform your first simulations with abstract Boolean functions that would appear to have little bearing on our understanding of the development of psychological processes. However, we have several reasons for starting in this fashion. First, the networks that you will use to learn these functions are quite simple and relatively easy to construct. They provide a (hopefully) painless introduction to conducting your own simulations. Second, the type of networks that you will need to learn AND, OR and XOR differ in interesting and instructive ways. Many of the problems that you will encounter for these Boolean functions will illustrate some fundamental computational properties of networks that will have direct implications for your understanding of the application of networks to more complex problems. Indeed, we still often go back to these simple Boolean functions to help us work through issues which seem unfathomable in more complicated networks.

Defining the architecture

Before you can configure the simulator, you need to decide what kind of network architecture to use for the problem at hand. Let us start with the Boolean function AND. There are 4 input patterns as specified in Table 3.1 and 2 distinct outputs. Each input pattern specifies 2 activation values and each output a single activation. For every input pattern, there exists a well-defined output. So for this problem, you should use a simple feedforward network with 2 input units and a single output unit. An example of the type of network architecture you require is shown in Figure 3.1. These networks are called “single-lay-

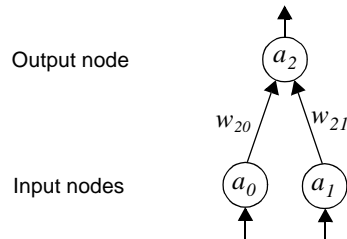


FIGURE 3.1 A single-layered perceptron for learning the Boolean function AND.

ered perceptrons” because they have single layer of weights between input and output nodes. In this book, these networks are trained with the “delta rule” described in Equation 1.5 on page 13.

We have now completed the conceptual analysis that is required to set up the simulator. You are now ready to move on to learning about some of the technical issues involved.

Setting up the simulator

The **tlearn** neural network simulator has been programmed to run on a variety of computing platforms including Macintoshes, Windows and most Unix machines that run X-windows. Furthermore, we have designed the user interface to look more or less identical across these different platforms. So the details involved in setting up your simula-

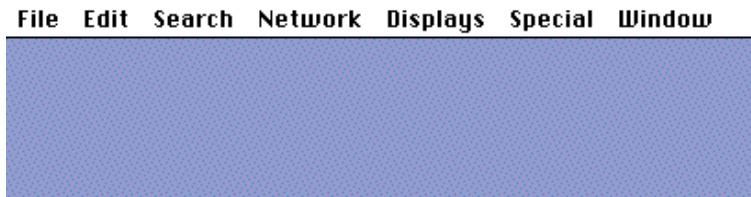


FIGURE 3.2 Startup menu for **tlearn**.

tions will be more or less the same irrespective of the type of computer system you are using. Where differences do occur, and we promise

that they will be only minor differences, we will point them out. The description we give here is for the Macintosh version of the application. For information on how to install **tlearn** on your computer system, please refer to the instructions in Appendix A.



You start up **tlearn** on your computer in the same way as any other application, such as double-clicking on the **tlearn** icon or through a sequence of keyboard commands. We will assume that you have done this. When **tlearn** has started up, your computer monitor should display a set of menus like those depicted in Figure 3.2. These menus are accessed in the standard fashion for your computer system. For example, you may click on one of the menu items and hold down the mouse button to display the options associated with that menu.

You begin the process of constructing a new network from the Network menu. Select the Network menu and drag the mouse so that the New Project option is highlighted as shown in Figure 3.3. When

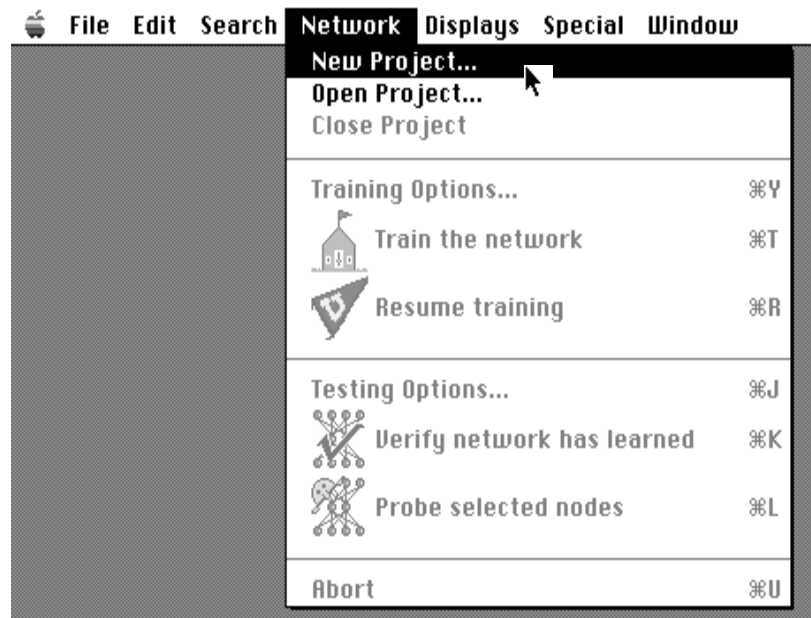


FIGURE 3.3 Select the New Project option from the Network menu.

you release the mouse button the New Project dialogue box is dis-

played as shown in Figure 3.4. In this dialogue box you can select a

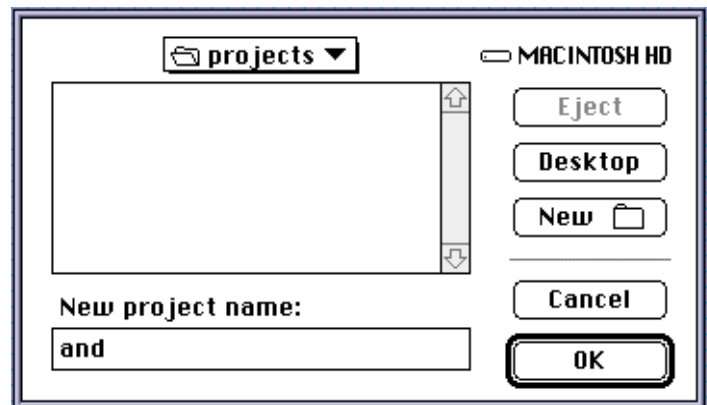


FIGURE 3.4 The New Project dialogue box. Select a folder or directory in which to save your current project and name the project. In this case, name the project **and**.

directory or folder in which to save your project files and you can name your project. In this case, call the project **and** since you are building a network to learn Boolean AND. Then click on to initiate the relevant files in your selected directory or folder. The display on your monitor should now resemble that depicted in Figure 3.5. Notice that **tlearn** has created 3 different windows—**and.cf**, **and.data** and **and.teach**. Each window will be used for entering information relevant to a different aspect of the network architecture or training environment:

- The **and.cf** file is used to define the number of nodes in a network and the initial pattern of connectivity between the nodes before training begins.
- The **and.data** file defines the input patterns to the network, how many there are and the format by which they are represented in the file.
- The **and.teach** file defines the output patterns to the network, how many there are and the format by which they are represented in the file.

By convention, **tlearn** requires that any simulation project possesses these 3 files and expects them to possess the file extensions **.cf**, **.data** and **.teach**. You can choose any name you like for the

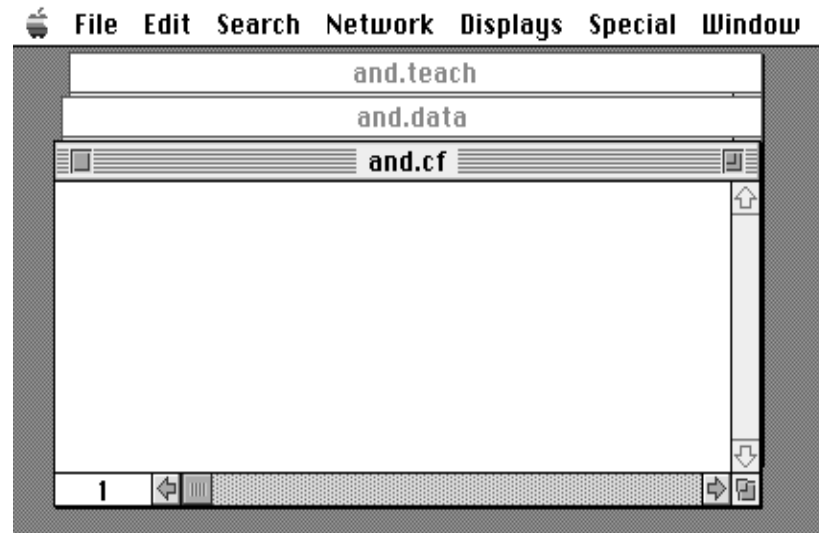
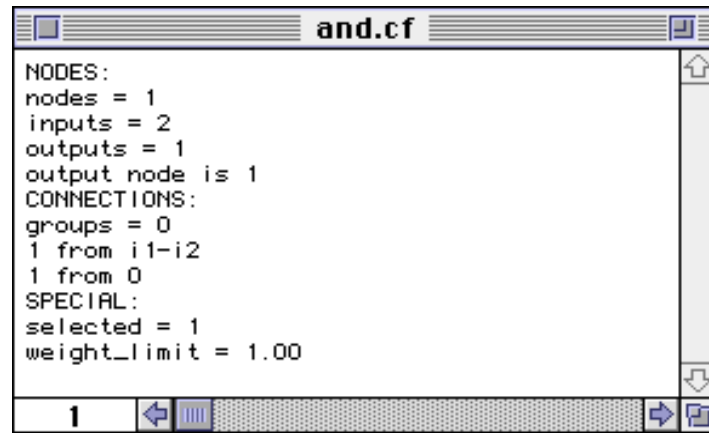


FIGURE 3.5 Startup files for a new **tlearn** project include `.cf`, `.data` and `.teach` files. The `.data` and `.teach` files define the network's training environment. The `.cf` file defines the network architecture.



filename. For the current project, we have chosen the filename **and**. However, all the files that belong to a project should have the same filename. Project information is stored in a special file so that you can activate previously created projects simply by activating (e.g., double-clicking) that file. The filename identifies the project file. In this case, **tlearn** has created a project file called **and** as well as the 3 required files.

