

---

## Learning in PDP Models: The Pattern Associator

---

In previous chapters we have seen how PDP models can be used as content-addressable memories and constraint-satisfaction mechanisms. PDP models are also of interest because of their learning capabilities. They learn, naturally and incrementally, in the course of processing. In this chapter, we will begin to explore learning in PDP models. We will consider two "classical" procedures for learning: the so-called Hebbian, or correlational learning rule, described by Hebb (1949) and before him by William James (1890), and the error-correcting or "delta" learning rule, as studied in slightly different forms by Widrow and Hoff (1960) and by Rosenblatt (1959).

We will also explore the characteristics of one of the most basic network architectures that has been widely used in distributed memory modeling with the Hebb rule and the delta rule. This is the *pattern associator*. The pattern associator has a set of input units connected to a set of output units by a single layer of modifiable connections that are suitable for training with the Hebb rule and the delta rule. Models of this type have been extensively studied by James Anderson (see Anderson, 1983, for a more recent review), Kohonen (1977), and many others; a number of the papers in the Hinton and Anderson (1981) volume describe models of this type. The models of past-tense learning and of case-role assignment in *PDP:18* and *PDP:19* are pattern associators trained with the delta rule. An analysis of the delta rule in pattern associator models is described in *PDP:11*.

As these works point out, one-layer pattern associators have several suggestive properties that have made them attractive as models of learning and memory. They can learn to act as content-addressable memories; they generalize the responses they make to novel inputs that are similar to the inputs that they have been trained on; they learn to extract the prototype of a set of repeated experiences in ways that are very similar to the concept-learning characteristics seen in human cognitive processes; and they

degrade gracefully with damage and noise. In this chapter our aim is to help you develop a basic understanding of the characteristics of these simple parallel networks. However, it must be noted that these kinds of networks have limitations. In the next chapter we will examine these limitations and consider learning procedures that allow the same positive characteristics of pattern associators to manifest themselves in networks and overcome one important class of limitations.

We begin this chapter by presenting a basic description of the learning rules and how they work in training connections coming into a single unit. We will then apply them to learning in the pattern associator.

## BACKGROUND

### The Hebb Rule

In Hebb's own formulation, this learning rule was somewhat vaguely described. He suggested that when two cells fire at the same time, the strength of the connection between them should be increased. There are a variety of mathematical formulations of this principle that may be given. The simplest by far is the following:

$$\Delta w_{ij} = \epsilon a_i a_j. \quad (1)$$

Here we use  $\epsilon$  to refer to the value of the learning rate parameter. This version has been used extensively in the early work of James Anderson (e.g., Anderson, 1977). If we start from all-zero weights, then expose the network to a sequence of learning events indexed by  $l$ , the value of any weight at the end of a series of learning events will be

$$w_{ij} = \epsilon \sum_l a_{il} a_{jl}. \quad (2)$$

In studying this rule, we will assume that activations are distributed around 0 and that the units in the network have activations that can be set in either of two ways: They may be *clamped* to particular values by external inputs or they may be determined by inputs via their connections to other units in the network. In the latter case, we will initially focus on the case where the units are completely linear; that is, on the case in which the activation and the output of the unit are simply set equal to the net input:

$$a_i = \sum_j a_j w_{ij}. \quad (3)$$

In this formulation, with the activations distributed around 0, the  $w_{ij}$  assigned by Equation 2 will be proportional to the correlation between the

activations of units  $i$  and  $j$ ; normalizations can be used to preserve this correlational property when units have mean activations that vary from 0.

The correlational character of the Hebbian learning rule is at once the strength of the procedure and its weakness. It is a strength because these correlations can sometimes produce useful associative learning; that is, particular units, when active, will tend to excite other units whose activations have been correlated with them in the past. It can be a weakness, though, since correlations between unit activations often are not sufficient to allow a network to learn even very simple associations between patterns of activation.

First let's examine a positive case: a simple network consisting of two input units and one output unit (Figure 1A). Suppose that we arrange things so that by means of inputs external to this network we are able to impose patterns of activation on these units, and suppose that we use the Hebb rule (Equation 1 above) to train the connections from the two input units to the output unit. Suppose further that we use the four patterns shown in Figure 1B; that is, we present each pattern, forcing the units to

