CHAPTER 8

Simple recurrent networks

Introduction

In Chapter 7, you trained a network to detect patterns which were displaced in space. Your solution involved a hand-crafted network with constrained weights (so that different hidden nodes could benefit from the learning of others). Now turn your attention to the problem of detecting patterns displaced in *time*.

This problem requires a network architecture with dynamic properties. In this chapter, you'll follow the approach of Elman (1990) which involves the use of *recurrent* connections in order to provide the network with a dynamic memory. Specifically, the hidden node activation pattern at one time step will be fed back to the hidden nodes at the next time step (along with the new input pattern). The internal representations will, thus, reflect task demands in the context of prior internal states. An example of a recurrent network is depicted in Figure 8.1.

In this chapter, you will:

- Train a recurrent network to predict the next letter in a sequence of letters.
- Test how the network generalizes to novel sequences.
- Analyze the network's method of solving the prediction task by examining the error patterns and hidden unit activations.

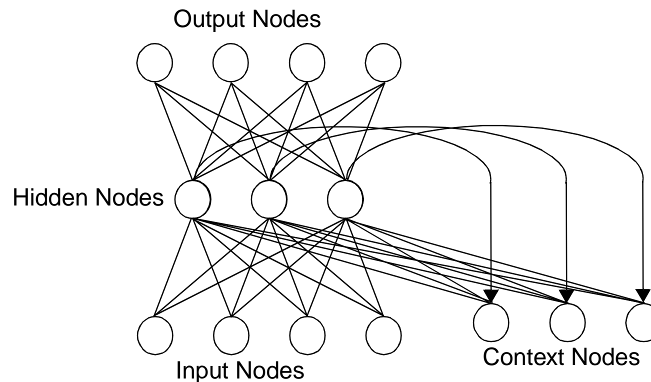


FIGURE 8.1 A simple recurrent network.

File configuration

Starting on the Training Data

Imagine a letter sequence consisting of only three unique strings:

`ba dii guuu`

For example, part of the sequence might look like this:

`babaguuudiiguuubadiidiibaguuuguuu…`

Note that the sequence is only semi-random: the consonants occur randomly, but the identity and number of the following vowels is regular (i.e., whenever a **`d`** occurs, it is always followed by exactly two **`i`**'s). If we trained a dynamic network to predict successive letters in the sequence, the best we could expect would be for the network to say that all three consonants are equally likely to occur in word-initial position; but once a consonant is received, the identity and number of the following vowels should be predicted with certainty.

A file called **`letters`** exists in your **`tlearn`** folder. It contains a random sequence of 1000 words, each 'word' consisting of one of the

3 consonant/vowel combinations depicted above. Open… the **letters** file. Each letter occupies its own line. Translate these letters into a distributed representation suitable for presenting to a network. Create a file called **codes** which contains these lines:

```
b 1 1 0 0
d 1 0 1 0
g 1 0 0 1
a 0 1 0 0
i 0 0 1 0
u 0 0 0 1
```

Now with the **letters** file open and active, select the Translate… option from the Edit menu. The Translate dialogue box will appear as shown in Figure 8.2. Set the Pattern file: box to codes and check that

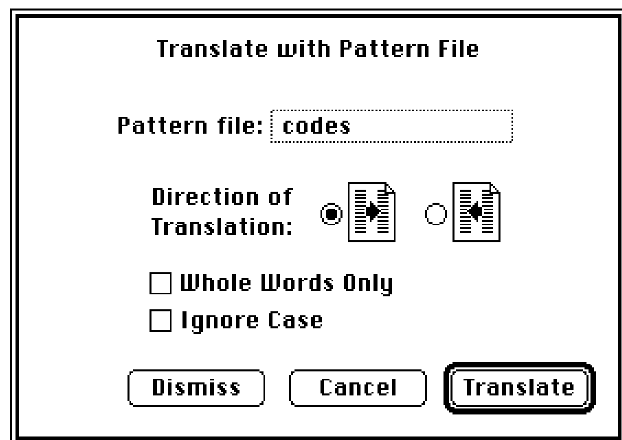
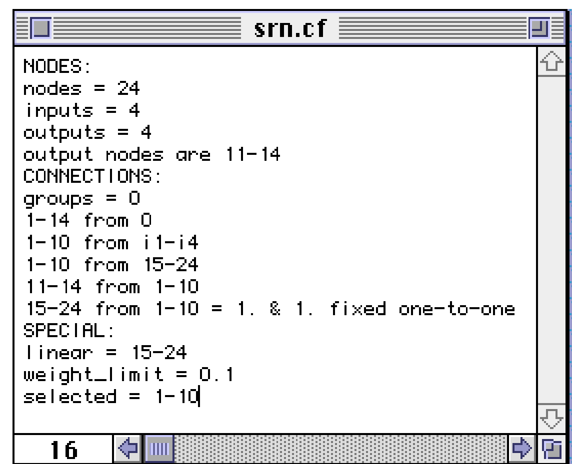