Now let us consider in detail what information must be stored in the different files. It is your responsibility to enter this information. For now, we will assume that you will use the simple file creation and editing facilities that **tlearn** provides. However, you can create these files in the text editor or word processor of your choice, just so long as you remember to save the files in ASCII format.¹ The information required in the **and.cf** file is displayed in Figure 3.6. You should enter this information exactly as displayed, honoring upper case and lower case distinctions, spaces and colons. If you make a mistake while entering the characters, you can use the arrow keys on



```

and.cf
NODES:
nodes = 1
inputs = 2
outputs = 1
output node is 1
CONNECTIONS:
groups = 0
1 from i1-i2
1 from 0
SPECIAL:
selected = 1
weight_limit = 1.00

```

FIGURE 3.6 The `and.cf` file contains 3 sections: The `NODES :` section specifies the total number of units in the network and identifies which nodes play the role of input and output. The `CONNECTIONS :` section specifies how the units are connected to each other. The `SPECIAL :` section provides information that determines the initial value of the connection strengths and specifies the units whose activation values are available for inspection.

your keyboard or the mouse to navigate in the file. Use the “delete” or “backspace” key to edit out any unwanted material. When you have finished creating the file and you are sure there are no errors then save the file to disk using the standard technique on your computer system. Alternatively, you can use the **Save** command in the **File** menu of **tlearn** as shown in Figure 3.7.

1. Most word processors introduce formatting information that involve control characters like `^S` which **tlearn** doesn't understand. It is imperative that you save files created by a word processor in ASCII format (also often referred to as “Text” format in many word processors) otherwise **tlearn** may fail to run. Also notice that if you create your `.cf`, `.data` and `.teach` files in another text editor, you will still have to define the project in the **New Project** dialogue box in the **Network** menu. Remember to use the same name for the project as you use for the filename in your `.cf`, `.data` and `.teach` files so that **tlearn** knows which files to look for.

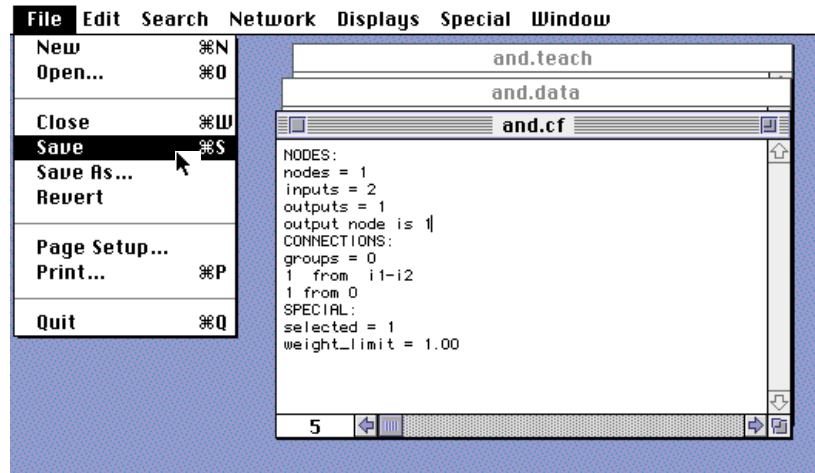


FIGURE 3.7 Saving the `and.cf` file from the `tlearn` File menu.

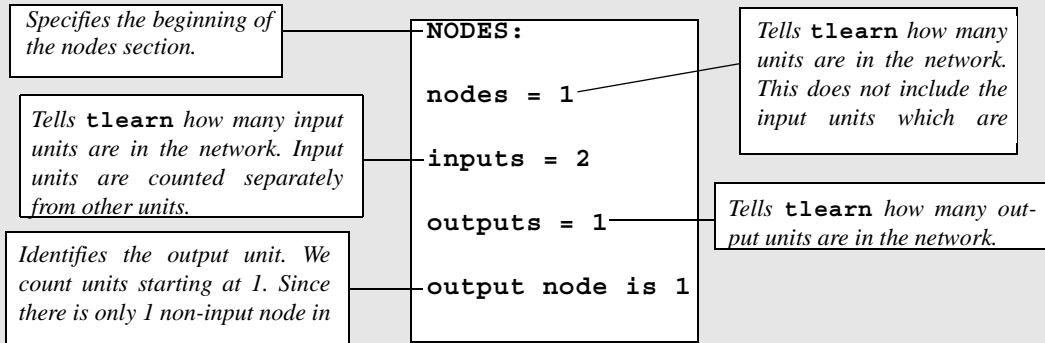
The Configuration (`.cf`) file

The `.cf` file is the key to setting up the simulator. This file describes the configuration of the network. It must conform to a fairly rigid format, but in return offers considerable flexibility in architecture. We shall now dissect the innards of the `.cf` file in some detail.

There are three sections to this file (see Figure 3.6). Each section begins with a keyword in upper case, flush-left. The three section keywords are **NODES:**, **CONNECTIONS:** and **SPECIAL:**. Note the colon. Sections must be described in the above order.

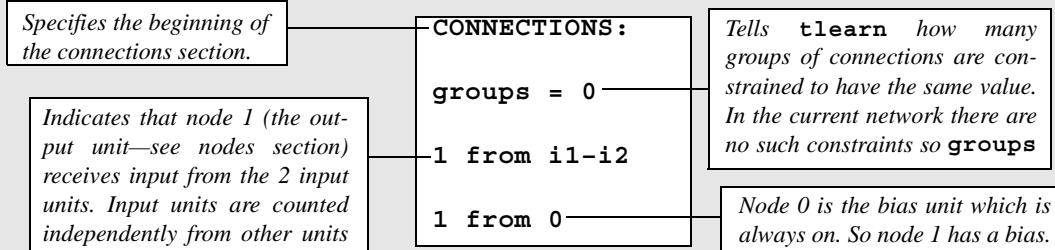
NODES: This is the first line in the file. The second line specifies the total number of nodes in the network as “**nodes = #**”. *Inputs do not count as nodes.* The total number of inputs is specified in the third line as “**inputs = #**”. The fourth line specifies the number of nodes which are designated as outputs according to “**outputs = #**”. Lastly, the output nodes are listed specifically by number (counting the first node in the network as 1). The form of the specification is “**output nodes are <node-list>**”. (If only a single output is present one can write “**output node is #**” as we have done in the `and.cf` file). *Spaces are critical.*

Dissecting the NODES : section of the and.cf file



`CONNECTIONS :` This is the first line of the next section. The line fol-

Dissecting the CONNECTIONS : section of the and.cf file



lowing this must specify the number of groups, as in “`groups = #`”. All connections in a group are constrained to be of identical strength—though in many cases “`groups = 0`” as in the `and.cf` file. Following this, information about connections is given in the form:

```
<node-list> from <node-list>
```

A **<node-list>** is a comma-separated list of node numbers, with dashes indicating that intermediate node numbers are included. A **<node-list>** contains *no spaces*. Nodes are numbered counting from 1. Inputs are likewise numbered counting from 1, but are designated as **i1**, **i2**, etc. **Node 0** always outputs a 1 and serves as the bias node. If biases are desired, connections *must* be specified from **node 0** to specific other nodes (not all nodes need be biased).

SPECIAL: This is the first line of the third and final section. Optional lines can be used to specify whether some nodes are to be linear ("**linear = <node-list>**"), which nodes are to be bipolar (meaning their values range from -1 to +1 rather than 0 to +1 which is the default "**bipolar = <node-list>**"), which nodes are selected for special printout ("**selected = <node-list>**"), and the initial weight limit on the random initialization of weights ("**weight_limit = #**"). Again, *spaces are critical*.

This may seem an overly complicated procedure for defining a network with just 2 input units and 1 output unit. You are right, it is! However, you will see that this format enables you to define far more complicated network architectures as well. So it is worth the effort to invest time to make sure you understand how the **.cf** file works.

Dissecting the SPECIAL: section of the and.cf file

Specifies the beginning of the special section.

Tells **tlearn** how to initialize the weights in the network. This line will cause **tlearn** to set the start weights (from the input to the output and the bias to the output) randomly in the range

```
SPECIAL:
selected = 1
weight_limit = 1.00
```

Tells **tlearn** which units are to be selected for special printout. In this case, we select node

The Data (.data) file

The **and.data** file defines the input patterns which are presented to **tlearn**. Enter the data as shown in Figure 3.8. Don't forget to save the file when you have finished.

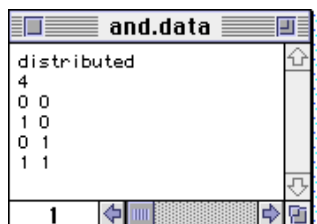


FIGURE 3.8 The **and.data** file.

The first line must either be **distributed** (the normal case) or **localist** (when only a few numbers of many input lines are non-zero). The next line is an integer specifying the number of input vectors to follow. The remainder of the **.data** file consists of the input. These may be input as integers or floating-point numbers.

In the (normal) **distributed** case, the input is a set of vectors. Each vector contains i floating point numbers, where i is the number of inputs to the network. Spaces between the numbers are critical. In the **localist** case, the input is a set of **<node-list>**'s listing only the numbers of those nodes whose values are to be set to one. Node lists follow the conventions described in the **.cf** file.