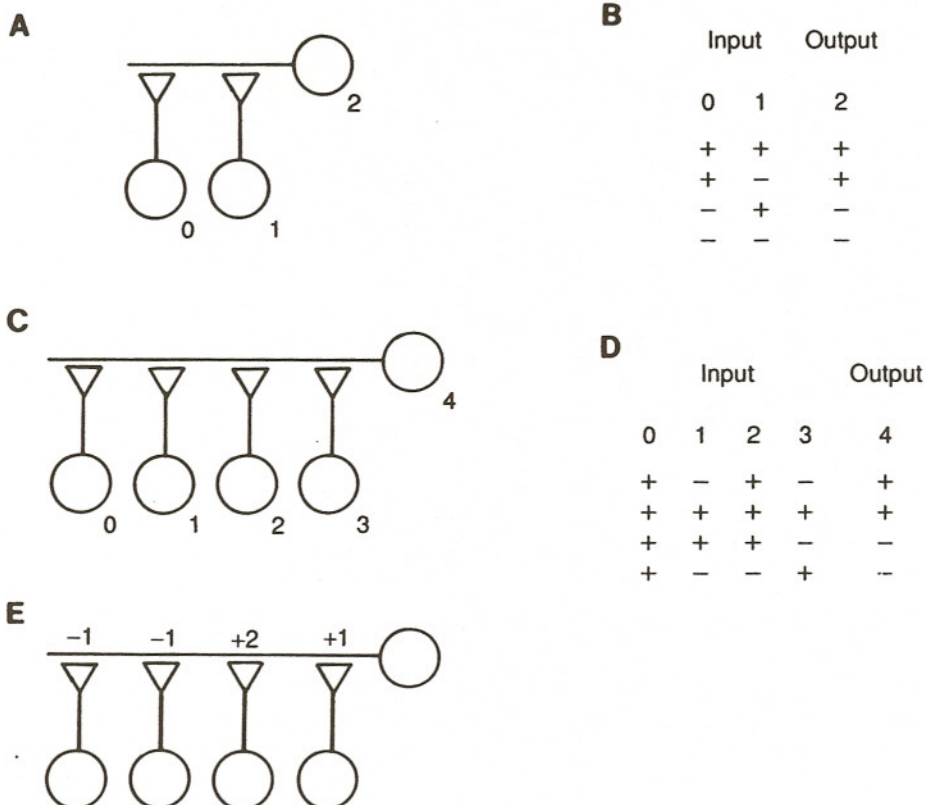


FIGURE 1. Two simple associative networks and the patterns used in training them.

the correct activation, then we adjust the strengths of the connections between the units. According to Equation 1,  $w_{20}$  (the weight on the connection to unit 2 from unit 0) will be increased in strength for each pattern by amount  $\epsilon$ , which in this case we will set to 1.0. On the other hand,  $w_{21}$  will be increased by amount  $\epsilon$  in one of the cases (pattern 0) and reduced by  $\epsilon$  in the other case, for a net change of 0.

As a result of this training, then, this simple network would have acquired a positive connection weight to unit 2 from unit 0. This connection will now allow unit 0 to make unit 2 take on an activation value correlated with that of unit 0. At the same time, the network would have acquired a null connection from unit 1 to unit 2, capturing the fact that the activation of unit 1 has no predictive relation to the activation of unit 2. In this way, it is possible to use Hebbian learning to learn associations that depend on the correlation between activations of units in a network.

Unfortunately, the correlational learning that is possible with a Hebbian learning rule is a "unitwise" correlation, and sometimes, these unitwise correlations are not sufficient to learn correct associations between whole input patterns and appropriate responses. To see that this is so, suppose we change our network so that there are now four input units and one output unit, as shown in Figure 1C. And suppose we want to train the connections in the network so that the output unit takes on the values given in Figure 1D for each of the four input patterns shown there. In this case, the Hebbian learning procedure will not produce correct results. To see why, we need to examine the values of the weights (equivalently, the pairwise correlations of the activations of each sending unit with the receiving unit). What we see is that three of the connections end up with 0 weights because the activation of the corresponding input unit is uncorrelated with the activation of the output unit. Only one of the input units, unit 2, has a positive correlation with unit 4 over this set of patterns. This means that the output unit will make the same response to the first three patterns since in all three of these cases the third unit is on, and this is the only unit with a nonzero connection to the output unit.

Before leaving this example, we should note that there are values of the connection strengths that will do the job. One such set is shown in Figure 1E. The reader can check that this set produces the correct results for each of the four input patterns by using Equation 3.

Apparently, then, successful learning requires finding connection strengths that are not proportional to the correlations of activations of the units. How can this be done?

## The Delta Rule

One answer that has occurred to many people over the years is the idea of using the difference between the desired, or *target*, activation and the

obtained activation to drive learning. The idea is to adjust the strengths of the connections so that they will tend to reduce this *difference* or *error* measure. Because the rule is driven by differences, we have tended to call it the delta rule. Others have called it the Widrow-Hoff learning rule or the least mean square (LMS) rule (Widrow & Hoff, 1960); it is related to the perceptron convergence procedure of Rosenblatt (1959).

This learning rule, in its simplest form, can be written

$$\Delta w_{ij} = \epsilon e_i a_j \quad (4)$$

where  $e_i$ , the error for unit  $i$ , is given by

$$e_i = t_i - a_i, \quad (5)$$

the difference between the teaching input to unit  $i$  and its obtained activation.

To see how this rule works, let's use it to train the five-unit network in Figure 1C on the patterns in Figure 1D. The training regime is a little different here: For each pattern, we turn the input units on, then we see what effect they have on the output unit; its activation reflects the effects of the current connections in the network. (As before we assume the units are linear.) We compute the difference between the obtained output and the teaching input (Equation 5). Then, we adjust the strengths of the connections according to Equation 4. We will follow this procedure as we cycle through the four patterns several times, and look at the resulting strengths of the connections as we go. The network is started with initial weights of 0. The results of this process for the first cycle through all four patterns are shown in the first four rows of Figure 2.

