CHAPTER 7

Translation invariance

Introduction

In this chapter you will examine the ability of the network to detect patterns which may be displaced in space. For example, you will attempt to train a network to recognise whenever a sequence of three adjacent 1's occurs in a vector, regardless of what other bits are on or where the three 1's are. Thus, `01110000`, `00011100`, `10101111` all contain this pattern, whereas `01100000`, `10101010`, `11011011` do not.

Why might such a problem be of interest? This task is schematic of a larger set of problems which we encounter constantly in everyday life and which are sometimes referred to as examples of *translation invariance*. When we recognize the letter **A** on a page, or identify other common objects regardless of their spatial location, we have solved the problem of perceiving something which has undergone a spatial translation. (We can also usually perceive objects which have been transformed in other ways, such as scaling, but here we address only displacement in space.) We carry out such recognition without apparent effort, and it probably does not even occur to us that a pattern which has been moved in space ought to be particularly difficult to recognize.

In fact, dealing with spatial translations is quite difficult for many visual recognition schemes, and we will find that it is also hard for networks. You will look at one solution, but first you will demonstrate that the problem is hard and try to understand just what the basis of the difficulty is.

In this chapter, you will learn:

- How to configure a neural network so that its hidden nodes have constrained receptive fields (instead of receiving connections from all the units in the previous layer).
- Show that receptive fields and unit groupings are important for solving the problem of translation invariance in a neural network.

Defining the problem

Create a New Project called **`shift`**. Build an **`8x6x1`** network. The training set should contain the following patterns (remember that the vector elements in **`shift.data`** should have spaces between them:

```
11100110 11101001 01110101 01110000
10111011 00111010 11011100 01011101
01101110 10001110 10010111 00100111

10011010 11001101 10000010 01001100
01000000 01101000 00100011 10100100
00110110 10010110 10011000 00010000
00011011 00010001 01010000 01011011
00000100 00001101 00000011 00001001
```

Be sure you have entered these patterns exactly. Check your files!

Of the 32 patterns, the first 12 contain the target string **`111`**, while the last 20 do not. Thus, your **`shift.teach`** file will have 1 as output for the first 12 patterns and 0 for the last 20. Try running the simulation with a learning rate of 0.3 and a momentum of 0.9 for 200 epochs (this means 6400 sweeps). Be sure to choose the Train Randomly option. Feel free to experiment with these parameters. Test the network on the training data.

Exercise 7.1

- Has the network learned the training data? If not, try training for another 200 epochs or run the simulation with a different random seed.

Now test the network's ability to generalize. Create a new **`data`** file containing novel patterns (call it **`novshift.data`**) using the following eight input patterns, four of which contain the target string and four of which do not.

```
00000111 11100100 11101100 01110011
10110001 10001101 11011011 01101101
```

Test the network's response to these novel patterns:

Exercise 7.2

1. How well has the network generalized? Use the clustering procedure you learned in Chapter 6 on the hidden node activation patterns of the training (not test) data.
2. Can you tell from the grouping pattern something about the generalization which the network has inferred?

Relative and absolute position

In the previous simulation you saw that although it is possible to train the network to correctly classify the training stimuli, the network does not generalise in the way you want. Note that this does not mean that the network failed to find some generalisation in the training data, simply that the generalisation was not the one you wanted.

It is important to try to understand why spatial translation is such a difficult problem. The basic fact to be explained is that although bit patterns such as `111000000` and `000111000` look very similar to us, the network sees them as very different. The absolute bit pattern (e.g., whether the first three bits are `0` or `1`) is a more important determinant of similarity than the relative bit pattern (e.g., whether any three adjacent bits are `0` or `1`).

One way to think about why this might be so is to realize that these bit patterns are also vectors, and that they have geometric interpretations. A 9-bit pattern is a vector which picks out a point in 9-

dimensional space; each element in the vector is responsible for locating the point with regard to its own specific dimension. Furthermore, the dimensions are not interchangeable.