FIGURE 8.2 The translate dialogue box.

the Direction of Translation is from left to right. Then click on Translate. Your **letters** file is translated into rows and columns of binary digits. (Note that both the **letters** and **codes** files must be in the same directory or folder.) Each row consists of one of the sequences of 4 digits taken from the **codes** file. Translate has replaced every letter in the **letters** file with a pattern vector. Every occurrence of a letter in the **letters** file which corresponds with a letter in the first

column of the **codes** file is replaced by the sequence of alphanumeric characters to the right of the first column in the **codes** file[1]. **tlearn** asks you for the name of file in which to save the translated data. To avoid overwriting the existing **letters** file, call it **srn.data**. Next copy this file to a file called **srn.teach** and edit the file, moving the first line to the end of the file. The **srn.teach** file is now one step ahead of the **srn.data** file in the sequence. Complete the **teach** and **data** files by including the appropriate header information.

Now build a **4x10x4** network with 10 context nodes. These special context nodes will store a copy of the hidden node activation pattern at one time step and feed it back to the hidden nodes at the subsequent time step (along with the new input). The **srn.cf** file is shown in Figure 8.3. Notice that the context nodes are specified as nodes **15-24** and are set as **linear** in the **SPECIAL:** section.



```
srn.cf
NODES:
nodes = 24
inputs = 4
outputs = 4
output nodes are 11-14
CONNECTIONS:
groups = 0
1-14 from 0
1-10 from i1-i4
1-10 from 15-24
11-14 from 1-10
15-24 from 1-10 = 1. & 1. fixed one-to-one
SPECIAL:
linear = 15-24
weight_limit = 0.1
selected = 1-10
16
```

FIGURE 8.3 The **srn.cf** file.

Nodes **1-10** receive connections from the context nodes. However, the context nodes also receive connections from nodes **1-10** (last line of the **CONNECTIONS:** section). This line also indicates that these ‘copy-back’ connections have a minimum & maximum value of 1.0,

1. Notice that you could translate your transformed file back again if you wish by using the alternative Direction of Translation in the Translate… dialogue box.

that they are fixed and that they are in a one-to-one relation to each other, i.e., node 1 sends a connection only to node 15, node 2 only sends a connection to node 16, etc.

Exercise 8.1

---

1. Why are the the hidden nodes and the context nodes fully connected in one direction but not in the other? Why do you think the context nodes are set to be linear?
2. Draw a diagram of your network. You may wish to use slabs to indicate layers (rather than drawing individual nodes). Indicate those weights which are fixed at 1.0.

You are now in a position to train the network. However, before you do so you may as well create a set of patterns for testing after you have trained the network.

Exercise 8.2

---

1. What patterns would you include in your test set? Construct the test set and call it **`predtest.data`**.
2. Set the learning rate parameter to 0.1 and momentum to 0.3. Train the network for 70,000 sweeps (since there are 2993 patterns in **`srn.data`**, this is approximately 23 epochs), using the Train sequentially option in the Training Options dialogue box. It is imperative that you train sequentially and not train randomly. Why?
3. To see how the network is progressing, keep track of the RMS error. Why do you think the RMS error is so large?

Exercise 8.2

4. Test the network using the **`predtest.data`** file. How well has the network learned to predict the next element in the sequence? Given a consonant, does it get the vowel identity and number correctly?