The Teach (.teach) file

The **.teach** file is required whenever learning is to be performed. Enter the data for the **and.teach** file as shown in Figure 3.9. Don't forget to save the file when you are finished. As with the **.data** file, the first line must be either **distributed** (the normal case) or **localist** (when only a few of many target values are nonzero). The next line is an integer specifying the number of output vectors to follow. The ordering of the output patterns matches the ordering of the corresponding input patterns in the **.data** file. In the (normal) **distributed** case, each output vector contains o floating point or integer numbers, where o is the number of outputs in the network. An

Dissecting the `and.data` file

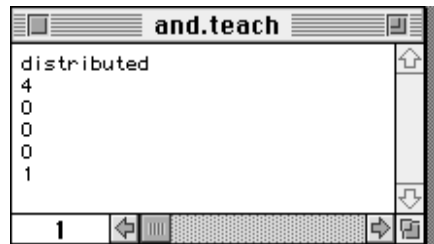
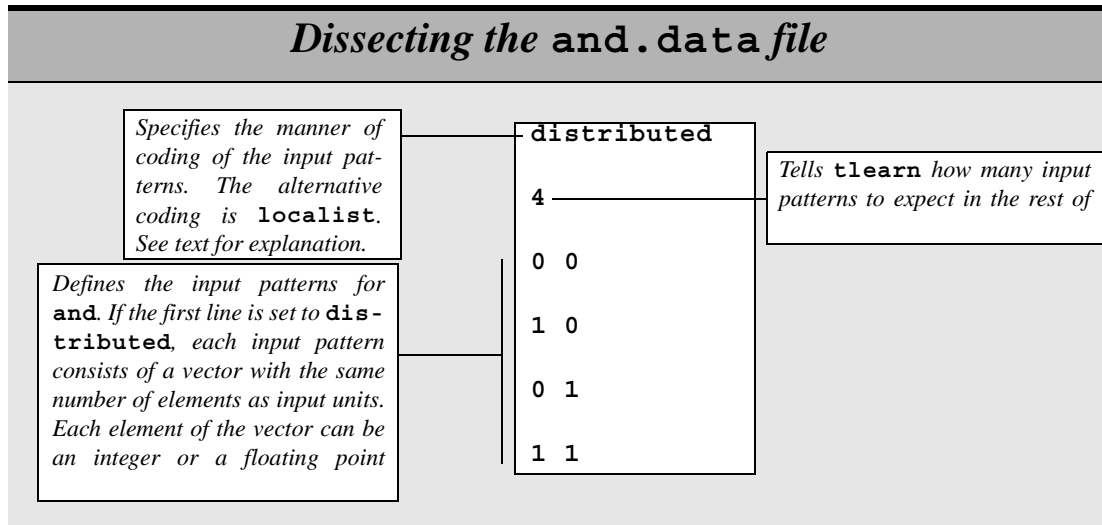
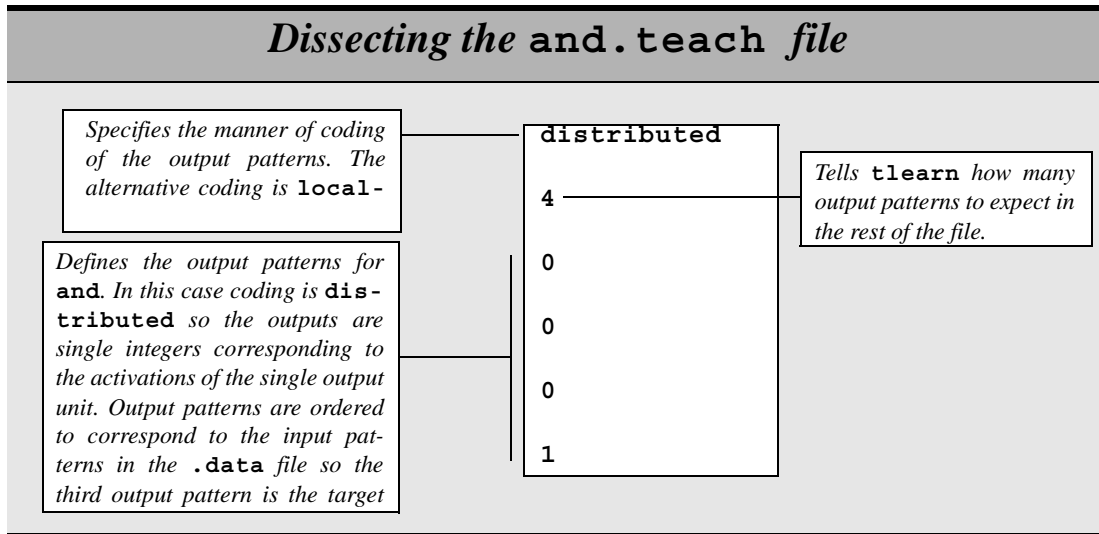


FIGURE 3.9 The `and.teach` file.

asterisk (*) may be used in place of a floating point number to indicate a “don't care” output. In the `localist` case, the input is a set of `<node-list>`'s listing only the numbers of those nodes whose values are to be set to one. Node lists follow the conventions described in the `.cf` file.

Checking the architecture

If you have typed in the information to the `and.cf`, `and.data` and `and.teach` files correctly then you should experience no problems running the simulation. However, `tlearn` offers a useful way to check whether the `and.cf` file has been correctly specified. You can



display a picture of your network architecture using the **Network Architecture** option in the **Displays** menu. The architecture for Boolean AND is shown in Figure 3.10. The buttons at the top of the display enable you to adjust your view of the network. For example, it is possible to view the network without the bias displayed simply by clicking on the **Bias** button. Note, however, that the adjustments you make to the display do not effect the contents of the network configuration file. `tlearn` will not complain if there are any mistakes in the training files (`.data` and `.teach`) when you display the network architecture. An error message will be displayed when you attempt to train the network, if there is a mistake in the syntax of your training files. Needless to say, `tlearn` cannot identify incorrect entries in the training set data. If you accidentally enter the input pattern 0 1 instead of 1 1, you will receive no notification. Check your network files carefully!

Running the simulation

Once you have specified the three input files and saved them to disc, you are almost ready to run your first simulation. First, however, you must specify a number of parameters for `tlearn` that will determine the initial start state of the network as well as the learning rate and

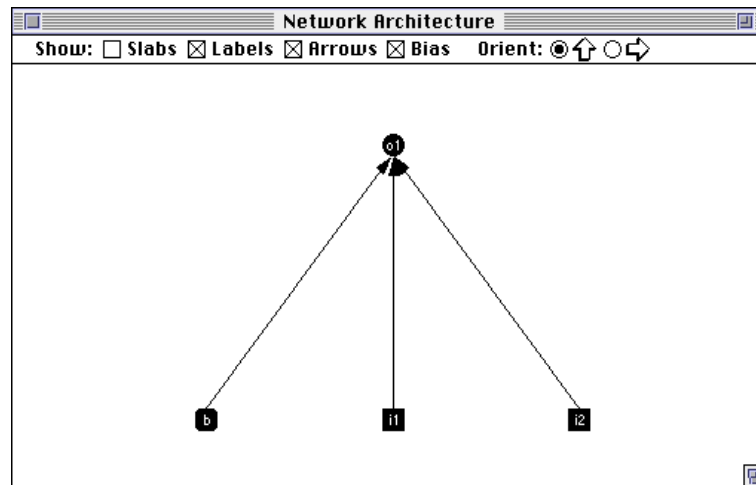


FIGURE 3.10 The Network Architecture display. **tlearn** reads the information in the **and.cf** file to construct this display. The user can customize the network display using the buttons at the top of the display. These changes do not effect the contents of the network configuration file.

momentum. For a brief overview of the kind of parameters that you can set in **tlearn** consult Figure 3.11. To activate this dialogue box you need to select the Training options... option in the Network menu. First, we indicate the number of training sweeps that the network should perform before halting. A training sweep consists of a single presentation of an input pattern causing activation to propagate through the network and the appropriate weight adjustments to be carried out. In Figure 3.11 the number of training sweeps has been set to 1000 which means that 1000 patterns are to be presented to the network. The order in which patterns are presented to the network is determined by the Train sequentially and Train randomly buttons. Activate the Train sequentially button (by clicking it with the mouse) to present patterns in the order in which they appear in the **.data** and **.teach** files. Activate the Train randomly button to present patterns in random order. In Figure 3.11 **tlearn** has been set to select patterns at random.

The initial state of the network is determined by the weight values assigned to the connections before training begins. Recall that the **.cf** file specifies a **weight_limit** in the **SPECIAL:** section (see

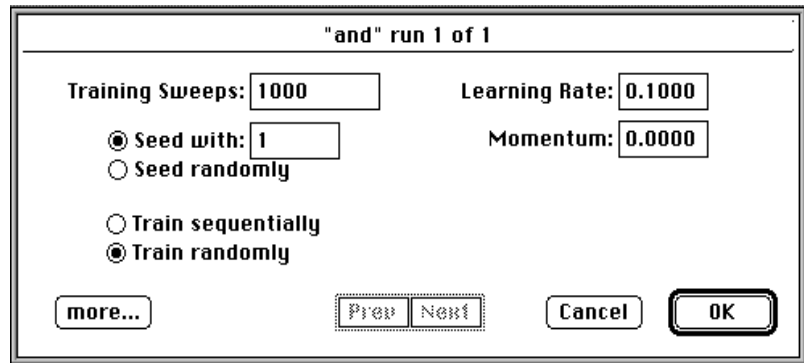


FIGURE 3.11 The Training Options dialog box activated from the Network menu. Here you specify many of the parameters for training the network including the number of sweeps, learning rate, momentum, the initial startup configuration of the weights and the manner in which patterns are selected for training.

Figure 3.6). Weights can be assigned according to a random seed indicated by the number next to the **Seed with:** button. You can select any number you like. The advantage of this approach is that a simulation can be replicated using the same random seed—meaning that the initial start weights of the network will be identical and patterns will be sampled in the same random order. Make sure that you activate the **Seed with:** button if you wish to use this method. Alternatively, you may wish to let the computer choose a random seed for you in which case activate the **Seed randomly** button. Note that *both* of these random seed procedures select a set of random start weights within the limits specified by the `weight_limit` parameter in the `.cf` file. The only difference between the procedures is that in one case you have control over the random seed and in the other case you relinquish control to the computer. In Figure 3.11 we have specified a random seed of 1. Make sure you choose the same random seed for purposes of this demonstration.