Consider a concrete example. Suppose we have a 4-element vector. To make things a bit easier to imagine, we will establish the convention that each position in the vector stands for a different compass point. Thus, the first position stands for North, the second for East, the third for South, and the fourth for West. We will also let the values in the vector be any positive integer. We might now think of the following vector: `1 2 1 1` as instructions to walk North 1 block, then East 2 blocks, South 1 block, and West 1 block. Consider the `4x4` city block map shown in Figure 7.1 and start in the lower left corner and see where you end up.

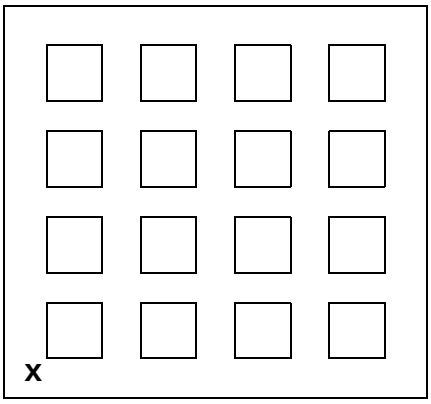

FIGURE 7.1 A geometric interpretation of input vectors

Now let us rotate the numbers in this vector, so that we have something that looks similar (to our eye): `2 1 1 1`. However,—and this is important—we keep the convention that the first number refers to Northern movement, etc. Now we have a different set of instructions. Start again in the lower left corner and see where you end up this time. Not surprisingly, it's a different location. You are probably not surprised because it is obvious that going 2 blocks North and then 1 block East is different from going 1 block North then 2 blocks East.

The situation with the network is similar. Each position in a vector is associated with a different dimension in the vector space; the dimensions are analogous to the compass points. The vector as a whole picks out a unique point.

When the network learns some task, it attempts to group the points which are picked up by the input vectors in some reasonable way which allows it to do the task. (Thinking back to the city-block metaphor, imagine a network making generalizations of the form "all points in the north-west quadrant" versus "points in the south-east quadrant.") Thus, the geometric interpretation of the input vectors is not just a useful fiction; it helps us understand how the network actually solves its problems, which is by spatial grouping.

You can now see why shifts or translations in input vector patterns are so disruptive. You can shift the numbers around, but you cannot shift the interpretation (dimension) that is associated with each absolute position. As a result, the shift yields a vector which "looks" very different to the network (i.e., picks out a different point in space) even though to the human eye the two patterns might seem similar.

These shifts do not necessarily make it impossible for the network to learn a task. After all, in the previous simulation the network succeeded in classifying various instances of shifted **…111…** patterns. XOR is another problem which is successfully learned by the network even though the vectors which have to be grouped are as different as possible (think about the locations of the points picked out in a square by **00**, **11**, **01**, **10**, and the groupings that are necessary for the task). The network can overcome the dissimilarity. (The spatial contortions necessary to do this, however, usually require a hidden layer; the hidden layer takes on the job of reorganizing the spatial structure of the input patterns into a form which facilitates the task.) The point is that the classification solution is not likely to generalize to novel patterns, just because what seemed like the obvious basis for similarity to us (three adjacent **1**'s) was for the network a dissimilarity which had to be ignored.

Again, it is worth thinking about why this problem might worry us. Many human behaviors—particularly those which involve visual perception—involve the perception of patterns which are defined in *relative* terms, and in which the *absolute* location of a pattern in space is irrelevant. Since it is the absolute location of pattern elements which is so salient to the networks you have studied so far, you now want to see if there are any network architectures which do not have

this problem. In the next simulation, you will study an architecture which builds in a sensitivity to the relative form of patterns. (This architecture was first described by Rumelhart, Hinton & Williams, (Chapter 8, PDP Vol. 1) and used in the task of discriminating T from C, regardless of the orientation and position of these characters in a visual array. You may wish to review that section of the chapter before proceeding.)

Receptive fields

Your guiding principle in creating an architecture to solve the shift invariance problem will be this: *Build as much useful structure into the network as possible*. In other words, the more tailored the network is to the problem at hand, the more successful the network is apt to be in devising a solution.