5. What does it predict when a consonant is the next element in the stream?

Run through these test patterns again but plot a graph of the error as the network processes the test file. To do this you will need to construct a **`predtest.teach`** file and make sure that the Calculate error box is checked in the Testing options... dialogue box. You should be able to see how the error declines as more of a word is presented. Thus, error should be initially low (as we predict an **`a`** following the first **`b`**, then increases when the **`a`** itself is input and the network attempts to predict the beginning of a new sequence.

Furthermore, if you look at the bit codes that were used to represent the consonants and vowels, you will see that the first bit encodes the C/V distinction; the last three encode the identity of the letter. When individual output bits are interpretable in this way, you might wish to look not only at the overall error for a given pattern (the sum-squared error across all bits in the pattern) but at the errors made on specific bits.

Exercise 8.3

1. What do you notice about the network's ability to predict the occurrence of a vowel versus a consonant as opposed to specific vowels and consonants?

2. Finally, investigate the network's solution by examining the hidden node activation patterns associated with each input pattern. Perform a cluster analysis on the test patterns. Can you infer something from the cluster analysis regarding the network's solution?

Answers to exercises

Exercise 8.1

1. The downward connections from the hidden units to the context units are not like the normal connections we have encountered so far. The purpose of the context units is to preserve a copy of the hidden units' activations at a given time step so that those activations can be made available to the hidden units on the next time step. So what we need, therefore, is a mechanism which will serve to copy, on a one-to-one basis, each of the hidden unit's activations into its corresponding context unit. We do this by setting up downward connections which are one-to-one and fixed (i.e., not changeable through learning) at a value of 1.0. That ensures that the input to each context unit is simply the activation value of its "twin" hidden unit. In addition, we define the context units themselves to have linear activation functions. This means that their activation is simply whatever is input to them, without undergoing the squashing which occurs with units which have (the normal) logistic activation function. The result is that the context unit preserves the value exactly.
2. Your network should look like the one shown in Figure 8.4. We have used slabs to indicate banks of units to avoid clutter. "Distributed" means that all units from one layer are connected to all other units in the second layer; "one-to-one" means that the first unit in one layer is connected only to the first unit in the second layer, etc.

Exercise 8.2

1. What we wish to verify is that the network has learned (a) that each consonant is itself unpredictable, and so should predict all three equally likely, whenever a consonant is expected; and (b) that each consonant predicts a specific vowel will occur a certain number of times. To do this, we need a test set which contains one occurrence of each possible 'word', e.g., the sequence **`b a d i i g u u u`**. As with the training data, you will have to be sure to use the Translate… option from

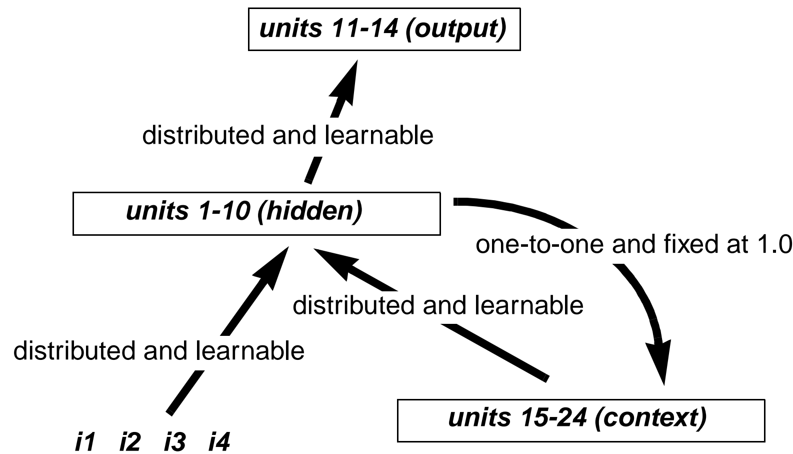


FIGURE 8.4 Schematic of network for the letter prediction task

the Edit menu to convert the letters to vector form. Or, since there are such a small number of patterns involved, you could create the vectors by hand (just be sure not to make mistakes!).