The other parameters specified in Figure 3.11 include **Learning Rate:** and **Momentum:** You have already met the learning rate parameter in Chapter 1. It determines how fast the weights are changed in response to a given error signal. The momentum parameter has not yet been properly introduced. For the moment, set these parameters to the values specified in Figure 3.11, i.e., 0.1 and 0.0. There are many

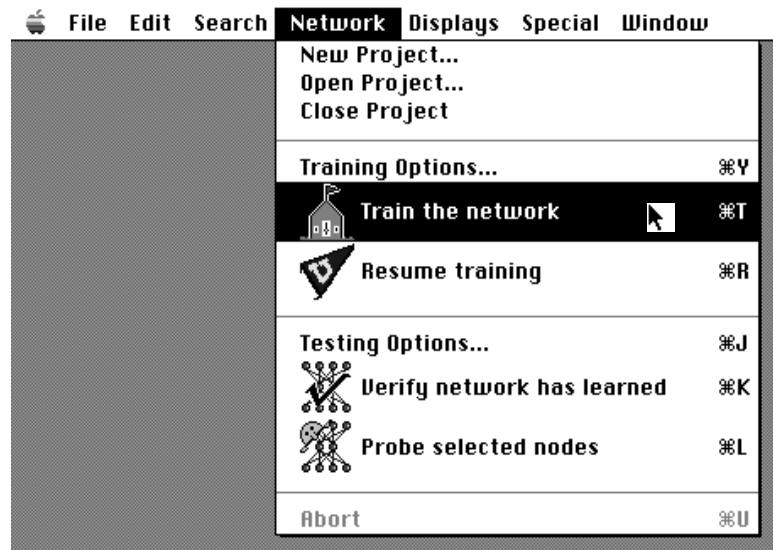


FIGURE 3.12 Use **Train the network** to begin a simulation. Alternatively, use the keyboard shortcut **⌘-T**

other parameters that can be set through the **Training options** dialogue box. For the time being, however, just click to accept the current configuration and close the dialogue box.

Now you can start to train the network. You can do this from the **Network** menu by choosing the option **Train the network** as shown in Figure 3.13. Immediately, you start training the network the **tlearn Status** display appears—see Figure 3.13 (a). The status display indicates how many sweeps have been completed and provides the opportunity to abort training, dump the current state of the network in a weights file and iconify the status display to clear the screen for other tasks while **tlearn** runs in the background. The status display also indicates when **tlearn** has completed the current round of training—see Figure 3.13 (b).

Now that you've trained the network for 1000 sweeps, let's see what it's learned. There are a variety of ways to determine whether the network has solved the problem you have set it. We will examine two methods here:

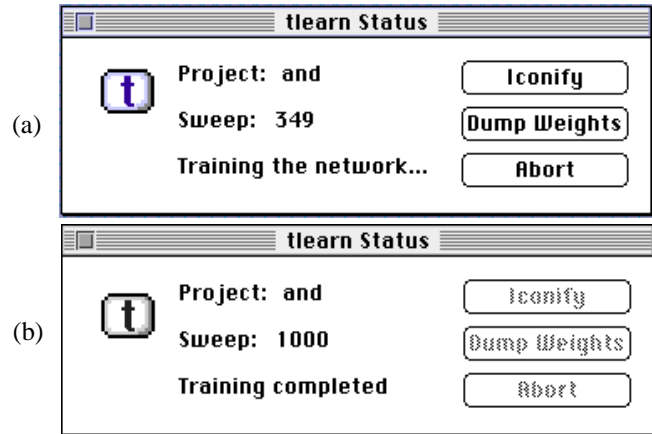


FIGURE 3.13 The **tlearn Status** display window indicates when the simulator is occupied training the network and when it has completed its training cycle. During training it is possible to **Iconify** the status window and revert to another task (while training continues in the background). Training can be terminated by clicking on the **Abort** button. Weights files can be saved by **Dumping Weights**.

1. Examine the global error produced at the output nodes averaged across patterns.
2. Examine the response of the network to individual input patterns.

Global error

For each input pattern, the network will produce an output response. As we saw in Chapter 1 the error at the output can be calculated simply by subtracting the actual response from the desired or target response. The value of this discrepancy can be either positive or negative—positive if the target is greater than the actual output and negative if the actual output is greater than the target output. We wish to determine the global performance of the network across all four of the input/output pairs that make up Boolean AND.

Let us define global error as the average error across the four pairs at a given point in training. In fact, **tlearn** provides a slightly more complicated measure of global error called the RMS or Root Mean Square error. To determine the RMS error **tlearn** takes the square of

the error for each pattern, calculates the average of the squared errors for all the patterns and then returns the square root of this average. Using the RMS error instead of a raw average prevents **tlearn** from cancelling out positive and negative errors. The calculation can be expressed more succinctly in mathematical notation as:

$$\text{rms error} = \sqrt{\frac{\sum_k (\vec{t}_k - \vec{o}_k)^2}{k}} \quad (\text{EQ 3.1})$$

In Equation 3.1 the symbol k indicates the number of input patterns and \vec{o}_k is the vector of output activations produced by input pattern k . The number of elements in the vector corresponds to the number of output nodes. Of course, in the current problem—Boolean AND—there is only one output node so the vector \vec{o}_k contains only one element. The vector \vec{t}_k specifies the desired or target activations for input pattern k .

Exercise 3.2

-
- What value should k take for Boolean AND in Equation 3.1?
-

tlearn keeps track of the RMS error throughout training. The easiest way to observe how RMS error changes is through the Error Display. Activate the Error Display from the Displays menu. An Error Display for the simulation we've just run is shown in Figure 3.14. Error is reported on the graph every 100 sweeps. The x-axis on the graph indicates the number of sweeps and the y-axis the RMS error. If you prefer error to be displayed by a continuous line then click on the Line button.

Notice that error decreases as training proceeds such that after 1000 sweeps RMS error ≈ 0.35 . What does this error level mean? It indicates that the average output error is just 0.35. So the output is off target by approximately 0.35 averaged across the 4 patterns. It would appear that the network has not solved the problem yet. In fact, this level of error may reveal that the network has solved the AND problem. It all depends on how we define an acceptable level of error. Unfortunately, it is not always easy to evaluate network performance on the basis of global error alone. Although the network may have a

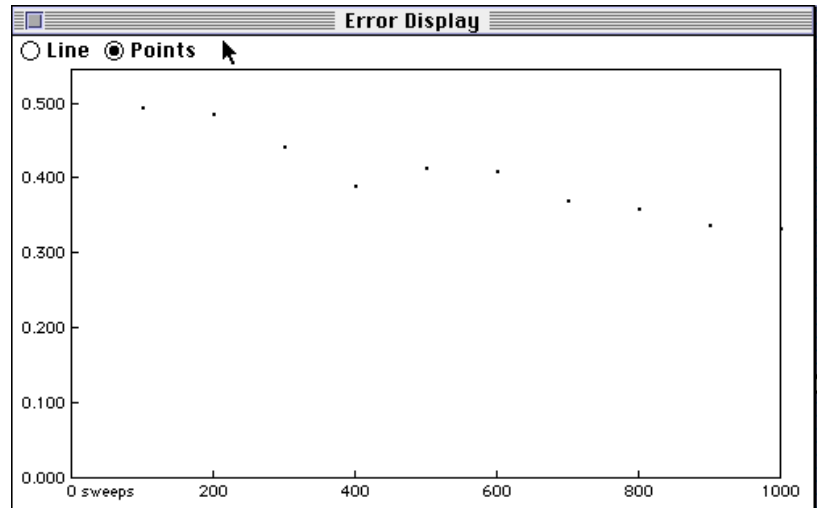


FIGURE 3.14 The error display for global error. The x-axis represents the number of sweeps and the y-axis the RMS error. The error is reported every 100 sweeps and is indicated by a dot. The dots can be joined to form a line by clicking on the Line button.

low RMS error, there is no guarantee that the network has categorized all the input patterns correctly (see Exercise 3.3).

Exercise 3.3

1. How many times has the network seen each input pattern after 1000 sweeps through the training set?
2. How small must the RMS error be before we can say the network has solved the problem?

Pattern error

Recall that RMS error reflects the average error across the 4 input patterns. It is difficult to know whether the error is uniformly distributed across different patterns or whether some patterns have been learned correctly while others remain incorrectly learned. In order to distinguish between these possibilities, we need to examine the output acti-

variations for individual patterns. **tlearn** provides several facilities for viewing the activation values of individual nodes. We will consider two of them.

The most accurate method for determining pattern output is to present each input pattern to the network, just once, and observe the resultant output node activations. The output activations can then be compared with the teacher signal in the **.teach** file. Present each input pattern to the network by selecting the Verify network has learned from the **Network** menu. The **tlearn** status window will indicate that it has conducted 4 sweeps (one for each training input) and that **Verification is Completed**. At the same time a new window called **Output** will open as shown in Figure 3.15. **Output** indicates that it has used

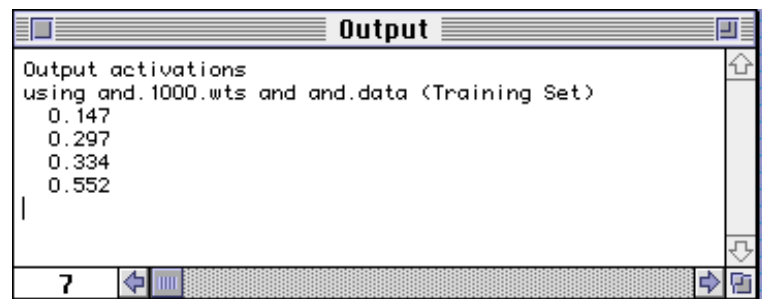


FIGURE 3.15 The **Output** activation window. The activation for each output node is displayed one pattern per row. There is only one output node so only one activation value is displayed. Activations are based on the most recent state of the network as recorded in **and.1000.wts**.

the file **and.1000.wts** as a specification of the state of the network (we will examine the **.wts** file in more detail shortly) and that it has used the **and.data** training patterns to verify network performance. **Output** then displays output activations one pattern per row. Since there is only one output node there will only be a single activation on each row. You can compare the activation values in Figure 3.15 to the target activations specified in your **and.teach** file.