The first time pattern 0 is presented, the response (that is, the obtained activation of the output unit) is 0, so the error is +1. This means that the changes in the weights are proportional to the activations of the input units. A value of 0.25 was used for the learning rate parameter, so each  $\Delta w$  is  $\pm 0.25$ . These are added to the existing weights (which are 0), so the resulting weights are equal to these initial increments. When pattern 1 is presented, it happens to be uncorrelated with pattern 0, and so again the obtained output is 0. (The output is obtained by summing up the pairwise products of the inputs on the current trial with the weights obtained at the end of the preceding trial.) Again the error is +1, and since all the input units are on in this case, the change in the weight is +0.25 for each input. When these increments are added to the original weights, the result is a value of +0.5 for  $w_{04}$  and  $w_{24}$ , and 0 for the other weights. When the next pattern is presented, these weights produce an output of +1. The error is therefore -2, and so relatively larger  $\Delta w$  terms result. Even so, when the final pattern is presented, it produces an output of +1 as well. When the weights are adjusted to take this into account, the weight from input unit 0 is negative and the weight from unit 2 is positive; the other weights are 0. This completes the first sweep through the set of patterns. At this point,

Ep	Pat	Input	Tgt	Output	Error	Delta w's	New values of w's
0	0	1 -1 1 -1	1	0.00	1.00	0.25-0.25 0.25-0.25	0.25-0.25 0.25-0.25
0	1	1 1 1 1	1	0.00	1.00	0.25 0.25 0.25 0.25	0.50 0.00 0.50 0.00
0	2	1 1 1 -1	-1	1.00	-2.00	-0.50-0.50-0.50 0.50	0.00-0.50 0.00 0.50
0	3	1 -1 -1 1	1	1.00	-2.00	-0.50 0.50 0.50-0.50	-0.50 0.00 0.50 0.00
tss: 10.00							
1	0	1 -1 1 -1	1	0.00	1.00	0.25-0.25 0.25-0.25	-0.25-0.25 0.75-0.25
1	1	1 1 1 1	1	0.00	1.00	0.25 0.25 0.25 0.25	0.00 0.00 1.00 0.00
1	2	1 1 1 -1	-1	1.00	-2.00	-0.50-0.50-0.50 0.50	-0.50-0.50 0.50 0.50
1	3	1 -1 -1 1	1	0.00	-1.00	-0.25 0.25 0.25-0.25	-0.75-0.25 0.75 0.25
tss: 7.00							
...							
3	0	1 -1 1 -1	1	0.25	0.75	0.19-0.19 0.19-0.19	-0.63-0.63 1.25 0.25
3	1	1 1 1 1	1	0.25	0.75	0.19 0.19 0.19 0.19	-0.44-0.44 1.44 0.44
3	2	1 1 1 -1	-1	0.13	-1.13	-0.28-0.28-0.28 0.28	-0.72-0.72 1.16 0.72
3	3	1 -1 -1 1	1	-0.44	-0.56	-0.14 0.14 0.14-0.14	-0.86-0.58 1.30 0.58
tss: 1.52							
...							
10	0	1 -1 1 -1	1	0.90	0.10	0.03-0.03 0.03-0.03	-0.95-0.95 1.90 0.90
10	1	1 1 1 1	1	0.90	0.10	0.03 0.03 0.03 0.03	-0.92-0.92 1.92 0.92
10	2	1 1 1 -1	-1	-0.85	-0.15	-0.04-0.04-0.04 0.04	-0.96-0.96 1.89 0.96
10	3	1 -1 -1 1	1	-0.92	-0.08	-0.02 0.02 0.02-0.02	-0.98-0.94 1.91 0.94
tss: 0.05							
...							
20	0	1 -1 1 -1	1	0.99	0.01	0.00-0.00 0.00-0.00	-1.00-1.00 1.99 0.99
20	1	1 1 1 1	1	0.99	0.01	0.00 0.00 0.00 0.00	-1.00-1.00 2.00 1.00
20	2	1 1 1 -1	-1	-0.99	-0.01	-0.00-0.00-0.00 0.00	-1.00-1.00 1.99 1.00
20	3	1 -1 -1 1	1	-1.00	-0.00	-0.00 0.00 0.00-0.00	-1.00-1.00 1.99 1.00
tss: 0.00							

FIGURE 2. Learning with the delta rule. See text for explanation.

the values of the weights are far from perfect; if we froze them at these values, the network would produce 0 output to the first three patterns. It would produce the correct answer (an output of  $-1$ ) only for the last pattern.

The correct set of weights is approached asymptotically if the training procedure is continued for several more sweeps through the set of patterns. Each of these sweeps, or *training epochs*, as we will call them henceforth, results in a set of weights that is closer to a perfect solution. To get a measure of the closeness of the approximation to a perfect solution, we can calculate an error measure for each pattern as that pattern is being processed. For each pattern, the error measure is the value of the error  $(t - a)$  squared. This measure is then summed over all patterns to get a *total sum of squares* or *tss* measure. The resulting error measure, shown for each of the illustrated epochs in Figure 2, gets smaller over epochs, as do the changes in the strengths of the connections. The weights that result at

the end of 20 epochs of training are very close to the perfect solution values. With more training, the weights converge to these values.

The error-correcting learning rule, then, is much more powerful than the Hebb rule. In fact, it can be proven rather easily that the error-correcting rule will find a set of weights that drives the error as close to 0 as we want for each and every pattern in the training set, provided such a set of weights exists. Many proofs of this theorem have been given; a particularly clear one may be found in Minsky and Papert (1969).

### The Linear Predictability Constraint

We have just noted that the delta rule will find a set of weights that solves a network learning problem, provided such a set of weights exists. What are the conditions under which such a set actually does exist?

Such a set of weights exists only if for each input-pattern-target-pair the target can be predicted from a weighted sum, or *linear combination*, of the activations of the input units. That is, the set of weights must satisfy

$$t_{ip} = \sum_j w_{ij} a_{jp} \quad (6)$$

for output unit  $i$  in all patterns  $p$ .

This constraint (which we called the *linear predictability constraint* in *PDP:17*) can be overcome by the use of hidden units, but hidden units cannot be trained using the delta rule as we have described it here because (by definition) there is no teacher for them. Procedures for training such units are discussed in Chapter 5.

