# CHAPTER 6 Generalization

## Why is generalization important?

We rarely train networks on random collections of data. Typically, we choose the data because they come from some domain in which we are interested. Sometimes we know in advance what the relationship is between the input/output pairs and our goal is to see whether the network can discover it. An example of this is the XOR function; we know that this is a difficult function and we may wish to study the conditions under which it may or may not be learned. Other times, we know there is a regular relationship but may not be able to formulate it precisely; our goal in this case is to use the network as a discovery device to uncover hidden relationships and make the mapping function explicit. For example, when Lehky & Sejnowski (1990) trained a network to determine shape from shading information, they were able to analyze the network afterwards in order to discover the computational primitives used to solve the problem.

The important point in both cases is that we assume the network has *extracted some generalization from the training data.* That is the whole point of training. Whether our purpose is to see whether the network *can* generalize, or use the network to help discover *what* the generalization is, we want the network to induce the underlying regularity from the examples given.

## *How do we know when a network has generalized?*

Let us imagine we train a network on some set of data. After some number of training cycles, we observe that the error on the dataset (say, averaged over all the items) has dropped to a low number.

How low is low enough? Well, here we must be careful. Let's say the training set consists of 100 patterns; each pattern is itself a 100-element vector with one of the bits set to 1 and the remaining 99 bits set to 0. That is, the desired targets would look something like:

**TABLE 6.1** Sample data

| **Pattern number** | **Desired outputs** (each vector has 100 elements) |
|---|---|
| #1 | 10000. . .000 |
| #2 | 01000. . .000 |
| #3 | 00100. . .000 |
| . . . | . . . |
| #99 | 00000. . .010 |
| #100 | 00000. . .001 |

Let's also assume that over the course of training, we monitor the RMS error and observe the pattern shown in Figure 6.1. At the outset of training, RMS error is close to 50; after 1,500 training sweeps the error has dropped to approximately 1.0. This seems good; based on the mean error we might feel the network has learned the dataset.

Surprise! In this example, a mean error of 1.0 might actually result from very bad performance. How might this be? Let's think about our training data in detail and ask ourselves what it would take for the network to achieve such a performance. Each output pattern is of the same basic form (100 0's with a different single bit set to 1) so we can pick an arbitrary teacher pattern and focus on that. Let's pick the first pattern: 10000. . .000. When networks are initially configured, connections are often assigned weights drawn from a uniform distribution ranging say from -1.0 to 1.0. The mean of the weights on incoming lines to a node is thus 0.0. Therefore, net input (the product of weight times activation on sending units) will tend to be close to

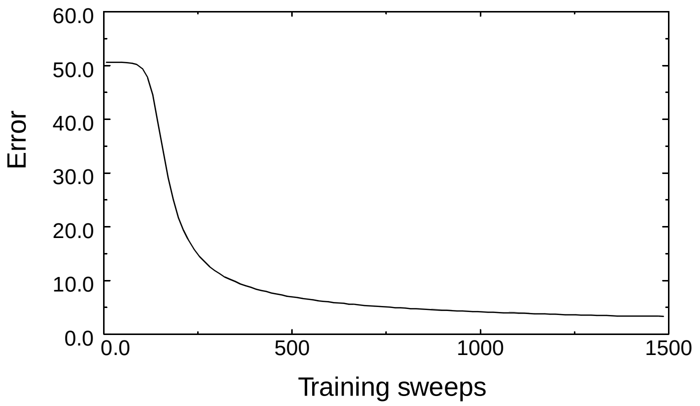


**FIGURE 6.1** Hypothetical error plot after training a network to produce the outputs shown above. After 1500 pattern presentations, the error is slightly above 1.0.

0.0. Assuming a sigmoid activation function, nodes will therefore usually have activations close to 0.5 at the outset of learning. (This actually makes sense, from the viewpoint of making it easy for a node to move in either direction.) Given a target pattern of 10000...000, this will yield an initial RMS error of approximately 50.0. (The first unit has an activation of 0.5 when it should be 1.0, giving an error of 0.5; and the remaining 99 units are 0.5 when they should be 0.0, yielding a summed error of 49.5) This is why the error shown in Figure 6.1 starts off close to 50.0.