2. The whole point of this chapter is that there are *sequential dependencies* in the data. Obviously, then, if the network is to learn these sequential dependencies, it must experience the data in their correct sequence. Random pattern presentation would mean that a **`d`** might sometimes be followed by an **`i`**, other times by an **`a`**, or **`u`**, or even another consonant. Train sequentially ensures that the network sees the patterns in their correct order, as they appear in the .**`data`** file.

3. In Figure 8.5 we show the error after 70,000 sweeps of training. From the fact that the error has not declined very much and asymptotes at a relatively high level, we might think that the network has not learned much. And in many cases, this would be a reasonable conclusion.

   But there is another possibility to consider before we give up in despair. Remember that the training data are only partially predictable. In fact, there are two things which the network can learn. (1) Once it sees a consonant, the identity and number of occurrences of the vowel that follows can be predicted with certainty; (2) a corollary of knowing

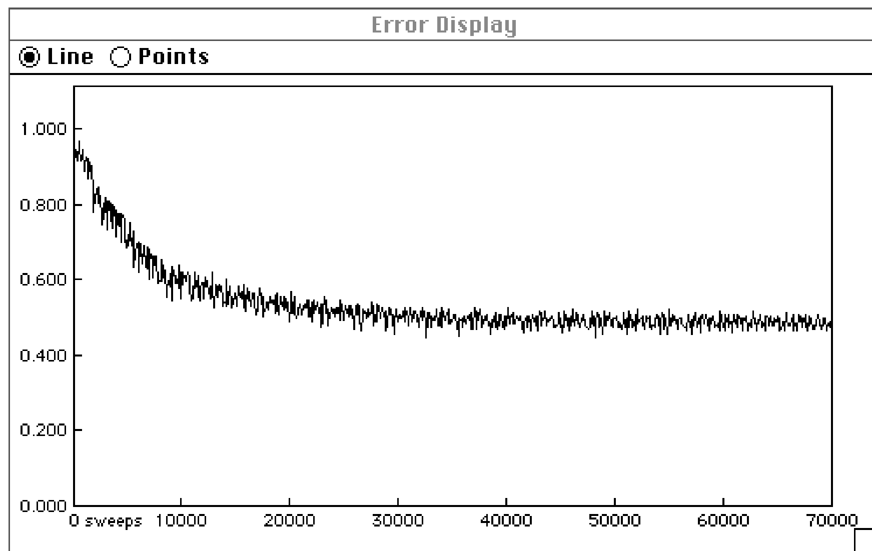


FIGURE 8.5 RMS error for the letter prediction task over 70,000 sweeps

the number of vowels which follows is that the network can know when to expect a consonant. However, the identity of the consonants themselves has been chosen at random. So—unless the network memorizes the training set (which is not likely)—we have given the network a task which inevitably will continue to generate error. Since the error plot averages the error over 100 sweeps and shows a data point only once every 100 sweeps, we are actually combining the error when vowels occur (which should be low) with the error when consonants occur (which should be high). Overall, the average declines as the network learns to get the vowels right; but since it can never predict the consonants, the error can never go to 0.0.

4. When you test the network, the output will be shown in vector form. To interpret this output, you will have to consult the **`codes`** file (shown also on page 153) and work backwards[2]: Find the letter which most closely resembles the vector output from the network. Remember that the network's output is a prediction of what the next letter will be, so the very first line should be similar to the second letter in the **`predtest.data`** file, which is an **`a`**, coded as **`0 1 0 0`**.

5. After the last vowel in a sequence has been input, the network prediction should be wrong—it may look something like one of the consonants, but the precise identity should not be correct.

Exercise 8.3

1. We can define Vowel as any vector with a `0` in the first position, and Consonant as any vector with a `1` in the first position (just because that is the way we set up the vectors in the **`codes`** file to begin with). So now let us look to see what the network is predicting, concentrating only on the first bit position. In one of our own runs, this is what we got (the vowel which should be predicted is shown in parentheses to the left):

(a)	0.000	0.969	0.083	0.004
(d)	0.977	0.345	0.409	0.220
(i)	0.000	0.019	0.985	0.005
(i)	0.006	0.031	0.919	0.039
(g)	0.987	0.444	0.224	0.331
(u)	0.000	0.013	0.010	0.984
(u)	0.001	0.008	0.039	0.979
(u)	0.138	0.099	0.065	0.853
(b)	0.995	0.417	0.512	0.132

Even though the network isn't able to predict *which* consonant to expect, it does clearly know when to expect *some* consonant, as evidenced by a high activation on the first bit.