Up to this point, we have considered the use of the Hebb rule and the delta rule for training connections coming into a single unit. We now consider how these learning rules produce the characteristics of *pattern associator* networks.

## THE PATTERN ASSOCIATOR

In a pattern associator, there are two sets of units: input units and output units. There is also a matrix representing the connections from the input units to the output units. A pattern associator is really just an extension of the simple networks we have been considering up to now, in which the number of output units is greater than one and each input unit has a connection to each output unit. An example of an eight-unit by eight-unit pattern associator is shown in Figure 3.

The pattern associator is a device that learns associations between input patterns and output patterns. It is interesting because what it learns about

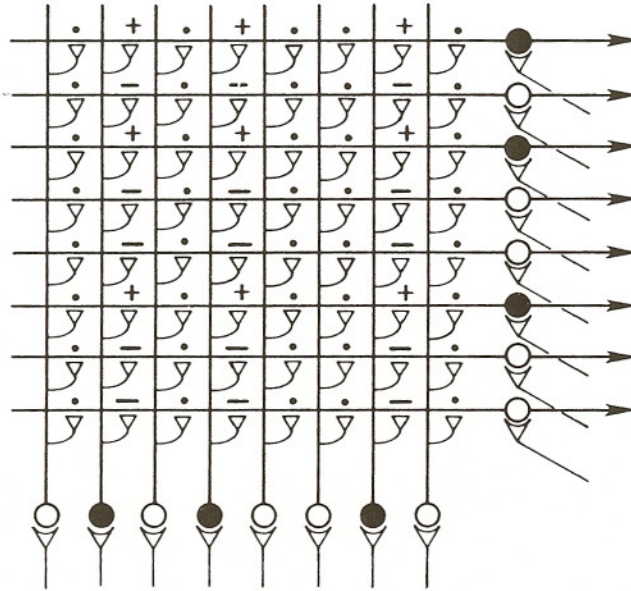


FIGURE 3. A schematic diagram of an eight-unit pattern associator. An input pattern, an output pattern, and values for the weights that will allow the input to produce the output are shown. (From *PDP:18*, p. 227.)

one pattern tends to generalize to other similar patterns. In what follows we will see how this property arises, first in the simplest possible pattern associator—a pattern associator consisting of linear units, trained by the Hebb rule.<sup>1</sup>

### The Hebb Rule in Pattern Associator Models

To begin, let us consider the effects of training a network with a single learning trial  $l$ , involving an input pattern  $\mathbf{i}_l$  and an output pattern  $\mathbf{o}_l$ . Assuming all the weights in the network are initially 0, we can express the value of each weight as

$$w_{ij} = \epsilon i_{jl} o_{il}. \tag{7}$$

Note that we are using the variable  $i_{jl}$  to stand for the activation of input unit  $j$  in input pattern  $\mathbf{i}_l$ , and we are using  $o_{il}$  to stand for the activation of output unit  $i$  in output pattern  $\mathbf{o}_l$ . Thus, each weight is just the product of

<sup>1</sup> Readers who wish to gain a better grasp on the mathematical basis of this class of models may find it worthwhile to read *PDP:9*. An in-depth analysis of the delta rule in pattern associators is in *PDP:11*.



the activation of the input unit times the activation of the output unit in the learning trial  $l$ .

Now let us present a test input pattern,  $\mathbf{i}_t$ , and examine the resulting output pattern it produces. Since the units are linear, the activation of output unit  $i$  when tested with input pattern  $\mathbf{i}_t$  is

$$o_{it} = \sum_j w_{ij} i_{jt}. \quad (8)$$

Substituting for  $w_{ij}$  from Equation 7 yields

$$o_{it} = \sum_j \epsilon i_{jl} o_{il} i_{jt}. \quad (9)$$

Since we are summing with respect to  $j$  in this last equation, we can pull out  $\epsilon$  and  $o_{il}$ :

$$o_{it} = \epsilon o_{il} \sum_j i_{jl} i_{jt}. \quad (10)$$

Equation 10 says that the output at the time of test will be proportional to the output at the time of learning times the sum of the elements of the input pattern at the time of learning, each multiplied by the corresponding element of the input pattern at the time of test.

This sum of products of corresponding elements is called the *dot product*. It is very important to our analysis because it expresses the *similarity* of the two patterns  $\mathbf{i}_l$  and  $\mathbf{i}_t$ . It is worth noting that we have already encountered an expression similar to this one in Equation 2. In that case, though, the quantity was proportional to the correlation of the activations of two *units* across an ensemble of *patterns*. Here, it is proportional to the correlation of two *patterns* across an ensemble of *units*. It is often convenient to normalize the dot product by taking out the effects of the number of elements in the vectors in question by dividing the dot product by the number of elements. We will call this quantity the *normalized dot product*. For patterns consisting of all +1s and -1s, it corresponds to the correlation between the two patterns. The normalized dot product has a value of 1 if the patterns are identical, a value of -1 if they are exactly opposite to each other, and a value of 0 if the elements of one vector are completely uncorrelated with the elements of the other.