Now consider the problem facing the network. Each pattern has 99 0's and a single 1, always in a different position. Given the large number of 0's which are present in each pattern, a quick way to reduce error across the entire pattern set would be to turn *all* of the output nodes off. If the network does this, it will get 99 of the output nodes correct (so there will be 0.0 error from them) and only one output node wrong. The total error will thus be 1.0—quite a dramatic improvement over the initial error! This is what we see in the latter portion of Figure 6.1. It should be clear, however, that although the network has in some sense learned an important feature of the training set (i.e., that patterns contain a preponderance of 0's), it hasn't learned the part we might be more interested in (i.e., which bit position has a 1, on any given pattern). The lesson of this example is that we need to think carefully about the criteria for success. If we are going to use a global statistic such as mean RMS error over a pattern set, we must be careful to figure out what performance could give rise to different ranges of error, and what error levels would be indicative of success (however we chose to define it).

***Exercise 6.1***

- Consider the pattern set described in the previous section (i.e., 100 patterns, each containing 99 0's and a single 1). Let us establish the criterion that to be successful, the network must (a) have output activations not greater than 0.1 on bits which should be 0.0, and (b) have an output activation not less than 0.9 on the bit which should be 1.0. (This is a somewhat stringent criterion; in reality, we might be satisfied with outputs of 0.3 or 0.7.) What error level must we have on a single pattern to be guaranteed that the network's performance meets this criterion?

In previous chapters, you've used a network to approximate a function associating a set of input patterns with a set of output patterns. In the XOR task explored in Chapter 4, for example, a network was trained by presenting all possible input patterns and the appropriate output. In many cases, however, it is preferable to withhold some input/output pairings from the training set, and to use these patterns to test the fully trained network. If the network performs correctly on these novel patterns (i.e., the network has never seen these particular patterns before), then one can feel confident that the network has induced a general function from the specific exemplars on which it was trained, rather than simply "memorizing" the training set. Such a network is said to have *generalized*.

One of the principal factors controlling generalization is the number of hidden nodes available to the network. If a network has sizable internal resources (relative to the task at hand), there will be little pressure on the network to find an efficient, and hence more general, solution. Rather, the network will simply memorize the patterns. (Note that the use of the term "memorize" in this context, while common among connectionist researchers, is somewhat misleading. A network *always* learns *some* function from input to output—in the sense that if you present a novel input pattern to the network, some pattern of activation will be produced across the output units—but the function may not correspond to what the researcher "had in mind.") If a network has too few hidden units, then it will simply be unable to learn the training set. A trade-off is thus established; give the network enough internal resources to learn the training set (to some criterial performance level), but no more (to encourage generalization). Finding this magic number of hidden units usually involves some trial and error.

**TABLE 6.2** An incomplete XOR

| INPUT | | OUTPUT |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |

A second important factor to keep in mind when trying to get a network to generalize is the selection of the training set. The subset of all possible patterns must properly sample the function space or the network won't have any chance of generalizing properly (i.e., to the function you had in mind). For example, what do you think would happen if you tried to get a **2x2x1** network (like the network in

***Exercise 6.2***

- What function do you think the network would learn from the data in Table 6.2? Would it generalize properly to the missing XOR pattern? Try it. Do you think it's possible to train a network on XOR with less than the full data set?

Figure 4.1) to learn XOR by training it on the three patterns in Table 6.2.

In this chapter, you will:

- Evaluate characteristics of the network's architecture and training data that promote generalization.
- Show how the response properties of the unit's activation function can influence generalization.

- Learn how to perform cluster analyses of hidden unit activations—another technique for getting at the similarity structures discovered by the network.

## *Analogue addition*

### Continuous Xor

In Chapter 4 you saw how a three-layer, feedforward network can learn to perform the XOR function via the backpropagation learning algorithm. The XOR task employs both binary inputs and binary outputs. But backpropagation is by no means restricted to binary inputs or outputs. For example, a continuous-input version of the XOR problem can be defined as in Table 6.3. The rule can also be expressed verbally as follows:

**TABLE 6.3** Continuous Xor

| INPUT | | OUTPUT |
|---|---|---|
| > 0.5 | < 0.5 | 1 |
| < 0.5 | > 0.5 | 1 |
| > 0.5 | > 0.5 | 0 |
| < 0.5 | < 0.5 | 0 |

> “Combine pairs of inputs such that if either is greater than 0.5 the output is a 1; however, if both are greater than 0.5 or less than 0.5, then the output is a 0.”

