CHAPTER 9

Critical points in learning

Introduction

An intriguing phenomenon in the developmental literature is the observation that many children undergo what appear to be two distinct phases in generalization. In the first stage, they are able to generalize to novel instances, but the generalizations are in some sense restricted to the types of data they have encountered. In the second stage, the generalization ability is unbounded.

For instance, Karmiloff-Smith has noted that children will often undergo a phase where they can demonstrate partial mastery of the principle of commutativity. They know that 2+5 is the same as 5+2, and are able to extend this to other sums which are similar in the sense of involving small numbers (e.g., 7+3, 8+4, etc.). They will fail to recognize, however that 432+85 is the same as 85+432. Thus they are able to generalize a basic notion of commutativity, but the generalizations are limited to sums which may be novel, but which must closely resemble the sums they have encountered before.

At a later stage, of course, children (and adults) recognize that for any arbitrary pair of numbers, their sum is the same regardless of their serial order. At this stage—“true generalization”—the principle of commutativity seems to exist independently of the examples through which it was learned. This is a level of abstraction which seems to be qualitatively different from the knowledge possessed at the earlier stage.

It is not apparent at first glance that a neural network might exhibit behavior which is similar. Neural networks frequently are able to learn functions, and these can then be applied to novel data. *A pri-*

ori, however, one might expect the level of generalization to be a graded and continuous phenomenon which increases smoothly with experience. One would not necessarily expect the sort of two-stage behavior which is observed in children. This is an important issue, because networks use learning devices which operate continuously in time. That is, the learning mechanism does not change. In children, on the other hand, the appearance of dramatic changes in behavior is often taken as evidence for a qualitative change in the underlying learning mechanism (e.g., through reorganization, reanalysis, redescription, etc.).

Surprisingly, such behavior *can* be observed in networks. Networks can undergo qualitative changes in their performance of just the sort which are demonstrated by children who are learning commutativity.

In the following simulation you will explore the generalization ability of a network over the time course of its learning a function. Rather than teaching the network commutativity, you will teach it to tell the difference between 'odd' and 'even' (the parity function, extended over time). You will find that the network shows two kinds of generalization which closely resemble the phases observed in children. Furthermore, you will localize the "magic" point in time where the first type of generalization gives way to the second.

The task

In this task, you will present the network with short sequences of 1's and 0's. Each bit is presented one at a time; the network's task is—at each point in time—to tell you whether an odd or even number of 1's has been input. Sequences will randomly vary in length between 2 and 6. After each sequence is done, the network will be reset (the activations of the context nodes set to 0) and you will start with a new sequence. For example (`/` marks the end of a sequence and beginning of the next):

```
input:   1 1 0 1 / 0 0 / 0 1 0 0 1 0 / 1 1 1 …
output:  1 0 0 1   0 0   0 1 1 1 0 0   1 0 1 …
```

In the **tlearn** folder there are three files called **teach**, **data** and **reset**. **data** contains the input sequence consisting of 1000 randomly generated sequences; **teach** contains the teacher signal for each input; and **reset** contains the list of numbers identifying sequence beginnings (in the example above, the file would have **0, 4, 6, 12,** etc.). When **tlearn** gets to the beginning of a new sequence, it resets all the context units to 0.0, so that the prior state is gone. Edit these files to create the files needed to run a simulation with a recurrent network containing one input node, 3 hidden nodes and 1 output node.

Exercise 9.1

- How many context nodes will the network contain?

Call the project **train1**. You will also need to rename the **reset** file to **train1.reset**. This file will need to be edited so that it contains a header line—an integer specifying the number of time stamps to follow. Each time stamp is an integer specifying the time step at which the context nodes are to be completely reset. The time stamps should appear in ascending order.

Training the network

You will go through several passes of training the network. First, train for about 10 epochs (which means 24880 sweeps). Use a **weight_limit** of 0.1 and select the Use X-entropy; Log RMS option in the Training Options… dialogue box. Set the learning rate and momentum parameters to quite low levels (say, 0.05 and 0.2 respectively). After you have trained the network, you will have a new file, called **train1.24880.wts**.