We can rewrite Equation 10, then, replacing the summed quantity by the normalized dot product of input pattern  $\mathbf{i}_l$  and input pattern  $\mathbf{i}_t$ , which we denote by  $(\mathbf{i}_l \cdot \mathbf{i}_t)_n$ :

$$o_{it} = k o_{il} (\mathbf{i}_l \cdot \mathbf{i}_t)_n \quad (11)$$

where  $k = n\epsilon$  ( $n$  is the number of units). Since Equation 11 applies to all of the elements of the output pattern  $\mathbf{o}_t$ , we can write

$$\mathbf{o}_t = k \mathbf{o}_l (\mathbf{i}_l \cdot \mathbf{i}_t)_n. \quad (12)$$

This result is very basic to thinking in terms of patterns since it demonstrates that what is crucial for the performance of the network is the similarity relations among the input patterns—their correlations—rather than their specific properties considered as individuals.<sup>2</sup> Thus Equation 12 says that the output pattern produced by our network at test is a scaled version of the pattern stored on the learning trial. The magnitude of the pattern is proportional to the similarity of the learning and test patterns. In particular, if  $k = 1$  and if the test pattern is identical to the training pattern, then the output at test will be identical to the output at learning.

An interesting special case occurs when the normalized dot product between the learned pattern and the test pattern is 0. In this case, the output is 0: There is no response whatever. Patterns that have this property are called *orthogonal* or *uncorrelated*; note that this is not the same as being *opposite* or *anticorrelated*.

To develop intuitions about orthogonality, you should compute the normalized dot products of each of the patterns  $b$ ,  $c$ ,  $d$ , and  $e$  below with pattern  $a$ :

$a$  ++--  
 $b$  +-+-  
 $c$  +--+  
 $d$  ++++  
 $e$  --++

You will see that patterns  $b$ ,  $c$ , and  $d$  are all orthogonal to pattern  $a$ ; in fact, they are all orthogonal to each other. Pattern  $e$ , on the other hand, is not orthogonal to pattern  $a$ , but is anticorrelated with it. Interestingly, it forms an orthogonal set with patterns  $b$ ,  $c$ , and  $d$ . When all the members of a set of patterns are orthogonal to each other, we call them an *orthogonal set*.

Now let us consider what happens when an entire ensemble of patterns is presented during learning. In the Hebbian learning situation, the set of weights resulting from an ensemble of patterns is just the sum of the sets of weights resulting from each individual pattern. That is, after learning trials on each of a set of input patterns  $\mathbf{i}_l$ , each paired with an output pattern  $\mathbf{o}_l$ , the value of each weight will be

$$w_{ij} = \epsilon \sum_l i_{jl} o_{il}. \quad (13)$$

Thus, the output produced by each test pattern is

$$\mathbf{o}_t = k \sum_l (\mathbf{i}_t \cdot \mathbf{i}_l) \mathbf{o}_l. \quad (14)$$

<sup>2</sup> Technically, performance depends on the similarity relations among the patterns and on their overall strength or magnitude. However, among vectors of equal strength (e.g., the vectors consisting of all +1s and -1s), only the similarity relations are important.

In words, the output of the network in response to input pattern  $t$  is the sum of the output patterns that occurred during learning, with each pattern's contribution weighted by the similarity of the corresponding input pattern to the test pattern. Three important facts follow from this:

1. If a test input pattern is orthogonal to all training input patterns, the output of the network will be 0; there will be no response to an input pattern that is completely orthogonal to all of the input patterns that occurred during learning.
2. If a test input pattern is similar to one of the learned input patterns and is uncorrelated with all the others, then the test output will be a scaled version of the output pattern that was paired with the similar input pattern during learning. The magnitude of the output will be proportional to the similarity of the test input pattern to the learned input pattern.
3. For other test input patterns, the output will always be a blend of the training outputs, with the contribution of each output pattern weighted by the similarity of the corresponding input pattern to the test input pattern.

In the exercises, we will see how these properties lead to several desirable features of pattern associator networks, particularly their ability to generalize based on similarity between test patterns and patterns presented during training.

These properties also reflect the limitations of the Hebbian learning rule; when the input patterns used in training the network do not form an orthogonal set, it is not in general possible to avoid contamination, or "cross-talk," between the response that is appropriate to one pattern and the response that occurs to the others. This accounts for the failure of Hebbian learning with the second set of training patterns considered in Figure 1. The reader can check that the input patterns we used in our first training example in Figure 1 (which was successful) were orthogonal but that the patterns used in the second example were not orthogonal.

### The Delta Rule in Pattern Associator Models

Once again, the delta rule allows us to overcome the orthogonality limitation imposed by the Hebb rule. For the pattern associator case, the delta rule for a particular input-target pair  $i_t, t_t$  is

$$\Delta w_{ij} = \epsilon (t_{it} - o_{it}) i_{jt}. \quad (15)$$

Therefore the weights that result from an ensemble of learning pairs indexed by  $l$  can be written:

$$w_{ij} = \epsilon \sum_l (t_{il} - o_{il}) i_{jl}. \quad (16)$$

It is interesting to compare this to the Hebb rule. Consider first the case where each of the learned patterns is orthogonal to every other one and is presented exactly once during learning. Then  $\mathbf{o}_l$  will be  $\mathbf{0}$  (a vector of all zeros) for all learned patterns  $l$ , and the above formula reduces to

$$w_{ij} = \epsilon \sum_l t_{il} i_{jl}. \quad (17)$$

In this case, the delta rule produces the same results as the Hebb rule; the teaching input simply replaces the output pattern from Equation 13. As long as the patterns remain orthogonal to each other, there will be no cross-talk between patterns. Learning will proceed independently for each pattern. There is one difference, however. If we continue learning beyond a single epoch, the delta rule will stop learning when the weights are such that they allow the network to produce the target patterns exactly. In the Hebb rule, the weights will grow linearly with each presentation of the set of patterns, getting stronger without bound.