First, you know that the pre-defined target string is exactly three units in length. Therefore, design your network so that each hidden node has a *receptive field* spanning exactly three adjacent input nodes. Hidden nodes will have overlapped receptive fields, staggered by one input. This means that if `111` is present at all in the input, one of the 6 hidden units will receive this as its exclusive input, and the two neighboring hidden units on each side will receive part of the pattern as their input (the hidden unit immediately to the left sees two of the `1`'s, and the unit beyond that sees only one).

Now what about the issue of shift invariance? Each receptive field has a hidden unit serving it exclusively. (For more complicated problems, we might wish to have several hidden units processing input from a receptive field, but for the current problem, a single unit is sufficient.) Let us designate the receptive field weights feeding into a hidden unit as RFW1, RFW2, and RFW3 (Receptive Field Weight 1 connects to the left-most input, RFW2 to the center input, and RFW3 to the right-most input). We have 6 hidden units, each of which has its own RFW1, RFW2, and RFW3. We can require that the RFW1's for all 6 hidden units have *identical* values, that all RFW2's be identical, and all RFW3's be identical. Similarly, the biases for the 6 hidden units will also be constrained to be identical. Thus, `11100000` will activate the hidden node assigned to the first receptive field in exactly

the same way that **01110000** will activate the hidden node assigned to the second receptive field. Finally, since all 6 hidden units are functionally identical, we want the weights from each hidden unit to the output unit to be identical. Figure 7.2 shows the architecture we have just described.

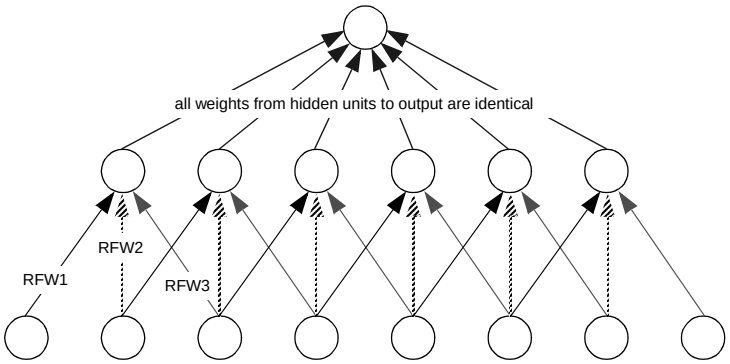

FIGURE 7.2 Network architecture for the translation invariance problem

How can we ensure that this occurs? Our solution will be to initialize each receptive field hidden unit to have an identical set of weights (compared with other units), and then to average together the weight changes computed (according to the backpropagation learning algorithm) for each connection and to adjust each such weight only by the average change for that position. Fortunately, the **tlearn** simulator has an option that will perform the necessary averaging automatically, but it is still necessary to tell the program which weights are to be made identical. To do this, we need to employ the **groups** option in the **.cf** file. All connections in the same group are constrained to be of identical strength.

The **NODES:**, **CONNECTIONS:** and **SPECIAL:** entries for the **.cf** file that you will need for this exercise are shown in Figure 7.3. Notice how each connection or set of connections is identified as belonging to one of 5 groups. The changes made to one weight will be the average of the changes computed by backpropagation for all the

```
NODES:              CONNECTIONS:            SPECIAL:
nodes = 7           groups = 5              selected = 1-6
inputs = 8          1-6 from 0 = group 1    weight_limit = 0.1
outputs = 1         7 from 0
output node is 7    7 from 1-6 = group 2
                    1 from i1 = group 3
                    1 from i2 = group 4
                    1 from i3 = group 5
                    2 from i2 = group 3
                    2 from i3 = group 4
                    2 from i4 = group 5
                    3 from i3 = group 3
                    3 from i4 = group 4
                    3 from i5 = group 5
                    4 from i4 = group 3
                    4 from i5 = group 4
                    4 from i6 = group 5
                    5 from i5 = group 3
                    5 from i6 = group 4
                    5 from i7 = group 5
                    6 from i6 = group 3
                    6 from i7 = group 4
                    6 from i8 = group 5
```

FIGURE 7.3 **`shift2.cf`** file for translation invariance problem.

weights with which it is grouped. (We have formatted this file in three columns; in the real file, the material shown in the **`CONNECTIONS`**: column would immediately follow the **`NODES`**: section, etc.)