To test the network, create a data file which contains a single sequence of 100 1's. This is a convenient pattern because:

- It is by far longer than any sequence the network has encountered; you can look to see at what point the network starts to fail.
- It will be easy for you to see visually how the network is doing.

What is cross-entropy, and why use it?

The cross-entropy measure has been used as an alternative to squared error. Cross-entropy can be used as an error measure when a network's output nodes can be thought of as representing independent hypotheses (e.g., each node stands for a different concept), and the node activations can be understood as representing the probability (or confidence) that each hypothesis might be true. In that case, the output vector represents a probability distribution, and our error measure—cross-entropy—indicates the distance between what the network believes this distribution should be, and what the teacher says it should be.

There is a practical reason to use cross-entropy as well. It may be more useful in problems in which the targets are 0 and 1 (though the outputs obviously may assume values in between.) Cross-entropy tends to allow errors to change weights even when nodes saturate (which means their derivatives are asymptotically close to 0).

The correct answer will be an alternating sequence of 1's and 0's (1 0 1 0 1 0), just because the sequence flip/flops back and forth between beyond "odd" and "even."

However, although we can see this output in the Output window, **tlearn** does not currently have the capability of displaying the output graphically; only the error can be graphed (in the Error Display window). We can use the following trick to generate an error display which actually shows what the output looks like. Create a teacher file for the test sequence which consists of 100 1's. This teacher file is not actually the "correct" one (which would be an alternating sequence of 1's and 0's); however, if the network is actually producing the right output (the alternating 1010.... sequence), then the error that will be displaying with this teacher file will itself be an alternating sequence of 0101....[1]

If the network has not learned to generalize at least beyond the longest training sequence (i.e., 6 inputs), repeat the training starting from scratch, but using a different random seed.

1. For example, if the network is performing correctly, its first output is 1; this is compared with the teacher file's first pattern, which is 1, giving an error of 0. If the network's second output is 0, that is compared with the teacher file's second pattern, which is also 1, so the "error" is 1-0=1. Thus, what is displayed is not really the error, but the network's output subtracted from 1, which should—if the network has learned the task—be an alternating sequence of 0101....

Exercise 9.2

1. Create suitable data and teacher files for your test set, as described in the text. Do you need a **`reset`** file for the test sequence?
2. Test the network, plotting the error in the Error Display. Use the trick described in the text (see Footnote 1). How well has the network learned to keep track of odd/even? For how many input bits is it successful?
3. Has it gone beyond the length of the longest training sequence?
4. Is there some point beyond which it fails?
5. How would you characterize the network's generalization performance?

If all has gone well, you will find that a network trained for 10 epochs will generalize its odd/even knowledge to sequences longer than 6, but not much longer. Eventually it gets lost and can't keep track of whether the longer sequence is odd or even. Note that the performance gradually decays: the output activation is more clearly right (close to 1.0 or 0.0) at the beginning, but gradually attenuates. Nonetheless, there is some point at which it just doesn't know.

(It is possible that you may find networks which either fail completely to learn at all, or succeed completely on the 100-item test string. This reflects the ways in which different learning parameters may interact with each other, as well as the initial random weights generated by the seed you use. Catching the network at just the point where it has partially learned may therefore take some time. If you wish, you may use the weights file we provide in your **`tlearn`** folder; this is called **`train1.demo.24880.wts`**. You can use this file for testing purposes by selecting it as the weights file to load with the Testing Options... menu. You can also use these weights as a starting point for continued learning by loading them within the Training Options... menu.)

Now repeat the training procedure from scratch. This time, train for 15 epochs (37320 sweeps). (Again, if you have not gotten good results with your own networks, use the one provided in **`train1.demo.24880.wts`**.)

Exercise 9.3

1. How well has the network learned to keep track of odd/even?
2. Is there some point beyond which it fails?
3. Has it learned to make the odd/even distinction for the entire 100-bit test sequence?
4. How would you characterize the network's generalization performance?

After 15 epochs of training the network should have learned to make the odd/even distinction for the entire 100-bit test sequence. It does so with great confidence: The activation of the output node should look like a saw-tooth wave, swinging sharply between 1 to 0.