A shortcut method exists for observing output activations for individual input patterns. This facility is accessed through the **Node activa-**

Exercise 3.4

-
1. Has **tlearn** solved Boolean AND?
 2. Calculate the exact value of the RMS error and compare it to the value plotted in Figure 3.14.
-

tions option in the Displays menu. Activating this option will yield the display in Figure 3.16. The Node Activations Display shows the

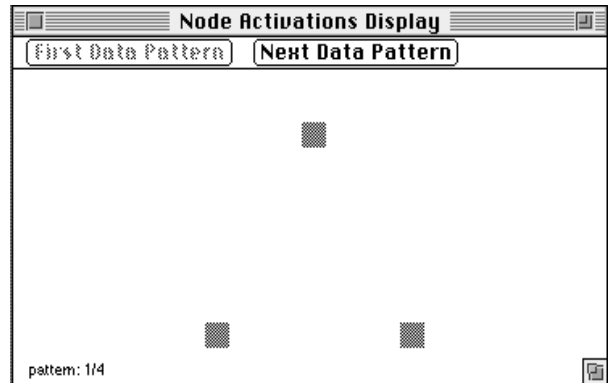


FIGURE 3.16 The node activation display. Click on the **First Data Pattern** button to view node activations when the first input pattern is presented. The size of the white square indicates level of activation. Grey squares indicate zero activation. The current pattern (input pattern 1 of 4) has 0 0 as its input and 0 (almost) as its output. This is a correct response. Scroll through the other patterns by clicking on **Next Data Pattern**.

activations of the two input nodes and the output node. Activation levels are indicated by white bordered squares. Large white squares indicate high activations. Small white squares indicate low activations. A grey square indicates an inactive node. In Figure 3.16 the two input nodes are inactive—corresponding to the input pattern 0 0—and the output node is only slightly active, i.e., more off than on. You can display node activations for the other three input patterns simply by clicking on the **Next Data Pattern** button. **tlearn** will then display activation values in the order they are listed in the training files.

Examining the weights

Input activations are transmitted to other nodes along modifiable connections. The performance of the network is determined by the strength of the connections—also called their weight values. To understand how the network accomplishes its task, it is important to examine the weights in the network. **tlearn** offers several options for displaying the weights in the network. The shortcut method utilizes the Connection Weights option in the Display menu. The connections weights display (shown in Figure 3.17) depicts the weight

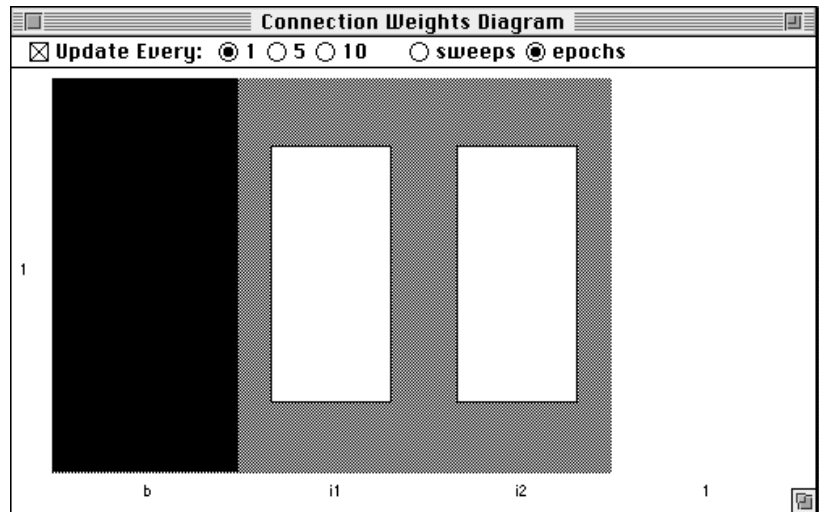


FIGURE 3.17 The Connection Weights display uses a Hinton diagram to plot the *relative* strength of each of the weights (including bias weights) in the network. Black rectangles signify negative weights and white rectangles signify positive weights. Weights are identified by their row and column in the Hinton diagram matrix (see text for further explanation).

values as white (positive) or black (negative) rectangles. The size of the rectangle reflects the absolute size of the connection. In the literature, these displays are usually called “Hinton diagrams.” The array, or matrix, of rectangles are organized in rows and columns. You read Hinton diagrams in accordance with this row/column arrangement. All

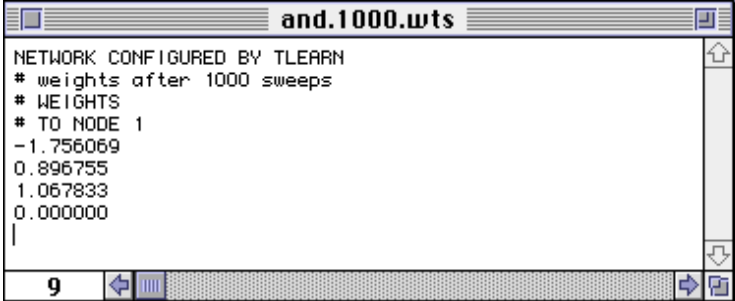
the rectangles in the first column code the values of the connections emanating from the bias node. The rectangles in the second column code the connections emanating from the first input unit. As we read across the columns we observe the connections emanating from higher numbered nodes (as defined by the `.cf` file). The rows in each column identify the destination node of the connection. Again, higher numbered rows indicate higher numbered destination nodes. In the current example, there is only one node that receives inputs (the output node). All the other nodes are input nodes themselves and so by definition receive no incoming connections. The large black rectangle in the first column refers to the value of the weight connecting the bias to the output node. The smaller white rectangle in the second column codes the connection from the first input node to the output node. The slightly larger white rectangle in the third column codes the connection from the second input node to the output node.

Exercise 3.5

-
- Why do you think the fourth column is empty?
-

The Hinton diagram in Figure 3.17 gives us a fairly good idea as to how the network has solved Boolean AND. Notice that the bias has a strong negative connection to the output node while the two input nodes have moderately sized positive connections to the output node. This means that one active input node by itself cannot provide enough activation to overcome the strong negative bias and turn the output node on. However, two active input nodes together can overcome the negative bias. This situation is exactly what we need to solve Boolean AND: The output node only turns on if both input nodes are active.

Sometimes, we need a more accurate picture of the internal structure of the network. For example, we might need to know the exact value of the weights in the network. `tlearn` keeps an up-to-date record of the network's state in a weights file. These files are saved on the disk at regular intervals (which you, the user, can specify). `tlearn` also saves a weights file at the end of each training session. You will find the current weights file in the same folder as your project files. The weights file should be called `and.1000.wts`. You can Open this file from the File menu in `tlearn`. The contents of the file are shown in Figure 3.18. The file lists all the connections in the network grouped according to receiving node. In the `and.cf` file only



```

and.1000.wts
NETWORK CONFIGURED BY TLEARN
# weights after 1000 sweeps
# WEIGHTS
# TO NODE 1
-1.756069
0.896755
1.067833
0.000000
|
9

```

FIGURE 3.18 The weight configuration stored after 1000 sweeps of training in the file `and.1000.wts`. The file lists the connections to each receiving node. For each receiving node, input connections are listed starting with the bias, then the input nodes and all other nodes in the network in ascending order as identified in the `.cf` file.

one receiving node is specified—the output node or node 1. Connections going into each node are listed from the bias node through the input nodes to the higher numbered nodes as specified in the `.cf` file. The first number in the `and.1000.wts` file (after the `# TO NODE 1` line) represents the weight on the connections from the bias node to the output node. The second number (0.896755) shows the connection from the first input node to the output node. The third number shows the connection from the second input node to the output node. The final number (0.000000) shows the connection from the output node to itself. Recall that this connection is non-existent—we are using a simple feedforward network here. Nevertheless, the format of the weights file is defined such that all possible input connections from every potential sending node are specified. With more complicated network architectures you will find that this seemingly unnecessary complexity has some saving graces!

Exercise 3.6

-
- Now that you know the precise configuration of the network, calculate by hand output activations for each input pattern and see if you can confirm the network's calculations as depicted in the Output window (see Figure 3.15).
-

Network training can be continued by selecting the **Resume training** option on the **Network** menu. **tlearn** will automatically extend training by another 1000 sweeps and adjust the error display to accommodate the extra training sweeps. Try training the network for an extra couple of thousand sweeps and observe whether the RMS error decreases significantly. Then practice the techniques you have learned in this section for evaluating performance and the state of the network.

The role of the start state

The network solved Boolean AND starting with a particular set of random weights and biases. Now run the simulation again but with a different set of initial weights and biases. This is easy to do. Just use a different random seed. Open up the **Training Options** dialogue box and select a different random seed (say 2). When you issue the command **Train the network**, **tlearn** wipes out the learning that has taken place in the network and provides a new set of random weights determined by the random seed you have used. Training continues in the usual fashion. You can resume training beyond the specified number of sweeps using the **Resume training** option.

Exercise 3.7

-
1. Train the network using a variety of different random seeds. Does the network show the same pattern of error for all the random start states?
 2. Does the error always decrease from one point on the error plot to the next?
 3. If two simulations exhibit the same level of error at the end of training, does this mean that the connections weights in the network are identical?
 4. Does the size of the weight limit parameter (specified in the **.cf** file) influence the outcome of network training? Try running a simulation with a large weight limit such as 4.0.
-

Start states can have a dramatic impact on the way the network attempts to solve a problem and on the final solution it discovers.

Researchers often attempt to replicate their simulations using different random seeds to determine the robustness or reliability of network performance on a given type of problem. Training networks with different random seeds is like running subjects on experiments. You repeat the experiment to determine the degree to which the outcome depends upon the participating individual or other factors of interest. Alternatively, running a simulation with different random seeds might be likened to evaluating the fitness of different organisms to adapt to an environmental niche. The initial state of the network corresponds to the organism's phenotype.

The role of learning rate

Recall that you specified the **Learning rate** parameter in the **Training options** dialogue box to be 0.1. Now investigate the impact of the learning rate for the network's performance on Boolean AND. Learning rate determines the proportion of the error signal (or more accurately δ) which is used to change the weights in the network. Large learning rates lead to big weight changes. Small learning rates lead to small weight changes (see Equation 1.5 on page 13). In order to examine the effect of learning rate on performance, you need to run the simulation in such a fashion that learning rate is the only factor that has been changed. In particular, you need to start the network off in the same state as before, i.e., with the same set of random weights and biases. Then you can compare the results of the new simulation with your previous run. Again notice the similarity to running controlled experiments. We know that the start state of the network can have a dramatic effect on learning so we avoid confounding experimental variables by using an identical start state.

Open the **Training Options** dialogue box and select a random seed of 1. This ensures that the network starts off in an identical fashion to previous runs we have observed. Next set the **Learning rate** parameter to 0.5. Finally, make sure that you have selected the **Train Randomly** button. Close the dialogue box and **Train the network**.