In the case where the input patterns  $\mathbf{i}_l$  are not orthogonal, the results of the two learning procedures are more distinct. In this case, though, we can observe the following interesting fact: We can read Equation 15 as indicating that the change in the weights that occurs on a learning trial is storing an association of the input pattern with the *error* pattern; that is, we are adding to each weight an increment that can be thought of as an association between the *error* for the output unit and the activation of the input unit. To see the implications of this, let's examine the effects of a learning trial with input pattern  $\mathbf{i}_l$  paired with output pattern  $\mathbf{t}_l$  on the output produced by test pattern  $\mathbf{i}_t$ . The effect of the change in the weights due to this learning trial (as given by Equation 15) will be to change the output of some output unit  $i$  by an amount proportional to the error that occurred for that unit on the learning trial,  $e_i$ , times the dot product of the learned pattern with the test pattern:

$$\Delta o_{it} = k e_{il} (\mathbf{i}_l \cdot \mathbf{i}_t)_n.$$

Here  $k$  is again equal to  $\epsilon$  times the number of input units  $n$ . In vector notation, the change in the output pattern  $\mathbf{o}_t$  can be expressed as

$$\Delta \mathbf{o}_t = k e_l (\mathbf{i}_l \cdot \mathbf{i}_t)_n$$

Thus, the change in the output pattern at test is proportional to the error vector times the normalized dot product of the input pattern that occurred

during learning and the input pattern that occurred during test. Two facts follow from this:

1. If the input on the learning trial is identical to the input on the test trial so that the normalized dot product is 1.0 and if  $k = 1.0$ , then the change in the output pattern will be exactly equal to the error pattern. Since the error pattern is equal to the difference between the target and the obtained output on the learning trial, this amounts to one trial learning of the desired association between the input pattern on the training trial and the target on this trial.
2. However, if  $\mathbf{i}_l$  is different from  $\mathbf{i}_t$  but not completely different so that  $(\mathbf{i}_l \cdot \mathbf{i}_t)_n$  is not equal to either 1 or 0, then the output produced by  $\mathbf{i}_t$  will be affected by the learning trial. The magnitude of the effect will be proportional to the magnitude of  $(\mathbf{i}_l \cdot \mathbf{i}_t)_n$ .

The second effect—the transfer from learning one pattern to performance on another—may be either beneficial or interfering. Importantly, for patterns of all +1s and -1s, the transfer is always less than the effect on the pattern used on the learning trial itself, since the normalized dot product of two different patterns must be less than the normalized dot product of a pattern with itself. This fact plays a role in several proofs concerning the convergence of the delta rule learning procedure (see Kohonen, 1977, and *PDP:11* for further discussion).

### The Linear Predictability Constraint Again

Earlier we considered the linear predictability constraint for training a single output unit. Since the pattern associator can be viewed as a collection of several different output units, the constraint applies to each unit in the pattern associator. Thus, to master a set of patterns there must exist a set of weights  $w_{ij}$  such that

$$t_{ip} = \sum_j w_{ij} i_{jp} \quad (18)$$

for all output units  $i$  for all target-input pattern pairs  $p$ .

Another way of putting this set of constraints that is appropriate for the pattern associator is as follows: An arbitrary output pattern  $\mathbf{o}_p$  can be correctly associated with a particular input pattern  $\mathbf{i}_p$  without ruining associations between other input-output pairs, only if  $\mathbf{i}_p$  cannot be written as a *linear combination* of the other input patterns. A pattern that cannot be written as a linear combination of a set of other patterns is said to be

*linearly independent* from these other patterns. When all the members of a set of patterns are linearly independent, we say they form a *linearly independent set*. To ensure that arbitrary associations to each of a set of input patterns can be learned, the input patterns must form a linearly independent set.

It is worth noting that the linear independence constraint is primarily a constraint on the similarity relations among input patterns. If we consider the input patterns to be representations of environmental inputs, then whether a set of weights exists that allows us to associate arbitrary responses with each environmental input depends on the way in which these environmental inputs are represented as patterns of activation over a set of input units inside the system. As long as we already have a way of representing a set of environmental inputs so that they are linearly independent, the delta rule will be able to associate any arbitrary responses with these environmental inputs.

Although this is a serious constraint, it is worth noting that there are cases in which the response that we need to make to one input pattern can be predictable from the responses that we make to other patterns with which they overlap. In these cases, the fact that the pattern associator produces a response that is a combination of the responses to other patterns allows it to produce very efficient, often rule-like solutions to the problem of mapping each of a set of input patterns to the appropriate response. We will examine this property of pattern associators in the exercises.

### Nonlinear Pattern Associators

Not all pattern associator models that have been studied in the literature make use of the linear activation assumptions we have been using in this analysis. Several different kinds of nonlinear pattern associators (i.e., associators in which the output units have nonlinear activation functions) fall within the general class of pattern associator models. These nonlinearities have effects on performance, but the basic principles that we have observed here are preserved even when these nonlinearities are in place. In particular:

1. Orthogonal inputs are mutually transparent.
2. The learning process converges with the delta rule as long as there is a set of weights that will solve the learning problem; the nonlinearities that have been tried tend not to have much effect on which sets of patterns can be learned and which cannot.
3. What is learned about one pattern tends to transfer to others.