An interesting question now is how does the network go from the first stage to the second? Is there a gradual increase in ability? Or is the change abrupt?

In order to discover this, you would have to gradually hone in on the point where the network change occurs. We know already that generalization is incomplete at 10 epochs. We know that it appears to be perfect at 15 epochs. However, to be sure, we should really test the network with a much longer test sequence (e.g., a sequence of 1,000 consecutive 1's). You would then have to (painstakingly) locate the number of epochs where things change by training for different numbers of epochs, carefully picking your tests to locate the change point. Thus, do a series of experiments in which you train for a number of epochs, run your test, and examine the end of the test output sequence. If the output is flat, you know the network has not learned to generalize to 1,000 bits. If the output alternates and achieves stable values, then you might surmise the network has learned to generalize to at least 1,000 bits (and probably, it turns out, forever).

Exercise 9.4

- Whether or not you actually run the extended test in order to localize the “turning point,” what do you think happens at the point where the network shifts from not being able to perform the task for any length string, to the the state where it can? The performance seems to exhibit a qualitatitive change. Is this matched by an underlying change which is better described as being qualitative or quantitative in nature?

Answers to exercises

Exercise 9.1

- Since context units are usually used to hold copies of hidden unit activations, there should be as many context units as hidden units; in this case, that means 3.

Exercise 9.2

1. The point of the test sequence is to test the ability of the network to generalize to a very long string—in this case, a string of 100 1's and 0's. Since the test pattern is one unbroken sequence, we do not need any resets.
2. After training for 10 epochs, test the network with the **`test1.data`** provided in the **`tlearn`** folder for Chapter 9, with the Error Display window visible. You should see something like that shown in Figure 9.1. Remember that we are using the trick (see Footnote 1) of subtracting network output from 1. The correct response would be an alternating sequence of 010.… which, when graphed, would look like a sawtooth wave. We can count any value less than 0.5 as in the right direction when the output should be 1.0 (and, similarly, as correct if the answer should be 0.0 and the output is greater than 0.5). In our simulation, the network's output has this shape for the first 10 inputs. Your simulation result might vary somewhat, but we would expect performance which is roughly comparable to this.

   If you fail to find a network which performs in this way, you may choose to use the weights file we provide in the **`tlearn`** folder, called **`train1.demo.24880.wts`**.
3. Since 10 inputs (speaking now of the first 10 which the network gets right) is a longer sequence then the network has been trained on, the network appears to have generalized beyond the training data.
4. The generalization is limited however. After 10 inputs the network loses track of whether the sequence is odd or even; it goes "brain-dead" on the 11th input.

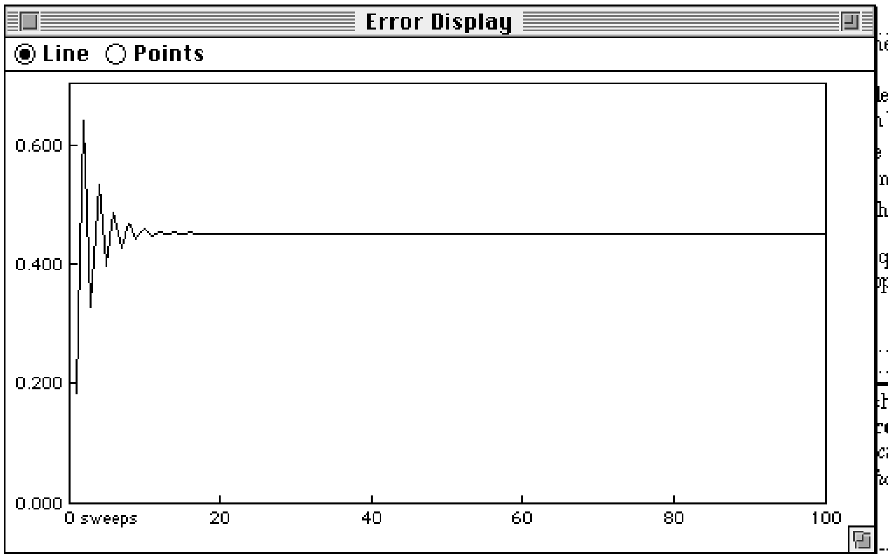

FIGURE 9.1 Error curve for the temporal parity problem when tested after 10 training epochs