2. To do a cluster analysis of the hidden unit activations produced by the inputs in **`predtest.data`**, we need first to clear the Output display (if it is open) (in the Edit menu, Select All), and then in the Network menu, Probe selected nodes. This will place the hidden unit activations in the Output display. With that window active, go into the Edit menu and Save As... a file called **`predtest.hid`**. This will be the

2. There is also an Output Translation… utility available in the Special menu and described in Appendix B (page 263 and page 289) which can be used to read output vectors as letters. An example of the use of the Output Translation… utility is described in Chapter 11 on page 212.

vector file we cluster. We will also need a Names file, which we can create by hand and call **`predtest.lab.`** This file should consist of the inputs which produced the hidden unit activations, i.e., the letters **`b a d i i g u u u`**, one per line. Since there are several instances of some of the vowels, we might wish to mark them individually, e.g., **`b a d i1 i2 g u1 u2 u3`** so that we can tell them apart on the plot. Then choose Cluster Analysis from the Special menu, checking Output to Graphics. When we did this, this is the plot we got:

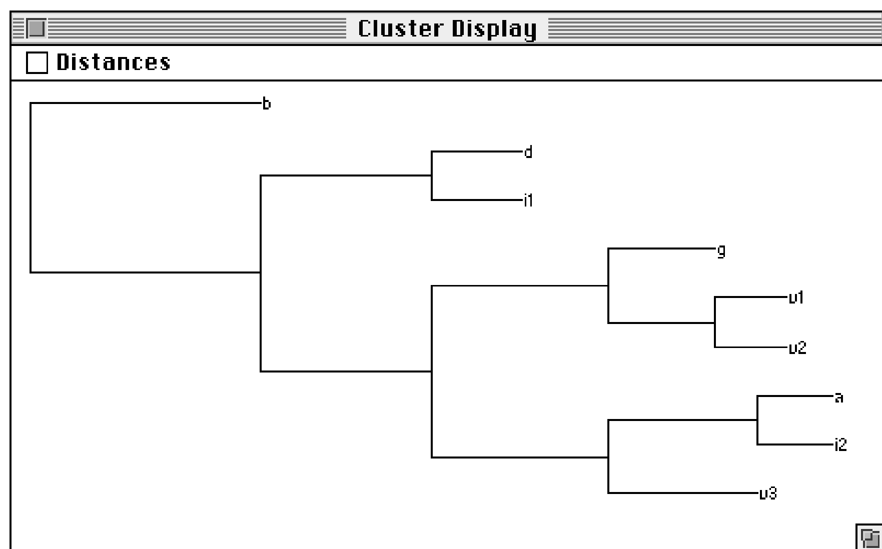


FIGURE 8.6 Cluster analysis of the hidden unit activations for letters in the prediction task

At first, some aspects of this plot may appear a bit odd. For instance, why should the hidden unit patterns when a **`g`** is input be similar to **`u`**? And why should the hidden unit patterns in response to **`a`**, **`i2`**, and **`u3`** be similar to each other (as shown in the bottom branch)?

One thing to remember is that the network's task is *prediction*. We might expect it, therefore, to develop similar hidden unit patterns when it has to predict similar outputs. Viewed this way, in terms of what each of the inputs above *predicts*, the clustering makes much more sense. The **`a`**, **`i2`**, and **`u3`** in the bottom branch all have something very important

in common: They are the final vowels in their subsequence. When the network encounters any **`a`**, it knows that a consonant should follow. Similarly, when the network encounters the second **`i`** or third **`u`** in a sequence, the next letter will be a consonant. In the second branch from the bottom, the **`g`**, **`u1`**, and **`u2`**, all predict a **`u`**. And the **`b`** at the top is different from all other letters because it alone predicts a following **`a`**. So the network's internal representations reflect what it has learned about the classes of inputs with regard to what they predict.