Thus, we have created a problem which—instead of having binary inputs and binary outputs—has continuous inputs and binary outputs.

Binary outputs are typical of *classification* problems in which the inputs (consisting either of binary or continuous-valued units) are classified by the network into one of several categories. Problems of

***Exercise 6.3***

- If we plot the input pairs as points on a Cartesian plane, the space of all possible inputs is the square region bounded by the points (0,0), (0,1), (1,0), and (1,1). The continuous-input XOR partitions this space into two distinct regions. Draw a graph illustrating these regions. Do you think this version of the XOR problem is easier or harder than the binary version for a network to solve? Why?

this sort essentially require the network to partition the input space into as many disjoint regions as there are output categories.

In the more general case, input and output units are *both* continuously valued. The role of the network then is to construct a continuous mapping from input values to output values based on the examples provided by the training data. In principle, a three-layer feedforward network is capable of performing *any* desired mapping from input to output. But this ignores the crucial questions of how many hidden units will be required, and how easily and accurately the mapping will be learned. Nevertheless, networks using backpropagation to learn continuous mappings have been quite successful in a number of real-life applications.

## The addition problem

In this section, you will examine a very simple example of a network learning to perform a continuous mapping from input to output: analog addition (i.e., the output value is to be the sum of the input values). As with XOR you will have two input nodes, two hidden nodes, and one output node. Since we are simulating an addition process, set the output node to have a linear activation function. To achieve this, just include the line

```
linear = 3
```

in the `SPECIAL:` section of the `.cf` file. This instruction sets all the nodes to the right of the equals sign to be linear units, i.e., their activation is the same as their net input. In this case there is only one output node—node 3. You may think that including linear nodes in the network trivializes the problem. Note however that all input activations must first pass through the layer of nonlinear hidden nodes before reaching the linear output node.

Create a New Project and call it **`addition`**. Let us agree to allow only inputs between 0.0 and 0.5; then we only expect output between 0 and 1. Furthermore, let's restrict the training data to numbers with only a single decimal place. This means that there are 36 possible training patterns. You can test the network on higher precision numbers later. Now you need to create a set of training data. This training set will consist of pairs of numbers between 0.0 and 0.5 and their corresponding sums. Unlike the XOR problem, do not present the network with an exhaustive list of all possible input/output combinations. Instead, devise what you believe is a representative subset of perhaps ten or so possibilities.

***Exercise 6.4***

- Run the network for 200 epochs[a] with a learning rate of 0.3 and momentum of 0.9. Now test the network. How well has the network learned the training data?

a. An epoch consists of a single presentation of all the training examples in the **`data`** file. Since there are, say, 10 input patterns in **`addition.data`** then 200 epochs consists of 2000 sweeps. Note that if you choose toTrain randomly there is no guarantee that all the input patterns will be presented on a given epoch, unless you determine that pattern selection is conducted in a random manner *without replacement* in the Training options dialogue box by deselecting the with replacement check box.

The deeper question here is how well the network has *generalized* from the training data. After all, the training data only informs the network as to what the input-output mapping should be for a small set of points in the input space. How do you know that the network will give the desired answers for inputs that were not included in the training set?

Another interesting question is how the network actually performs the addition, in view of the inescapable nonlinearity in the two hidden nodes.

***Exercise 6.5***

- How well does the network add *any* valid pair of inputs? You can create novel pairs of inputs in a new **data** file called **novels.data** and load those patterns into the network using the Novel data: option in the Testing Options… dialogue box. You do not need to specify an equivalent **`novels.teach`** file just as long as you switch off the Calculate error option in the Testing Options… dialogue box. Note however that the test file must end with the extension **`.data`**. before testing the network. Try a smaller or larger training set; how sensitive is the network to the size of the training set? What if the training set draws most of its examples from cases where both inputs are less than 0.25? How accurately does the resulting network add inputs greater than 0.25?