## THE FAMILY OF PATTERN ASSOCIATOR MODELS

With the above as background, we turn to a brief specification of several members of the class of pattern associator models that are available through the **pa** program. These are all variants on the pattern associator theme. Each model consists of a set of input units and a set of output units. The activations of the input units are *clamped* by externally supplied input patterns. The activations of the output units are determined in a single two-phase processing cycle. First, the net input to each output unit is computed. This is the sum of the activations of the input units times the corresponding weights, plus an optional bias term associated with the output unit:

$$net_i = \sum_j w_{ij} + bias_i. \quad (19)$$

### Activation Functions

After computing the net input to each output unit, the activation of the output unit is then determined according to an activation function. Several variants are available:

- *Linear.* Here the activation of output unit  $i$  is simply equal to the net input.
- *Linear threshold.* In this variant, each of the output units is a *linear threshold unit*; that is, its activation is set to 1 if its net input exceeds 0 and is set to 0 otherwise. Units of this kind were used by Rosenblatt in his work on the perceptron (1959).
- *Stochastic.* This is the activation function used in *PDP:18* and *PDP:19*. Here, the output is set to 1, with a probability  $p$  given by the logistic function:

$$p(o_i=1) = \frac{1}{1 + e^{-net_i/T}} \quad (20)$$

This is the same activation function used in Boltzmann machines.

- *Continuous sigmoid.* In this variant, each of the output units takes on an activation that is nonlinearly related to its input according to the logistic function:

$$o_i = \frac{1}{1 + e^{-net_i/T}} \quad (21)$$

Note that this is a continuous function that transforms net inputs between  $+\infty$  and  $-\infty$  into real numbers between 0 and 1. This is the activation function used in the back propagation networks we will study in Chapter 5.

## Learning Assumptions

Two different learning rules are available in the **pa** program:

- *The Hebb rule.* Hebbian learning in the pattern associator model works as follows. Activations of input units are clamped based on an externally supplied input pattern, and activations of the output units are clamped to the values given by some externally supplied target pattern. Learning then occurs by adjusting the strengths of the connections according to the Hebbian rule:

$$\Delta w_{ij} = \epsilon o_i i_j. \quad (22)$$

- *The delta rule.* Error-correcting learning in the pattern associator model works as follows. Activations of input units are clamped to values determined by an externally supplied input pattern, and activations of the output units are calculated as described earlier. The difference between the obtained activation of the output units and the target activation, as specified in an externally supplied target pattern, is then used in changing the weights according to the following formula:

$$\Delta w_{ij} = \epsilon (t_i - o_i) i_j. \quad (23)$$

## The Environment and the Training Epoch

In the pattern associator models, there is a notion of an *environment* of pattern pairs. Each pair consists of an input pattern and a corresponding output pattern. A training *epoch* consists of one learning trial on each pattern pair in the environment. On each trial, the input is presented, the corresponding output is computed, and the weights are updated. Patterns may be presented in fixed sequential order or in permuted order within each epoch.



## Performance Measures

After processing each pattern, several measures of the output that is produced and its relation to the target are computed. One of these is the normalized dot product of the output pattern with the target. This measure is called the *ndp*. We have already described this measure quantitatively; here we note that it gives a kind of combined indication of the similarity of two patterns and their magnitudes. In the cases where this measure is most useful—where the target is a pattern of +1s and -1s—the magnitude of the target is fixed and the normalized dot product varies with the similarity of the output to the target and the magnitude of the output itself. To unconfound these factors, we provide two further measures: the *normalized vector length*, or *nvl*, of the output vector and the *vector correlation*, or *vcor*, of the output vector with the target vector. The *nvl* measures the magnitude of the output vector, normalizing for the number of elements in the vector. It has a value of 1.0 for vectors consisting of all +1s and -1s. The *vcor* measures the similarity of the vectors independent of their length; it has a value of 1.0 for vectors that are perfectly correlated, 0.0 for orthogonal vectors, and -1.0 for anticorrelated vectors.

Quantitative definitions of vector length and vector correlation are given in *PDP:9* (pp. 376-379).<sup>3</sup> For our purposes, it suffices to note the relation of these three measures. When the target pattern consists of +1s and -1s, the normalized dot product of the output pattern and the target pattern is equal to the normalized vector length of the output pattern times the vector correlation of the output pattern and the target:

$$ndp = nvl \times vcor. \quad (24)$$

In addition to these measures, we also compute the *pattern sum of squares* or *pss* and the *total sum of squares* or *tss*. The *pss* is the sum over all output units of the squared error, where the error for each output unit is the difference between the target and the obtained activation of the unit. This quantity is computed for each pattern processed. The *tss* is just the sum over the *pss* values computed for each pattern in the training set. These measures are not very meaningful when learning occurs by the Hebb rule, but they are meaningful when learning occurs by the delta rule.

<sup>3</sup> The normalization of vector length is not discussed in *PDP:9*. To compute the normalized vector length, one first divides the magnitude of each element by the square root of the number of elements in the vector, then computes the length of the resulting vector according to the formula given in *PDP:9*, p. 376.

## IMPLEMENTATION

The `pa` program implements the pattern associator models in a very straightforward way. The program is initialized by defining a network, as in previous chapters. A PA network is assumed to consist of some number of input units (*ninputs*) and some number of output units (*noutputs*). Connections are allowed from input units to output units only. Each output unit may also have a bias term, which is treated as though it were a weight from a unit whose activation was set to 1.0, as in previous chapters. The network specification file defines the number of input units and output units, as well as the total number of units, and indicates which connections and bias terms to output units exist and which are modifiable. Usually, each input unit has a modifiable connection to each output unit, and all of these are initialized to 0, but other possibilities may be specified in the network specification file. It is also generally necessary to read in a file specifying the set of pattern pairs that make up the environment of the model.