Generally, modelers use a small learning rate to avoid large weight changes. Large weight changes made in response to one pattern can disrupt changes made in response to other patterns so that learning is continually undone on consecutive pattern presentations. In addition, large weight changes can be counter-productive when the network is

Exercise 3.8

-
- Do you notice any difference from the previous run in the final solution that the network discovers? What is the effect of changing the learning rate? Try repeating this experiment with two or three other values of the learning rate parameter.
-

close to a solution. The weights may end up in a configuration which is further away from the optimal state than it was prior to the weight change!

Logical Or

Boolean AND is solved relatively quickly by a single-layered perceptron across a fairly wide-range of learning conditions (start state, learning rate, etc.). Now evaluate the network's capacity to master Boolean OR—the second of the Boolean functions listed in Table 3.1.

Exercise 3.9

-
- What type of network architecture should you use for Boolean OR?
-

You can set up the files necessary for running an OR simulation by opening the **New project...** option in the **Network** menu. Three files will be opened by **tlearn—or.cf**, **or.data** and **or.teach**. Initially, they will be empty files. Configure these files to contain the same information as the corresponding **and** files, except for the **or.teach** in which the target patterns should be those indicated in the output activations of the OR column in Table 3.1. If you wish, you can just copy all the **and** files and rename them with their appropriate **or** titles. When you open a **New Project...** entitled **or**, **tlearn** will use the duplicate files you have created. You will then only need to edit the **or.teach** file.

Activate the **Error Display** and then **Train the network** using the same **Training options...** that you used for the initial run on Boolean AND

Exercise 3.10

1. Before you set up the network and run the simulation, can you predict how the network will attempt to solve this problem?
2. What does the `or.teach` file look like?

(see Figure 3.11). Remember to set the training options before you attempt to train the network. Use a random seed of 1, a learning rate of 0.1 and select patterns to Train randomly. In this manner, we can compare network performance on Boolean OR and AND directly. All we've changed is the target output activations. Let's evaluate network performance after 1000 sweeps of training. The RMS error curve is shown Figure 3.19. Compare this with the performance on Boolean

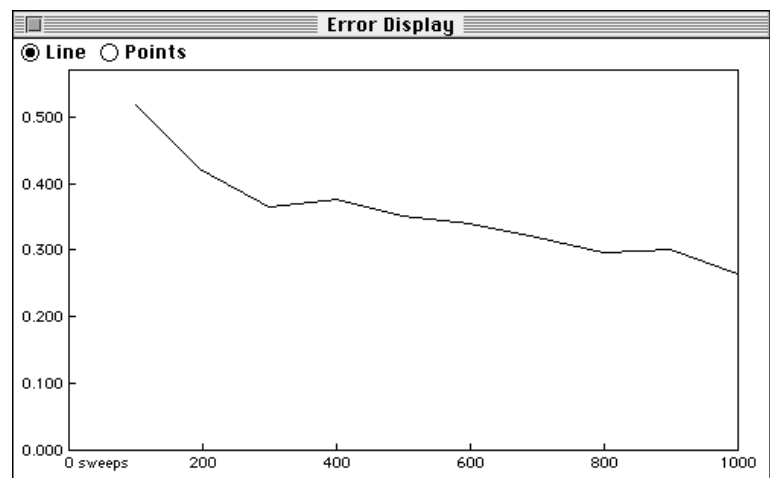


FIGURE 3.19 Network performance over 1000 sweeps of training on Boolean OR.

AND depicted in Figure 3.14. The error decreases faster and to a lower level by 1000 sweeps. This suggests that the network might have solved Boolean OR if we evaluate error in accordance with the “rounding-off” criterion (see Exercise 3.3 on page 63). However, as we saw in the section on **Pattern Error** on page 49, the RMS error may provide a misleading picture of network performance. It is necessary to

examine the errors on individual patterns to be confident that the network has indeed solved the problem.

Earlier in the chapter (see Exercise 3.4 on page 65), we used the Verify network has learned option in the Network menu to obtain a set of output activations for the network trained on Boolean AND. We used a rounding-off method to compare output activations with the target activations and thereby determined whether the network had solved the problem. However, **tlearn** provides additional techniques for evaluating the error on individual patterns. Select Testing Options... in the Network menu. The dialogue box shown in Figure 3.20 will appear. Testing Options... sets a variety of parame-

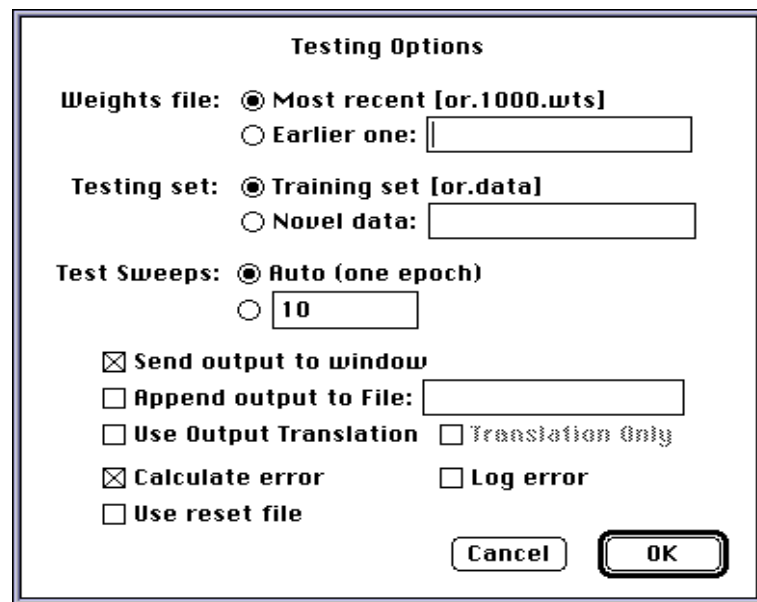


FIGURE 3.20 The Testing Options... network dialogue box which is accessed from the Network menu.

ters for evaluating network performance without further learning taking place when the Verify network has learned or Probe selected nodes options in the Network menu is selected. Take some time to

browse through the options in this dialogue box and see if you can guess their functions.

We want to test the network's output for individual patterns so we will eventually use the **Verify network has learned** option in the **Network** menu. By default, this option tests the network's response to the input patterns in the training set using the current state of the weights in the network. In Figure 3.20 you can see that the **Weights file:** option is selected for the **Most recent** weights file, i.e., the current state of the network. This is just the weights file **or.1000.wts** which **tlearn** selects automatically. If you wanted you could select another weights file by selecting the **Earlier one:** button and typing the name of an appropriate weights file in the text box. Alternatively, you can double-click on the text box to bring up a list of all the weights files in your current folder. This is a useful aid to memory if you can't remember the names of your weights files but make sure you select a weights file that is appropriate to your current network architecture. So far this isn't a problem because we've always used networks with identical architectures. But once you start using networks that vary in the number of connection weights, it will be crucial to select the right type of weights files. **tlearn** will not complain that you've chosen the wrong file. It will even produce output activations when you **Verify network has learned!** Load your weights files carefully.

The second decision to make in the **Testing Options...** dialogue box concerns the **Testing set:** Do you want to evaluate performance on the patterns in the training set or the response of the network to an entirely new set of patterns. The latter can be important for discovering how the network generalizes (more of that in Chapter 6). For the moment, we are interested in performance on the training set so select the **Training set** button. Make sure that **tlearn** has selected the correct training set—in this case **or.data**.

Next set **Test Sweeps:** to **Auto (one epoch)**. This tells **tlearn** to present each training pattern to the network just once and in the order specified in the **.data** file when we select the **Verify network has learned** option. Finally, set all the other squareboxes at the bottom of the dialogue box in Figure 3.20 as indicated. In particular, make sure that the **Calculate error** box is on. This tells **tlearn** to display the error for individual patterns when you choose the **Verify network has learned** option. The **Error Display** used for the RMS error plot is also used for this option.

When you have selected the relevant options, close the Testing options... dialogue box and select Verify network has learned. If all is well, you should see an Error Display like that in Figure 3.21. Per-

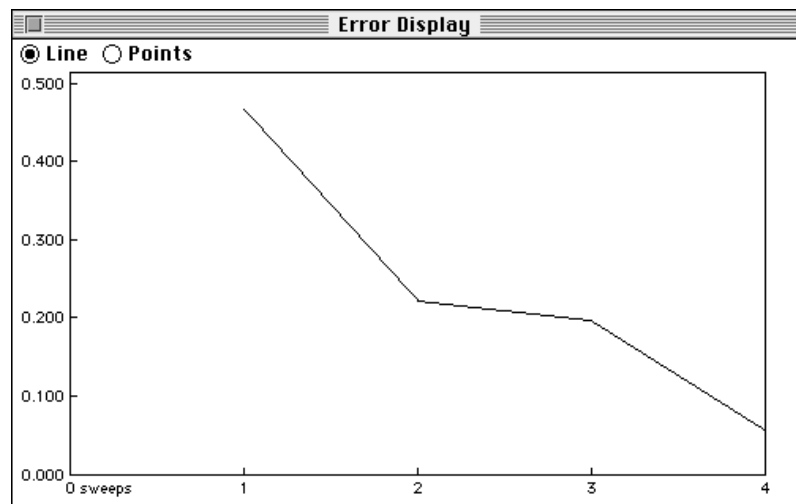


FIGURE 3.21 Error display for individual patterns in Boolean OR.

formance on all patterns is within criterion though the error is least on pattern 4 (1 1). You might like to train the network further (using the Resume training command) to see how quickly the error reduces on the other training patterns.

Exercise 3.11

-
- How did the network solve Boolean OR? Use the techniques that we reviewed in Exercise 3.6 on page 66 to answer this question. Is the network sensitive to training with different random seeds?
-

Exclusive Or

Create a new set of files for the EXCLUSIVE OR problem shown in Table 3.1. Create the project file by selecting the **New Project** option in the **File** menu. Call the project **xor**. If you have already copied the files over from the **or** project to create **xor.cf**, **xor.data** and **xor.teach**, all you need to do is edit the **xor.teach** file so that it has the right outputs as specified in Table 3.1. Otherwise, you will need to create the network files from scratch.

Once you have created the necessary files, set the training options so that they conform to the parameters that you have used previously for **and** and **or**. Now train the network for 1000 sweeps. Evaluate performance on XOR.