***Exercise 6.6***

- Draw the network. Show weights on each of the connections, and put the biases inside the nodes. Can you explain the principle that the network is using to perform the addition? Can you think of an alternative principle, i.e., a fundamentally different approach to getting a network with nonlinear hidden nodes to output a linear combination of its inputs?

## *Categorization*

Next consider a categorization function which takes 4-bit vectors as input and sorts them into two categories. The single output bit is ON when the input vector has exactly two bits ON (two and only two), in all other cases the output bit is OFF. Thus, six of the 16 possible inputs produce a 1 at the output, the rest produce a 0.

***Exercise 6.7***

1. Build a **`4x3x1`** network. Call the project **`gen`**. Create the necessary **`data`** and **`teach`** files according to the function described above, but withhold two of the 16 patterns as indicated in Table 6.4. Train the network using a learning rate of 0.3 and a momentum of 0.9 for 400 epochs. Test the network. Has the network learned to categorize the patterns correctly? If not, repeat the experiment with a different random seed until it succeeds.
2. Now test the network for generalization using the two patterns which you withheld from the training set. You will need to create a new **`data`** file and load it into the program with the appropriate set of weights. Has the network generalized properly? Do you think the same thing would have happened if you had withheld *any* two patterns? Test your hypothesis by repeating the experiment with different pairs withheld.
3. Try repeating the experiment with a **`4x2x1`** network. Does the network learn the training set? It may well not. Try different random seeds until the net converges. Why do you think it's so much more difficult with only two hidden nodes? That is, given that a solution *does* exist with two hidden nodes, why does the network fail so frequently to find it?

An additional hidden unit gives the network another dimension of freedom in traversing the weight space. Thus, the network is less likely to get caught in a local minimum. When you've found a random seed which works with a **`4x2x1`** network save the weights file. In the next section, you'll explore some techniques for analyzing the network's solution.

**TABLE 6.4** Test examples for **`gen`** project.

| INPUT | | | | OUTPUT |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |

## *Network analysis*

Once a network has learned a training set to some criterial performance level, the next step is to analyze the solution which the network has found. A variety of techniques are employed to do this, each of which gives valuable clues regarding network behavior. For example, you might probe the network with carefully selected inputs and look at the resultant hidden node and output patterns. This should give you some idea of the manner in which the network is transforming information. This is analogous to conducting a psycholinguistic experiment in which the network is the subject and the probe stimuli are selected to test hypotheses about what the network has learned. You should also examine the network performance across *each pattern* in the training set. Even though the total error may have dropped to criterion, the network may in fact be performing very well on some patterns, and only moderately well on others.

***Exercise 6.8***

- Does this differential performance suggest something about the nature of the network's solution?

Once these holistic techniques have been exhausted, however, the next step is to open up the network and look inside. Unfortunately, much like a brain, most networks simply look like mush when you examine their innards. Nonetheless, a couple of feasible alternatives do exist. With a small network, for example, you can simply draw (by hand) a big version of the network, write in the connection strengths and biases, then pass activation through the network by calculating it yourself. This was the technique you used in analyzing the XOR network in Chapter 4.

### Hand analysis

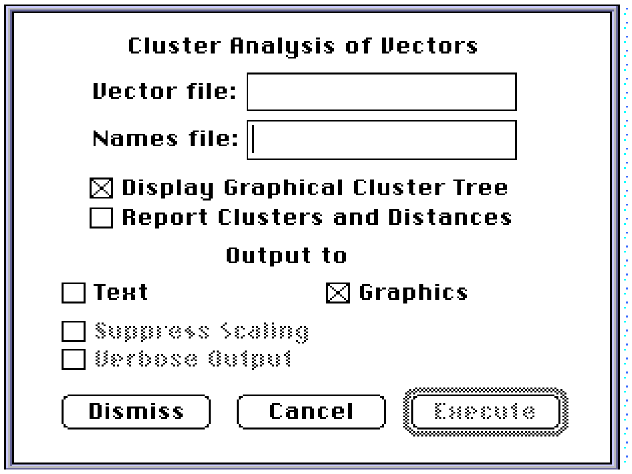
Try this technique again using the weights file you generated at the end of the section on categorization. Recall that in that experiment a network learned to take a 4-bit input string and (using just two hidden nodes) to output a 1 if the input had exactly two bits turned ON (two and only two). All other inputs produced a 0 on the output.