5. The fact that the network succeeds in dealing with sequences which are longer than those it has been trained on suggests that it has succeeded in generalizing the odd/even function. There are two interesting respects in which this generalization is limited, however. First, the generalization is partial; the network gives approximately correct responses for longer strings provided the length is not too great. Past some point (in our example, the 11th input), the network fails. Second, if we interpret the magnitude of the output as an indicator of the network's "confidence," then it seems that as the test string increases in length, beyond that seen in training, the network's confidence decreases steadily. Thus, the generalization is not only partial, but graded, diminishing as the input resembles the training examples less and less.

Exercise 9.3

1. With 15 epochs of training the network should correctly respond to all 100 inputs. (If this is not true for you, load in the **`train1.demo.24880.wts`** file we provide, and train for an additional 5 epochs—i.e., another 12,440 sweeps) using our weights file as the starting point.)
2. The network should succeed for the entire 100 input test string. The error we get is shown in Figure 9.2.

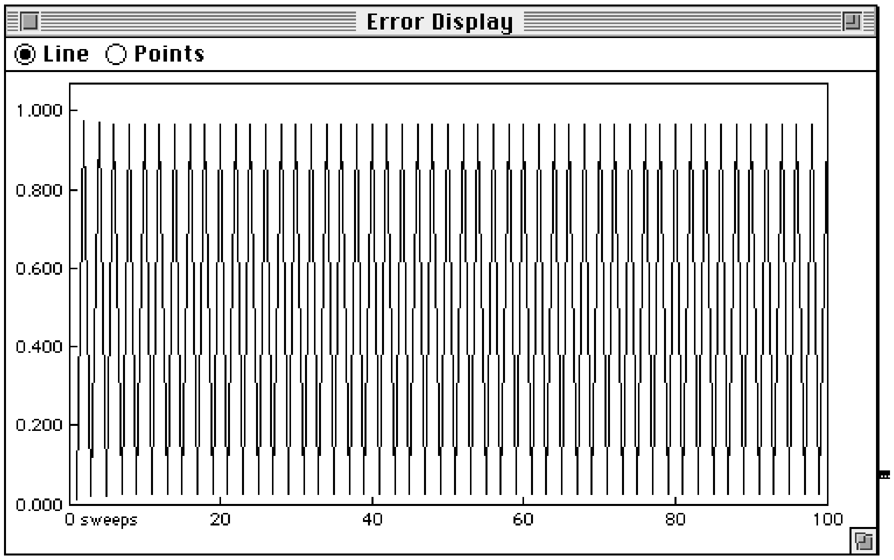

FIGURE 9.2 Error curve for temporal parity when tested after 15 epochs

3. Yes, the network correctly keeps track of the odd/even distinction for the entire test sequence.
4. This network has also learned to generalize, but the generalization appears qualitatively different than when it only has 10 epochs of training. First, the network generalizes well beyond the length of the training data. Second, the network's output does not degrade with increasing length. It appears "confident" of its response throughout the entire sequence. Given this lack of degradation, we might reasonably infer that

the network's generalization is probably absolute: It ought to perform perfectly for strings of indefinite length. (If we really wanted to be sure, however, we ought to test the network at least on strings of length 1,000 or even 100,000.).

Exercise 9.4

- The key to understanding what is happening lies in recognizing two things:

  The first fact is that the network's memory (which is what it's relying on to know whether the current state is "odd" or "even") depends on (1) the current activations of the hidden units, which must somehow encode the current state; and (2) the recurrent connections (context-to-hidden weights), which allow the network to retain information over time (consider a worst case scenario where these connections are 0; then nothing from the past would be remembered). But notice that the current activations of the hidden units themselves are changed by the recurrent weights, which serve as multipliers on the inputs to the hidden units. So the recurrent weights are a critical factor in learning this task.

  Second, remember that the hidden units' activations are a nonlinear function of their inputs. That means that within certain ranges of magnitude, inputs which vary by a great deal may produce hidden unit activations which differ by very little. Within other ranges, however, slight differences in the magnitude of inputs may produce large differences in activation.