Exercise 3.12

- Has the network solved XOR? If not, try **Resume training** for a further 4000 sweeps. Does the network solve the problem?
-

You will discover that your network architecture has considerable difficulties solving XOR over a wide range of start states (random seeds) and learning rates. In fact, a single-layered perceptron is unable to solve XOR. The difficulty arises with the non-linearity inherent in the Boolean mapping. In the next chapter we will examine the type of network XOR architecture and training regime that permits the solution of this problem.

Answers to exercises

Exercise 3.1

- The set of input patterns are identical for all the Boolean functions represented in Table 3.1. AND, OR and XOR differ in the set of responses to the 4 inputs. For example, AND demands the response 0 to the input pattern 0 1 but OR demands a response of 1 to the same input pattern. Therefore, we must expect our network to exploit a different set of connection weights depending upon which Boolean function it is attempting to compute.

The output responses required for the Boolean functions AND and OR increases with the amount of activity at the input. So the input 0 0 produces 0 output for AND and OR while the input 1 1 produces an output of 1. In contrast, for the Boolean function XOR, there is a non-linear relation between the amount of activity at the input and the activity at the output—the inputs 0 0 and 1 1 both produce the output 0. In this respect, XOR is the odd-one-out of these 3 Boolean functions. This fact will have important implications for network training as we shall see in “Exclusive Or” on page 62.

Exercise 3.2

- There are 4 different input patterns so the value of $k = 4$.

Exercise 3.3

1. There are 4 input patterns in the training environment so within 1000 sweeps the network will see each pattern 250 times. This is, in fact, only an approximation since the simulator selects patterns at random from the training set. However, for increasing numbers of training cycles, the percentage difference in selection of given patterns diminishes. In other words, the patterns in the training set are selected with almost equal frequency. For the statistically minded reader, patterns are selected from the

training set *with replacement*. You can force the network to select randomly *every* pattern in each training epoch by deactivating the **With replacement** check box in the enlarged **Training options** dialogue box. Click on the **more** button to reveal a new universe of training options!

2. The answer to this question may seem obvious: The network has solved the problem when the RMS error has been reduced to zero. However, there are several complications which mitigate against this solution. First, recall that the activation function of the output unit is the sigmoid function defined in Equation 1.2 and depicted graphically in Figure 1.3 on page 5. The net input to the node determines the node's activation. You can observe in Figure 1.3 that the activation curve never quite reaches 1.0 nor reduces to 0.0. In fact, to achieve these values the net input to the node would need to be $\pm\infty$ (infinity) respectively. Nodes never receive $\pm\infty$ input so there will always be a residual finite error. The question then arises as to what level of error is acceptable.

There is no single correct answer. For example, you might suggest that all outputs should be within 0.1 of their target. So if the target is 0 then outputs > 0.1 should be considered incorrect and if the target is 1 then outputs < 0.9 should be considered incorrect. However, these criteria have been set in an arbitrary fashion. Why not choose a criterion of 0.2 rather than 0.1? An alternative solution would be to round off the activation values. Activations closest to 1.0 are judged to be correct if the target is 1.0. Activations closest to 0.0 are judged to be correct if the target is 0.0. In effect, this alternative sets the criterion to 0.5. To repeat, there is no *right* answer in setting the error criterion. Many researchers accept a rounding off procedure. Others set more stringent demands. It can be useful therefore when evaluating the performance of the network to determine how different values of the error criterion effect the picture of performance.

Let us assume for the time being that we use rounding off to determine whether the output is correct or incorrect. We want to know what level of global RMS error *guarantees* that all the patterns have been learned to criterion. Consider the case where all patterns are 0.5 off their targets, i.e. $|t - o| \leq 0.5$. Substituting in Equation 3.1 gives an overall RMS error of 0.5:

$$\text{rms error} = \sqrt{\frac{0.5^2 + 0.5^2 + 0.5^2 + 0.5^2}{4}} = 0.5 \quad (\text{EQ 3.2})$$

However, it is also possible to observe a RMS error ≤ 0.5 in which some of the patterns are still categorized incorrectly. For example, errors could be 0.6, 0.6, 0.1 and 0.1 to yield an RMS error of 0.43. So we need a more conservative error level to guarantee that all the patterns are correct. We can be confident that all the patterns are within criterion when only one pattern is contributing substantially to the error. This means that a maximum

$$\text{RMS error} \leq \sqrt{\frac{(0.5^2)}{4}} = 0.25$$

guarantees that all the patterns have been categorized correctly. Of course, this value holds only for binary targets to 4 training inputs. Different levels of global error will be appropriate for other problems.

Exercise 3.4

1. If we use a rounding-off method to evaluate network performance then we get the output values indicated in Table 3.2. The exact output values are indicated in the **Output** column and their rounded values in the

TABLE 3.2 Activation values and errors in Boolean AND.

Input	Output	Rounded Off	Target	Squared Error
0 0	0.147	0	0	0.022
1 0	0.297	0	0	0.088
0 1	0.334	0	0	0.112
1 1	0.552	1	1	0.201
RMS Error				0.323

Rounded Off column. The **Target** scores match the rounded values exactly. So by this error criterion, **tlearn** has solved Boolean AND.

2. The exact RMS error is calculated according to Equation 3.1 (see page 48). In Table 3.2 the **Squared Error** for each input pattern is calculated by squaring the difference between the output and target values. The RMS error is just the square root of the average of all the squared

errors, i.e., 0.323. Notice this error score is identical to the error plotted by `tlearn` in Figure 3.14 after 1000 sweeps of training. This should come as no surprise. The two error scores should be identical!

Exercise 3.5

- The fourth column codes the connection from the fourth node in the network to the output unit. In this case, the fourth node is just the output node itself. The first three nodes were the bias and the two input nodes. So the fourth column depicts the connection between the output node and itself—a recurrent connection. The network we are currently examining is a feedforward network. There are no recurrent connections so the fourth column is empty.

Exercise 3.6

- To calculate the activation of the output node for each input pattern we need to find the sum of the weighted input activations and use the sum as the input to the logistic activation function (see Exercise 1.1 on page 8). We'll display these calculations in table format—one section of the table

TABLE 3.3 Calculating output node activations

		Input Activation	Connection Strength	Weighted Activations	Sum of Weighted Activations	Logistic of Net Input
Pattern 1	Bias	1	-1.756069	-1.756069	-1.756069	0.147
	Input One	0	0.896755	0		
	Input Two	0	1.067833	0		
Pattern 2	Bias	1	-1.756069	-1.756069	-0.859314	0.297
	Input One	1	0.896755	0.896755		
	Input Two	0	1.067833	0		
Pattern 3	Bias	1	-1.756069	-1.756069	-0.688236	0.334
	Input One	0	0.896755	0		
	Input Two	1	1.067833	1.067833		

TABLE 3.3 Calculating output node activations

		Input Activation	Connection Strength	Weighted Activations	Sum of Weighted Activations	Logistic of Net Input
Pattern 4	Bias	1	-1.756069	-1.756069	0.208519	0.552
	Input One	1	0.896755	0.896755		
	Input Two	1	1.067833	1.067833		

for each input pattern in Table 3.3. The actual output from the network is the number listed in the column entitled “**Logistic of Net Input.**” These numbers match the output activations in Figure 3.15 exactly. Notice how the bias node effectively keeps the output node switched off (closer to 0 than 1) for the first 3 input patterns.

Exercise 3.7

1. Different random seeds result in different initial configurations of the connection weights in the network. So the presentation of training patterns to the network will result in different output activations for each random seed. This means that at the beginning of training different errors will be observed and this will, in turn, affect the way the weights are changed. In general, the error profile will be different for each random seed you train the network on. However, remember that each input pattern is trained on an associated target pattern. As the error on the output diminishes, the changes made to the weights in the network will diminish. The error then will change more slowly. In other words, different random seeds are likely to produce considerable variation in the error profile during the early stages of training but will show similar error profiles later in training when the error has been reduced. The error profile for running the network on two random seeds (1 and 6) are shown in Figure 3.22.
2. Although error generally decreases gradually during training, it need not do so in a monotonic fashion. For example, the error curve in Figure 3.22 (Seed 1) temporarily rises around the 400 sweep mark. This may seem odd given that the learning algorithm is continually attempting to reduce the error. There are several possible causes for U-shapes in our error curve. The less interesting reason is that the patterns most

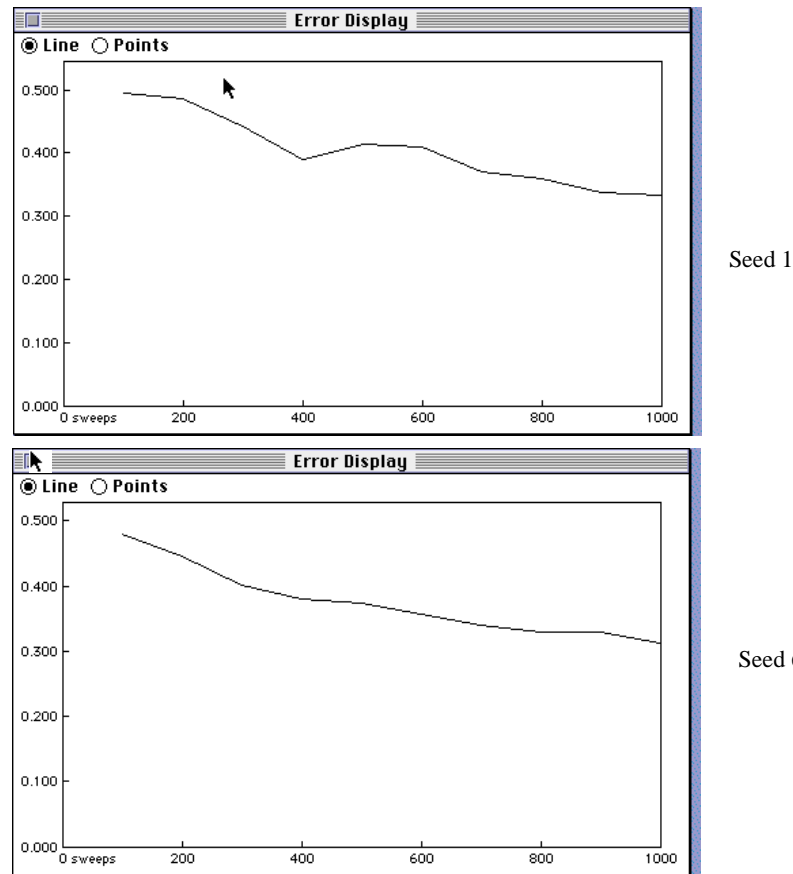


FIGURE 3.22 The error profiles for running Boolean AND on two different random seeds (1 and 6). Note that the final levels of error are very similar (about 0.35) but the trajectory of the error profiles are quite different.

recently sampled from the training set happen to have a large error associated with them. In other words, the reported error need not necessarily reflect the average RMS error for the 4 different training patterns—only a couple of them may have been selected for presentation to the network and those patterns may happen to have a large error.