***Exercise 6.9***

1. Think about this task for a moment. Can you think of a solution to this problem (across two hidden nodes)? That is, under what conditions should each of the two hidden units turn ON, such that the network could properly complete the problem in the next layer of connections (from hidden to output)?
2. Now draw out the network you actually trained and pass several test patterns through the connections. (Remember that you can use the logistic table in Exercise 1.1 on page 8 to calculate the activation from the net input.)
3. Can you characterize the function of each of the two hidden units? Under what conditions do they each turn ON? How does this solution compare to the one you invented?

## *Cluster analysis*

Another technique for analyzing a network solution is to look at how the similarity structure of the input patterns is changed as a result of going from the input to the internal representation. **`tlearn`** possesses a utility called Cluster Analysis… (an option in the Special menu) which exists to aid you in this regard. This utility takes as input a set of vectors and performs a hierarchical cluster analysis, drawing a tree diagram of the similarity structure. The Cluster Analysis… dialogue box is shown in Figure 6.2. Start by seeing what sort of similarity structure is inherent in your input patterns. Use the **`tlearn`** editor to



Cluster Analysis of Vectors

Vector file:

Names file:

[✓ Display Graphical Cluster Tree]

☐ Report Clusters and Distances

Output to

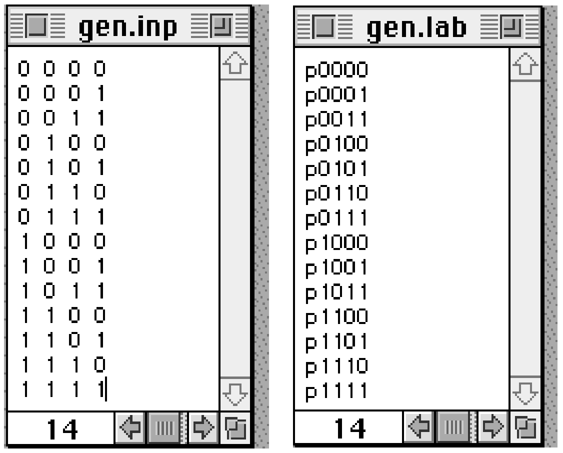
☐ Text [✓ Graphics]

☐ Suppress Scaling

☐ Verbose Output

**FIGURE 6.2** Cluster Analysis… dialogue box.

generate two files from your **`gen.data`** file. One file should contain just the 14 four-bit input vectors (call it **`gen.inp`**) and the other file should contain 14 pattern names. Your files should be identical with those shown in Figure 6.3. To cluster the input vectors open the relevant files in the Vector file: and Names file: boxes, select the Display



gen.inp

```
0 0 0 0
0 0 0 1
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

gen.lab