  Taken together, these two facts are the beginning to understanding what happens when the network transitions from its limited generalization to absolute generalization. The problem is simply that until weights are learned which are of a magnitude (and the right sign) to ensure that when hidden unit activations are fed back, their values are of a sufficient magnitude to be retained over time.

  We can illustrate this with an example drawn from Chapter 4 of the companion to this handbook, *Rethinking Innateness.* We will use a simpler network to make the issue clearer; you should be able to extrapolate from this example to what is happening with the odd/even network.

  Let us imagine that we have a network with one hidden unit, one input, and one output, as shown in Figure 9.3.

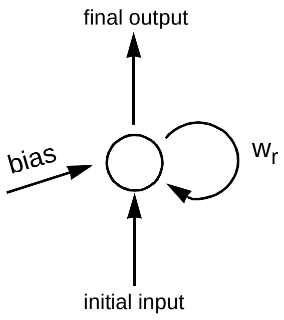

FIGURE 9.3 A one-node network which receives an initial input; the input is then removed, and processing consists of allowing the network to fold its activation back on itself through the recurrent weight. After some number of iterations, we examine the output.

Let us imagine what would happen if the recurrent weight has a value of $w_r = 1.0$ and there is a constant bias of $b = -0.5$ Then if we start with the node having an initial activation of 1.0, on the next cycle the activation will be given by Equation 9.1,

$$a(t+1) = \frac{1}{1+\exp(-(a(t)-0.5))} = \frac{1}{1+\exp^{-0.5}} = 0.62 \qquad \textbf{(EQ 9.1)}$$

or 0.62. If this diminished value is then fed back a second time, the next activation will be 0.53. After 10 iterations, the value is 0.50—and it remains at that level forever. This is the mid-range of the node's activation. It would appear that the network has rapidly lost the initial information that a 1.0 was presented.

This behavior, in which a dynamical system settles into a resting state from which it cannot be moved (absent additional external input) is called a *fixed point.* In this example, we find a fixed point in the middle of the node's activation range. What happens if we change parameters in this one-node network? Does the fixed point go away? Do we have other fixed points?

Let's give the same network a recurrent weight $w_r = 10.0$ and a bias $b = -5.0$. Beginning again with an initial activation of 1.0, we find that now the activation stays close to 1.0, no matter how long we iterate. This makes sense, because we have much larger recurrent weight and so the input to the node is multiplied by a large enough number to counteract the damping of the sigmoidal activation function. This network has a fixed point at 1.0. Interestingly, if we begin with an initial activation of 0.0, we see that also is a fixed point. So too is an initial value of 0.5. If we start with initial node activations at any of these three values, the network will retain those values forever.

What happens if we begin with activations at other values? As we see in Figure 9.4, starting with an initial value of 0.6 results over the next

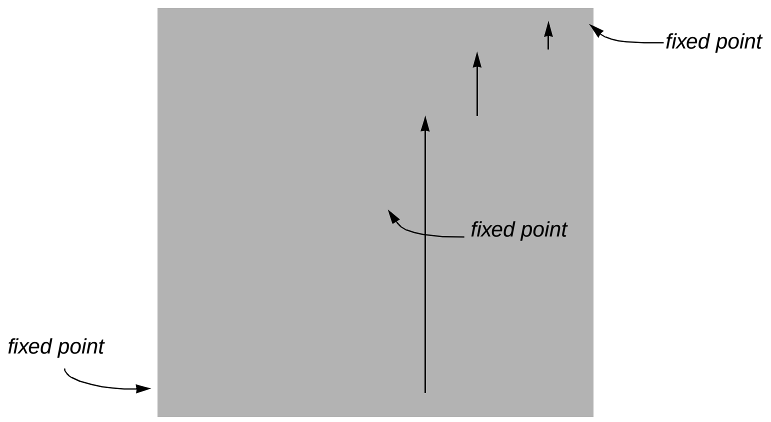

FIGURE 9.4 If a recurrent unit's initial activation value is set to 0.6, after successive iterations the activation will saturate close to 1.0. An initial value of 0.5 will remain constant; an initial value of less than 0.5 will tend to 0.0 (assumes a bias of -5.0 and recurrent weight of 10.0).

successive iterations in an increase in activation (it looks as if the node is "climbing" to its maximum activation value of 1.0). If we had started with a value of 0.4, we would have found successive decreases in activation until the node reached its fixed point close 0.0. Configured in this way, our simple one-node network has three stable fixed points which act as basins of attraction. No matter where the node begins in activation space, it will eventually converge on one of these three activation values.