Exercise 7.3

1. Draw a diagram of the **`8x6x1`** network, and indicate those weights and biases which are constrained to be identical. Check this with the way that **tlearn** has configured the network.
2. Train the network for 2000 epochs (64,000 sweeps) with a learning rate of 0.3 and momentum of 0.9. (Use random pattern selection.) Has the network learned the training set? If not, try training the network with a different random seed.

The fact that you have successfully trained this new network on the training data does not necessarily imply that the network has learned the translation invariance problem. After all, we saw that the **`8x6x1`** network in the first part of this chapter (the **`shift`** project) also learned the training data; crucially, its failure to generalize in the way we wanted was what told us that it had not extracted the desired regularity. (It's worth pointing out again that the network has undoubtedly generalized to *some* function of the input, but simply not to the one we wished.) Therefore you must test this network with new data.

Exercise 7.4

- When you have successfully trained the network, test its ability to generalize to the novel test patterns. Has the network generalized as desired?

It is possible that on the first attempt, your network may not have generalized correctly (but this is not common); if it fails, retrain with a different starting seed.

Finally, it is worth looking at the network to try to understand its solution. This involves examining the actual weights and drawing the network, with weight values shown, in order to determine what the network's solution is. When you do this, work backwards from the output unit: Ask yourself under what conditions the output unit will be activated (indicating that the target pattern of ...**`111`**... was found). Take into account both the output unit's bias and the activation received from the 6 hidden units. Then ask what input patterns will cause the hidden units to be activated, and what input patterns will cause them to turn off.

Exercise 7.5

- Examine the contents of the weight file. Draw out the weights for one hidden node, the weight connecting it to the output unit, and the biases for the hidden unit and output unit. (These should be identical across different hidden units). Do you understand the network's solution?

Answers to exercises

Exercise 7.1

- It may take a few attempts, but generally this network will succeed in learning the training data after a few attempts. If the network has learned the correctly, the first 12 outputs will be close to 1.0 (but values as low as 0.70 may be acceptable) and the last 20 outputs will be close to 0.0 (again, actual outputs will only approximate 0.0).

Exercise 7.2

1. The first four patterns in **`novshift.data`** all contain the …**`111`**… pattern, whereas the last four do not. If the network has generalized as desired, then the first four outputs will be close to 1.0 and the final four will be close to 0.0. This is not likely to be the case. (It is barely possible that your network, by chance, stumbles on the solution you want. If so, if you run the network another four or five times with different random seeds, you are not likely to replicate this initial success.)

   To do the clustering on the training data hidden unit patterns, you will need to go back to the Network menu, and in the Testing Options… submenu and for the Testing set, select Training set [shift.data]. Then, again in the Network menu, choose Probe selected nodes. This will run the network once more, sending the hidden unit outputs to the Output display window. Delete any extraneous material you have in the Output display and in the File menu, use Save As… to save the hidden unit activations in a new file called **`shift.hidden`**. Before clustering, you will also need to prepare a labels file (called **`shift.lab`**) which is identical to **`shift.data`**, but with the first two (non-pattern) lines removed, and with all spaces deleted. (Since this can be cumbersome, we have already prepared a file with this name and placed in the folder for Chapter 7.) If you now run the Cluster Analysis (found in the Special menu; send output to graphics), you might see something that looks like Figure 7.4:

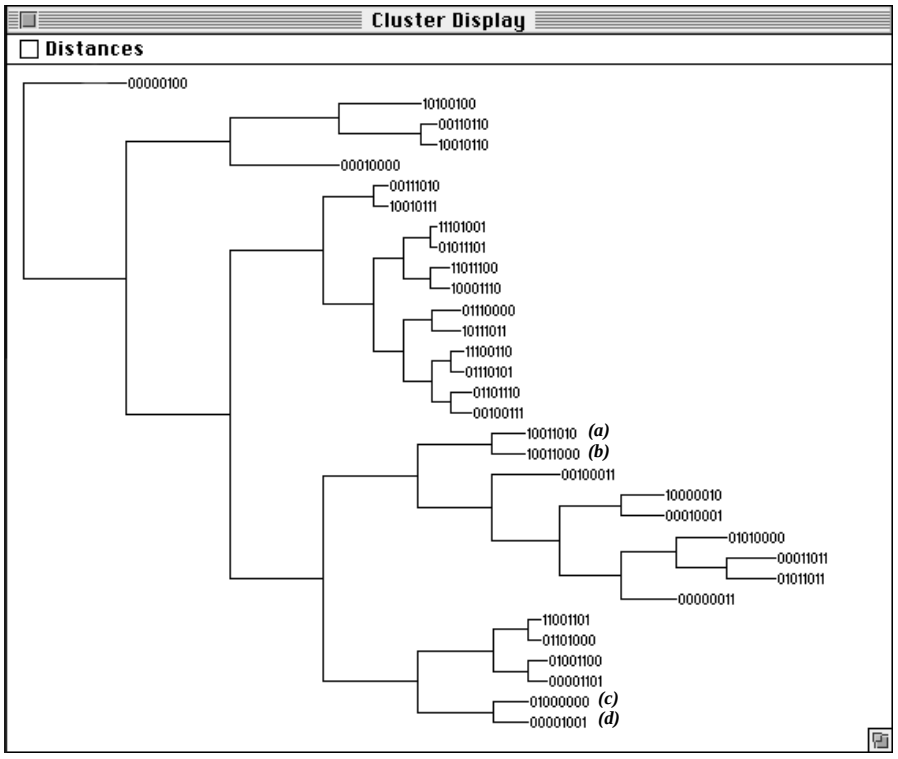

FIGURE 7.4 Cluster analysis of the hidden unit activations on the shift problem

2. Notice that all the patterns which contain …**111**… are clustered together on the same branch; this tells us that the hidden unit patterns produced by these inputs are more similar to each other than to any other inputs. That is what allows the network to treat them the same (i.e., output a **1** when they are input). However, if you look closely, you may also see that the principle by which inputs are grouped appears to have more to do with the degree to which patterns share **1's** and **0's** in the same position. This is particularly apparent for patterns *(a)* and *(b)*, and patterns *(c)* and *(d)*. The fact that inputs which do not contain the …**111**… target pattern also happen to have many **0's** in their initial portions—and that it is this latter feature which the network is picking up on—

should lead us to predict (correctly) that the network would classify the novel test pattern `00000111` on the basis of the initial `0`'s, and ignore the fact that it contains the target.

Exercise 7.3

1. After having drawn your network, display the architecture using the Network Architecture option in the Displays menu. You will see a diagram which looks like that shown in Figure 7.5.