```
p0000
p0001
p0011
p0100
p0101
p0110
p0111
p1000
p1001
p1011
p1100
p1101
p1110
p1111
```

**FIGURE 6.3** Sample input files for the Cluster Analysis… utility.

Graphical Cluster Tree box and Execute the analysis. **tlearn** will display a window containing a cluster analysis on your screen as shown in Figure 6.4.

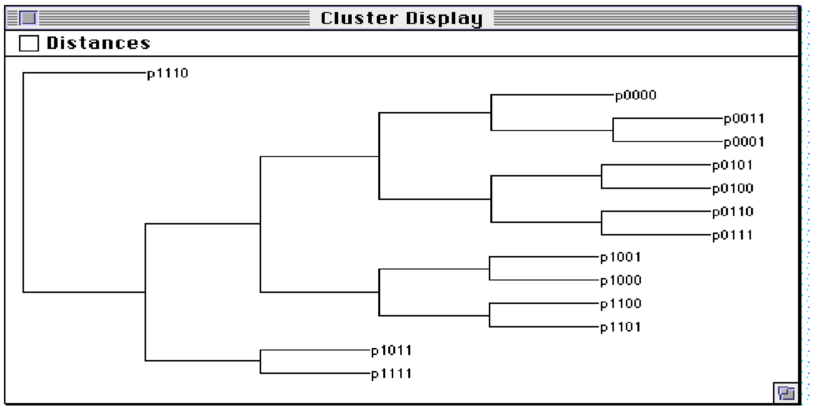


**FIGURE 6.4** Cluster analysis of the **gen.data** pattern files.

***Exercise 6.10***

- Can you see any similarity structure in the input patterns? Is this structure relevant to the task? Why not?

Now perform a similar analysis for the hidden node activations. You will need to test all the patterns in your training set and save the hidden node activations as you did in Exercise 4.5 on page 85. Then create the necessary files for the cluster analysis as above.

***Exercise 6.11***

- Has a meaningful similarity structure emerged at the hidden unit level? Does this agree with what you learned earlier about the function of each of the two hidden nodes?

## *The symmetry problem*

These are the basic techniques of network analysis. Try using them yourself on the following problem, called the symmetry problem. Build a **`6x2x1`** network, which will take 6-bit inputs and output a 1 if the input pattern is symmetric, i.e., if the last three bits mirror the first three. For example, **`0 1 0 0 1 0`** and **`1 1 0 0 1 1`** are symmetric, while **`0 1 1 0 1 1`** is not. Thus, eight of the 64 possible input patterns are symmetric. Note that it *is* possible for a network to solve this problem with only two hidden units, but it may require some fiddling with the random seed, learning rate, and momentum.

***Exercise 6.12***

- Once the network has learned the problem, analyze its solution using the techniques discussed in this chapter.

## *Answers to exercises*

### *Exercise 6.1*

- This is a trickier question than it might first seem. The answer is not 10.0, as you might think (assuming that each of 100 bits produces 0.1 error). Other scenarios exist which could give an even lower error but still not satisfy our conditions for success. Suppose, for example, the network produced a 0.0 on all the outputs. This would yield an error of only 1.0—but our condition (b) would not be satisfied. Clearly, only one output activation need be off by more than 0.1. So the highest level of error that meets the desired criterion for a single pattern is just 0.1.

### *Exercise 6.2*

- Unless you are really lucky, the network will learn Boolean OR when trained on the patterns listed in Table 6.2. When you test the network's generalization to the pattern 1 1, it will most likely respond with an output of 1. You are not providing the network with any evidence for it to believe this is not a linearly separable problem (see Exercise 4.6 on page 92). Under what circumstances might you get lucky and have the network generalize to the novel input 1 1 as if it had learned XOR?

### *Exercise 6.3*

- The partitioning of the Cartesian plane for Continuous XOR is shown in Figure 6.5. This is a more difficult problem than Binary XOR because the network has to be more precise about how it partitions the space. Recall from Chapter 4 that the network solved XOR by moving the 4 points around the Cartesian plane so that they could be partitioned appropriately (see Exercise 4.6 on page 92). Now the network must take more points into consideration.

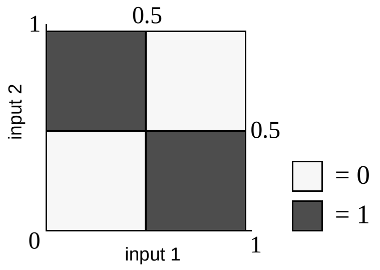


**FIGURE 6.5** Partitioning of the Cartesian plane for Continuous XOR.

***Exercise 6.4***

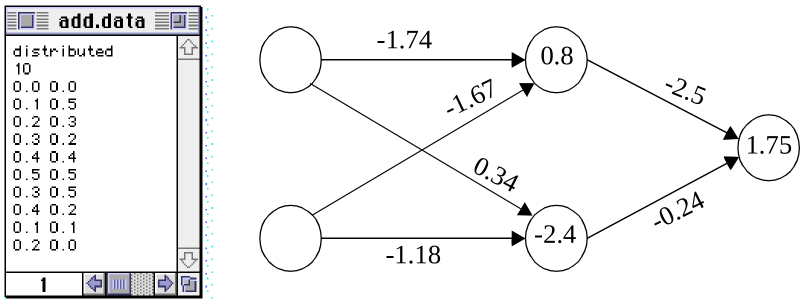
- Unless you were unlucky and chose a set of training parameters that got the network stuck in a local minimum, you should experience no difficulty in training the network to solve this problem. If the network did get stuck in a local minimum, then just try another random seed to initiate the weight matrix.

***Exercise 6.5***

- The network does surprisingly well at adding any legal combination of inputs, just so long as you have constructed a training set that is representative of the problem. For example, the network can learn to perform addition for all 36 input patterns when it has only been trained on 5 of them, if you make sure that the training set spans the full range of outcomes possible, i.e., 0.0 through 1.0. However, if you restrict the problem to inputs below 0.25 you will find that the network becomes increasingly inaccurate as the solutions extend above 0.5.

***Exercise 6.6***

- When the network is trained for 2000 sweeps with the inputs shown in Figure 6.6, with a learning rate of 0.3, momentum of 0.9 and a random seed of 1 (trained randomly without replacement), the final state of the network is as shown in Figure 6.6.