The critical parameter in this scheme is the recurrent weight (actually, the bias plays a role as well, although we shall not pursue that here). Weights which are too small will fail to preserve a desired value. Weights which are too large might cause the network to move too quickly toward a fixed point. What are good weights?

Working with a network similar to the one shown in Figure 9.3, we can systematically explore the effects of different recurrent weights. We will look to see what happens when a network begins with different initial activation states and is allowed to iterate for 21 cycles, and across a range of different recurrent weights. (This time we'll use negative weights to produce oscillation; but the principle is the same.) Figure 9.5 shows the result of our experiment.

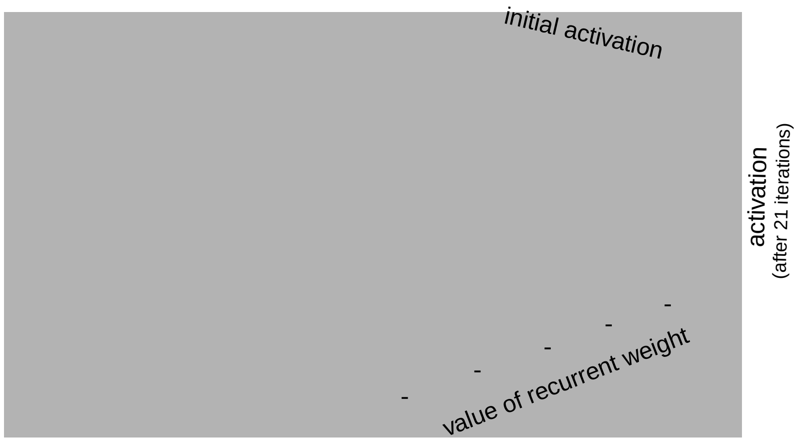

FIGURE 9.5 The surface shows the final activation of the node from the network shown in Figure 9.3 after 21 iterations. Final activations vary, depending on the initial activation (graphed along the width of the plot) and the value of the recurrent weight (graphed along the length of the plot). For weights smaller than approximately -5.0, the final activation is 0.5, regardless of what the initial activation is. For weights greater than -5.0, the final activation is close to 0.0 when the initial activation is above the node's mid-range (0.5); when the initial activation is below 0.5, the final activation is close to 1.0.

Along the base of the plot we have a range of possible recurrent weights, from 0.0 to -10.0. Across the width of the plot we have different initial activations, ranging from 0.0 to 1.0. And along the vertical axis, we plot the final activation after 21 iterations.

This figure shows us that when we have small recurrent weights (below about -5.0), no matter what the initial activation is (along the width of the plot), we end up in the middle of the vertical axis with a resting activation of 0.5. With very large values of weights, however, when our initial activation is greater than 0.5 (the portion of the surface closer to the front), after 21 iterations the final value is 0.0 (because the weight is negative and we iterate an odd number of times, the result is to be switched off; with 22 iterations we'd be back on again). If the initial activation is less than 0.5, after 21 iterations we've reached the 1.0 fixed point state. The important thing to note in the plot, however, is that the transition from the state of affairs where we have a weight which is too small to preserve information to the state where we hold on (and in fact amplify the initial starting activation) is relatively steep. Indeed, there is a very precise weight value which delineates these two regimes. The abruptness is the effect of taking the nonlinear activation function and folding it back on itself many times through network recurrence. This phenomenon (termed a bifurcation) occurs frequently in nonlinear dynamical systems.

To return to the example in this chapter, at the point in training where the network is able to generalize the odd/even solution forever, it has undergone a bifurcation in its weight values. The underlying change involves a process which operates continuously and without sharp transitions; but the effect on behavior is dramatic and appears abrupt.