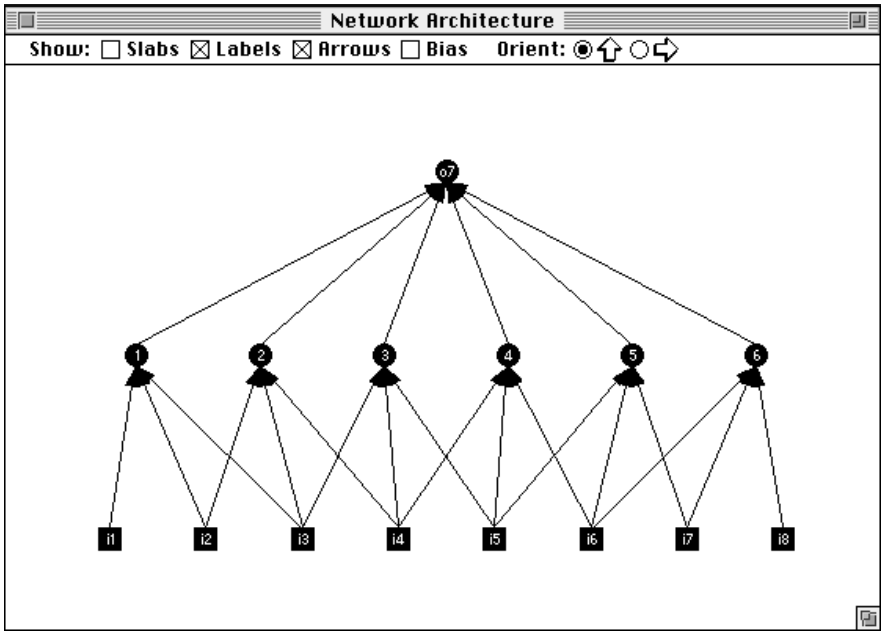

FIGURE 7.5 Translation invariance network architecture

2. We train the network for 2000 epochs simply to ensure that the weights have converged on relatively stable values. This will produce cleaner outputs and make subsequent analysis of the network a bit easier.

Exercise 7.4

- The network should have generalized successfully so that it recognizes the first four patterns in **`novshift.data`** as containing the target (i.e., the network output is close to 1.0), and the last as not containing the target (i.e., the output is close to 0.0). If this is not the case, retrain the network using a different random seed, or experiment with different values for the learning rate and momentum.

  You may find it useful to keep the Error Display active while you are training. If you see that the error does not appear to be declining after a while, you may choose to abort the current training run prematurely and restart with different values. After a while, you may begin to develop a sense of what error plots will ultimately lead to success and which ones are destined to result in failure.

Exercise 7.5

- Figure 7.6 shows the receptive field weights for one hidden unit. (All other input-to-hidden and hidden-to-output weights should be the same.) The biases are shown within each unit.

  Working backwards, we note that the output unit has a strong positive bias. By default, then, it will be on (signalling detection of the target pattern). So we then have to ask the question, What will turn the output unit off? We see that the input which the output unit receives from the 6 hidden units is always inhibitory (due to the weight of -8). However, the output unit's bias is sufficiently large (43) that *all* of the hidden units must be activated in order for their combined effect to be great enough to turn off the output (since -8x5 generates only -40). But if we look at the hidden units' biases, we see that they are strongly positive (14). This means that by default the hidden units *will* be activated. The hidden units' default function is therefore to suppress firing of the output. Overall, the default case is that the output says there is no target present.

  What will cause the output unit to fire, then? If a single hidden unit is turned off, then the remaining hidden units' output will not be sufficient to turn off the output unit and it will fire, indicating detection of the target. So what can turn off a hidden unit? Since the hidden unit bias is 14, and each input weight has an inhibitory weight of -5, all three inputs must be present to turn off a hidden unit, which then releases the output

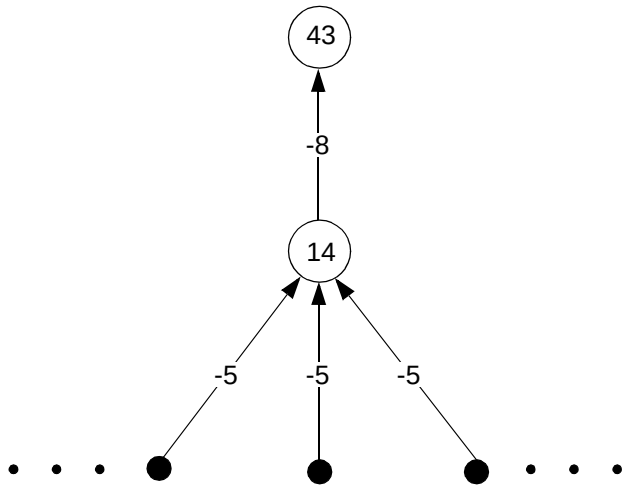

FIGURE 7.6 Receptive field weights for a hidden unit in the translation invariance network

unit from suppression and turns it on. If two or fewer adjacent inputs are present, they will be insufficient to turn off the hidden unit.

This may seem complicated at first, but it actually is a very sensible solution!