```
add.data
distributed
10
0.0 0.0
0.1 0.5
0.2 0.3
0.3 0.2
0.4 0.4
0.5 0.5
0.3 0.5
0.4 0.2
0.1 0.1
0.2 0.0
```

**FIGURE 6.6** Training set and weight matrix after 2000 sweeps on the addition problem

The network has discovered a rather clever solution to the problem. Notice that the bias on hidden node 2 is quite large and negative. With the negative connection from the second input node, the second hidden node will remain virtually inactive (output close to 0.0) for any input pattern. The second hidden node, therefore, has no effect on the output activation. In contrast, the first hidden node has a small positive bias and almost identical negative connections from the two input nodes. Consider the case when the input is 0 0. The net input to first hidden node is just the bias, i.e. 0.8. This produces an activity of around 0.7 on this hidden node (see Exercise 1.1 on page 8). When this activation is fed through the negative connection to the output node, it exactly cancels out the positive bias to produce an output of 0.0 (remember the output node is linear). Now consider the input 0.5 0.5 which is the largest legal input to which the network is exposed. In this case, the activity propagating up from the input nodes more or less cancels out the positive bias of the first hidden node to produce a net negative input of -0.87. This produces an activity of around 0.3 on the first hidden node. When this activation is fed through the negative connection to the output node, it counteracts the positive bias to produce an output of 1.0. The first hidden node, the connections feeding into and out of it, and the bias on the output node are doing all the work.

How has the network managed to solve the addition problem given that the hidden nodes are nonlinear? We have just seen that we only need to consider the first hidden node. The minimum net input arriving at this node is -0.87 and the maximum net input is 0.8. Now recall the node's sigmoid activation function shown in Figure 1.3 on page 5. When inputs are restricted to this range the sigmoid function is more or less linear. The network has found a solution which exploits the linear component of the sigmoid curve! In other words, it has effectively turned the nonlinear unit into a linear unit by suitable selection of weights and bias.

***Exercise 6.7***

- You should be able to crack this one on your own! However, if you experience difficulties finding a configuration of training parameters for the problem with 2 hidden units try a random seed of 6 (without replacement), a learning rate of 0.3 and a momentum of 0.9.

***Exercise 6-8***

- If the network makes larger errors on some of the input patterns than others, then it probably hasn't learned the appropriate generalization but some other function.