A more interesting cause of the increase in the error is *interference* between different training patterns on the connection weights in the net-

work. Consider a single input pattern. It is supposed to produce a specific output. There exists a configuration of connections in the network that will permit this. However, the network is also supposed to produce specific outputs for the other input patterns *using a single set of connections weights*. It is by no means obvious that the configuration of connections weights used for one pattern will be appropriate for the other patterns. If subsequent training patterns are not compatible with earlier ones, then further training will lead to changes in connections that mitigate against successful performance on the original pattern. Often the average error across all the patterns in the network will still decrease despite the increase in error on one of the patterns. However, sometimes the interference can be substantial and the average error may in fact increase.

3. Different networks can exhibit the same level of error for a given problem but need not arrive at those errors by the same route. In other words, the weights can look quite different even though they produce the same answer. For example, consider the weights files after 1000 sweeps of training for the two error curves in Figure 3.22. These are shown in Figure 3.23. The bias node has a stronger inhibitory effect in the “Seed

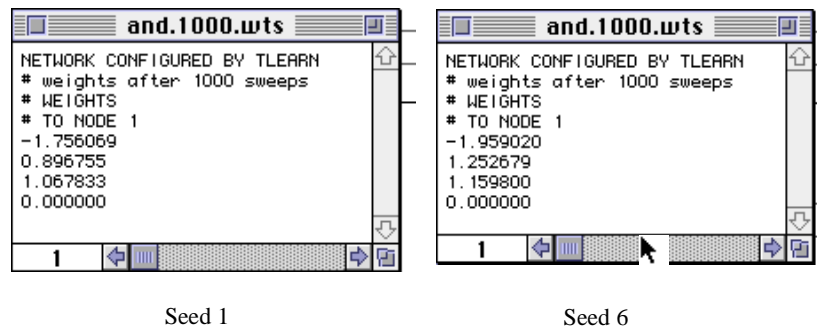


FIGURE 3.23 Two weights files for 1000 sweeps of training on Boolean AND. The final error is approximately the same but the weights are quite different. The bias node has a stronger inhibitory effect in the “Seed 6” simulation and the relative strengths of the connections from the input nodes to the output node are reversed.

6” simulation and the relative strengths of the connections from the input nodes to the output node are reversed. In general, there will be many solutions (configurations of the connection weights) to a problem. The

solution that the network finds will depend on the initial start state, the number of times it sees a particular pattern and the order in which it sees different patterns.

4. Recall that the weight limit parameter in the `.cf` file determines the range of weight values that are randomly assigned to the connections in the network when it is initialized. If the weight limit is set to 4.0, then all the connections in the network will be set to values in the range ± 2.0 . In other words, some of the connections can be randomly set to be strongly excitatory (2.0) or strongly inhibitory (-2.0). We have already seen that the network's solution to Boolean AND is to build a strong inhibitory bias to the output node and moderate excitatory connections from the input nodes. If the initial weight assigned to the bias is strongly excitatory, the learning algorithm has more work to do to change it into a strong negative value. Similar difficulties will be encountered if the other two connections in the network have inappropriately large values. So a large weight limit has the potential to slow down learning in the network. Conversely, learning can sometimes be accelerated if the the bias node is born with a strong inhibitory connection to the output node. Time to find a solution would then depend on the value of the other connections in the network.

The use of a large weight limit can create other learning problems. Recall from Equation 1.5 on page 13 that the changes made to a weight are proportional to the δ value calculated for the associated target unit. δ itself is determined by the error on the output unit multiplied by the first derivative of the unit's activation function (see Equation 1.3). The first derivative of the activation function is simply the slope of the activation curve shown in Figure 1.3 on page 5. For large positive net input the output activation of a sigmoid unit is at its maximum, i.e., 1.0. However, the slope of the curve is flat. In other words, the value of the derivative is close to 0.0. Similarly, with large negative net input the slope of the curve is flat so its derivative is close to 0.0. When the derivatives are small, Equation 1.3 forces the δ s to be small and so the weight changes will be small. In summary, extreme values of net input (positive or negative) lead to small weight changes. In effect, the units become saturated with input and find it difficult to learn. We will see later that this also has some beneficial side-effects.

Now a large weight limit can yield strong initial connections. Strong connections can yield extreme values of net input which lead to small weight changes. So large weight limits can slow down learning. Gener-

ally, connectionist modelers keep the initial weights small so they are not over-committed at the beginning of learning. A weight limit of 1.0, i.e., a range of ± 0.5 seems to work quite well. This tends to keep the sigmoid units within their most sensitive range for learning.

Exercise 3.8

- Sometimes a higher learning rate will give you faster learning. The error curve with learning rate set to 0.5 in the Boolean AND problem is shown in Figure 3.24. In this case, learning benefits from a higher learn-

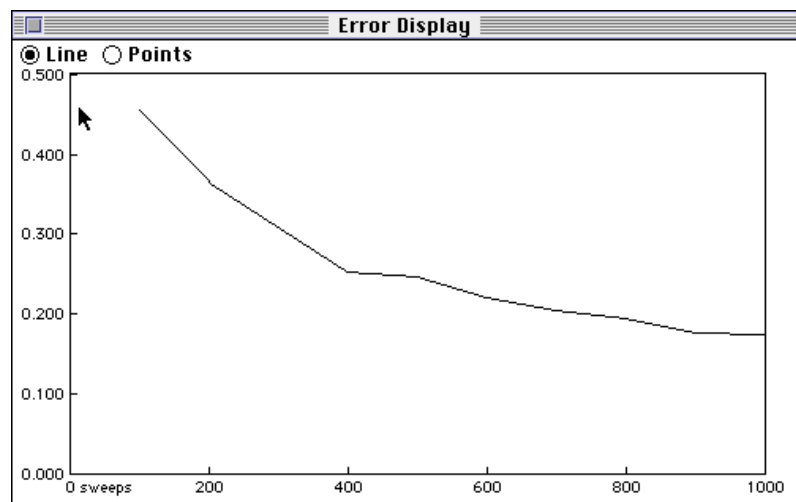


FIGURE 3.24 The error curve for Boolean AND with a random seed of 1 and a learning rate of 0.5. Learning is faster with the higher learning rate—compare with Figure 3.22.

ing rate. For example, the average error after 1000 sweeps is approximately 0.2. With a learning rate of 0.1 the average error after 1000 sweeps was closer to 0.35 (see Figure 3.22).

However, higher learning rate does not always lead to faster learning for all the patterns in the training set. We saw in Exercise 3.7 that input

patterns in the training set can interfere with each other. If the learning rate is high, the likelihood of interference occurring is enhanced. The weight changes made for a single pattern presentation can have a detrimental effect on earlier training. Normally, it is safer to train the network with the learning rate parameter set to a small value unless there is good reason to believe that interference between patterns is likely to be minimal. Can you think why a higher learning rate seems to help in the Boolean AND problem?

Exercise 3.9

- You need a network that takes two inputs and produces a single output activation, i.e., two input nodes and a single output node. For good measure include a bias node. We will investigate its role in a later section.

Exercise 3.10

1. In Boolean AND the bias played an important role in keeping the output node switched off for all the input patterns except for 1 1. In Boolean OR the output unit should be switched on for all patterns except the first—0 0. So we would expect the bias to play a different role in the network for this problem.

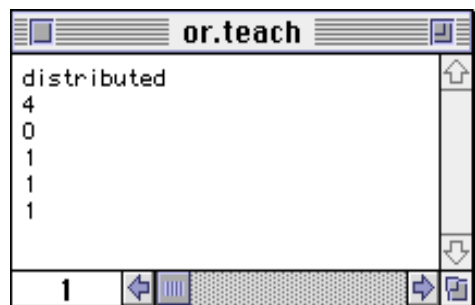



FIGURE 3.25 The `or.teach` file.

- The `or.teach` file is shown in Figure 3.25. All output targets are set to 1 except the first (0) which codes the target for the input 0 0. The other `or` files are identical with their `and` counterparts since the input patterns and network architecture are identical.

Exercise 3.11

- The weights file gives the best clue as to how the network has solved Boolean OR. Open the weights file `or.1000.wts` (or the most recently saved weights file) from the File menu. The weights file after 1000 sweeps with a random seed of 1 is shown in Figure 3.26. Notice



```

or.1000.wts
NETWORK CONFIGURED BY TLEARN
# weights after 1000 sweeps
# WEIGHTS
# TO NODE 1
-0.154080
1.481331
1.438950
0.000000

```

FIGURE 3.26 The `or.1000.wts` file. Note the small negative bias connection but relatively large positive connections from the input nodes.

that the connections from the input nodes are large and positive (1.48 and 1.44 respectively) which means that activity at either or both of the input nodes will tend to turn the output node on. The connection from the bias is negative but too small to counteract the contributions from active input nodes. In this example, the bias ensures that the output node is switched off when neither of the input nodes are active. Remember that the activation function for the output node varies continuously between 1 and 0 according to the logistic of the net input to the node (see Figure 1.3 on page 5). Net input of 0.0 to the output node produces an activation of 0.5 so a negative input is needed to switch off the output node when both input nodes are dormant (0 0). The mildly negative bias

connection fulfils this function. However, the absolute size of the bias connection must remain smaller than either of the other two weights so that a single active input node can switch on the output node.

Exercise 3.12

- Given the network architecture that you have employed, the network will fail to solve XOR. Correct responses may be produced for some of the input patterns. With a random seed of 1 the network fails to produce correct responses for any of the patterns after 5000 sweeps of training. Try a variety of random seeds and learning rates. You will discover that the network manages to get some of the responses correct but *never* all of them at the same time.