***Exercise 6.9***

1. It is quite easy to think up a solution to this problem using just 2 hidden units. One hidden unit can have a negative bias that keeps it switched off unless there are two active input units. This hidden unit can be connected to the output with a positive connection. The second hidden unit should have a negative bias that keeps it switched off unless there are 3 or more active input units. However, the connection from this unit to the output is large and negative to counteract the activity propagating from

the first hidden unit. Under these circumstances, the output unit will only fire when exactly 2 input units are active. A simple network like this is shown in Figure 6.7.

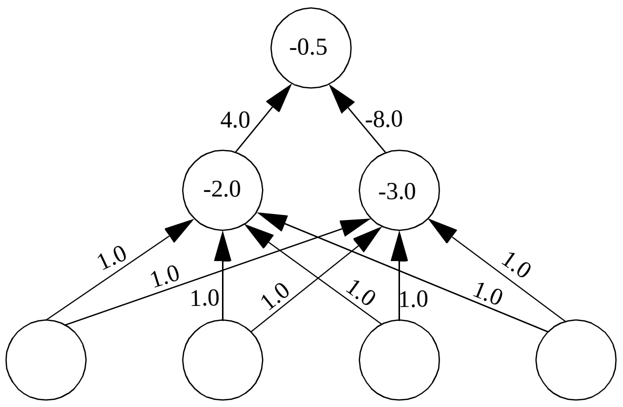


**FIGURE 6.7** A hand-wired network for solving the categorization problem.

2. The solution that your network found by itself might look like that in Figure 6.8. In this example, the first hidden node remains firmly off unless there are at least 3 active input lines. Once this node becomes active, its negative connection to the output node ensures that output is switched off too. The second hidden node has a strong positive bias and remains switched on if there are less than 2 active input lines (note the negative connections between the input nodes and the second hidden node). If the second node is switched on, the strong negative connection to the output node makes sure that the output node is switched off. In other words, the output node is switched off if there are at least 3 or less than 2 active nodes at the input. If there are just 2 active input nodes, the second hidden node switches off but the first hidden node doesn't switch on. In other words, both hidden nodes are dormant allowing the positive bias on the output node to activate it. This solution is quite different to the hand-wired solution in Figure 6.7.

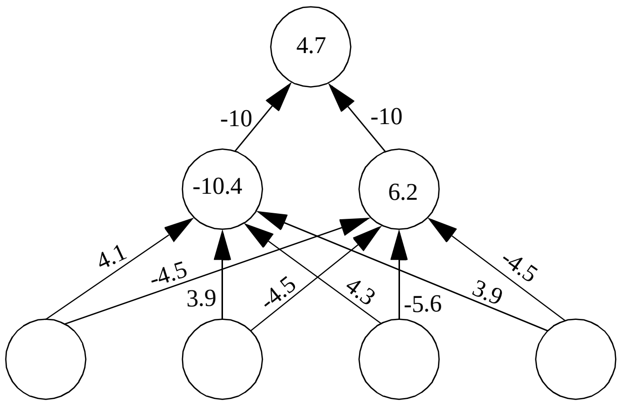


**FIGURE 6.8** A self-organized solution to the categorization problem.

***Exercise 6.10***

- The cluster analysis of the input patterns groups them together as best it can according to their position in the four dimensional input space. So the pattern 1011 is closer to the pattern 1111 than it is to the pattern 0000. This grouping is not relevant to the task that the network is being asked to perform. For example, the network is supposed to group 0011 together with 1100 but these patterns are quite far apart in the four dimensional input space (see Figure 6.4).

***Exercise 6.11***

- The cluster analysis of the hidden unit activations using the network shown in Figure 6.8 is given in Figure 6.9. Now the network has grouped together all the patterns in which just two input nodes are active (the middle branch of the tree). All patterns with just one input node are placed in the first branch of the tree. Patterns with 3 or 4 input nodes active are placed in the third branch of the tree. This organization corresponds exactly to the activities of the hidden nodes that we analyzed in Exercise 6.9.

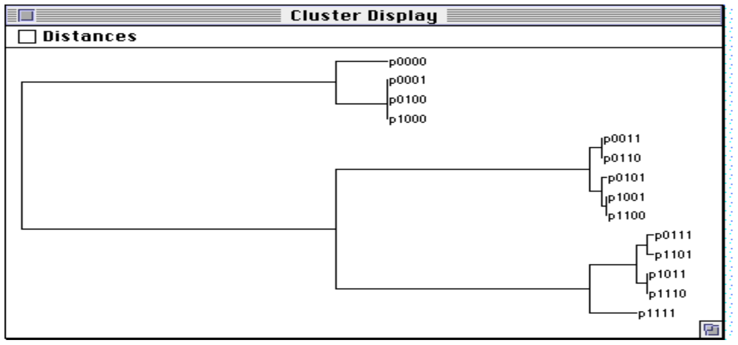


**FIGURE 6.9** Cluster analysis of the hidden unit activations in the generalization problem