Once the program is initialized, learning occurs through calls to a routine called *train*. This routine carries out *nepochs* of training, either in sequential order (if the routine is called by entering the *strain* command) or in permuted order (if the routine is called by entering the *ptrain* command). The routine exits if the total sum of squares measure, *tss*, is less than some criterion value, *ecrit*. Here is the *train* routine:

```
train(c) char c; {
/* c = 's' for strain, 'p' if ptrain */

    for (t = 0; t < nepochs; t++) {
        tss = 0.0;
        epochno++;

/* make a list of pattern numbers */
        for (i = 0; i < npatterns; i++)
            used[i] = i;

/* if ptrain, permute list */
        if (c == 'p') {
            for (i = 0; i < npatterns; i++) {
                npat = rnd() * (npatterns - i) + i;
                old = used[i];
                used[i] = used[npat];
                used[npat] = old;
            }
        }
        for (i = 0; i < npatterns; i++) {

/* set the pattern number, then do trial */
            patno = used[i];
```

```

        trial();
        if (lflag) change_weights();
    }
    if (tss < ecrit)
        return;
}
}

```

The *trial* routine runs each individual trial. It calls four other routines: one that sets the input and the target patterns, one that computes the activations of the output units from the activations of the input units, one that computes the error measure, and one that computes the various summary statistics:

```

trial() {
    setup_pattern();
    compute_output();
    compute_error();
    sumstats();
}

```

Below we show the *compute\_output* and the *compute\_error* routines. First, *compute\_output*:

```

compute_output() {
    for (i = ninputs; i < nunits; i++) {
        /* accumulate net input to each output unit */
        netinput[i] = bias[i];
        for (j = 0; j < ninputs; j++) {
            netinput[i] += activation[j]*weight[i][j];
        }
        /* set activation based on netinput */
        if (linear) {
            activation[i] = netinput[i];
        }
        else if (lt) {
            if (netinput[i] > 0)
                activation[i] = 1.0;
            else
                activation[i] = 0.0;
        }
        else if (cs) {
            activation[i] = logistic(netinput[i]);
        }
    }
}

```

```

else { /* default, stochastic mode */
    activation[i] =
        probability(logistic(netinput[i]));
}
}
}

```

The choice of activation function is indicated by the values of mode variables called *linear*, *lt*, and *cs*. The variable *lt* stands for linear threshold, and *cs* stands for continuous sigmoid. The default case, in which the output is stochastic, involves a call to the *logistic* function shown in Chapter 3. This function returns a number between 0 and 1. The result is then used to set the activation of the unit to 0 or 1 based on the *probability* function, which returns a 1 with probability equal to its argument.

The *compute\_error* function is exceptionally simple for the *pa* program:

```

compute_error() {

/* use i for output units, t for target elements */
for (i = ninputs, t = 0; i < nunits; t++, i++) {
    error[i] = target[t] - activation[i];
}
}

```

Note that when the targets and the activations of the output units are both specified in terms of 0s and 1s, the error will be 0, 1, or -1.

If learning is enabled (as it is by default in the program, as indicated by the value of the *lflag* variable), the *train* command calls the *change\_weights* routine, which actually carries out the learning:

```

change_weights() {

if (hebb) {
    for (i = ninputs; i < nunits; i++) {
        for (j = 0; j < ninputs; j++) {
            weight[i][j] +=
                epsilon[i][j]*target[i]*activation[j];
        }
        bias[i] += bepsilon[i]*target[i];
    }
}

else { /* delta rule, by default */
    for (i = ninputs; i < nunits; i++) {
        for (j = 0; j < ninputs; j++) {
            weight[i][j] +=
                epsilon[i][j]*error[i]*activation[j];
        }
    }
}
}

```

```

        bias[i] += bepsilon[i]*error[i];
    }
}

```

The matrix *epsilon[i][j]* contains modifiability parameters for each connection in the network. Generally, modifiable connections are set to a value equal to the parameter *lrte*; unmodifiable connections are set to 0. Similarly, the array *bepsilon[i]* has modifiability parameters for the bias terms for each output unit.

Note that for Hebbian learning, we use the target pattern directly in the learning rule, since this is mathematically equivalent to clamping the activations of the output units to equal the target pattern and then using these activations.

## RUNNING THE PROGRAM

The **pa** program is used much like the other programs we have described in earlier chapters. The main things that are new for this program are the *strain* and *ptrain* commands for training pattern associator networks.

There are also changes to the *test* command, and there is a new command called *tall*. The *test* command allows you to test the network's response to a particular input pattern, either one in the program's list of pattern pairs or one entered directly from the keyboard; *tall* allows you to test the network's response to all of the patterns in the list of pattern pairs with learning turned off so as not to change the weights while testing. As in the **cs** program, commands *newstart* and *reset* are both available as alternative methods for reinitializing the programs. Recall that *reset* reinitializes the random number generator with the same seed used the last time the program was initialized, whereas *newstart* seeds the random number generator with a new random seed. Although there can be some randomness in **pa**, the problem of local minima does not arise and different random sequences will generally produce qualitatively similar results, so there is little reason to use *reset* as opposed to *newstart*. There are several *mode* variables in **pa**: *linear*, *lt* (for linear threshold), and *cs* (for continuous sigmoid) that determine the activation rule that is in force. The default is stochastic, which is used when *linear*, *lt*, and *cs* are all 0. Another mode variable, *hebb*, determines whether the learning rule is the Hebb rule (*hebb*=1) or the delta rule (which is the default). The new control variables include *nepochs*, the number of training epochs run when the *strain* and *ptrain* commands are entered, and *ecrit*, the criterion value for the error measure. The *lflag* variable allows direct control over whether learning is on, and the *stepsize* variable now has two new values, which are *pattern* and *epoch*. When *stepsize* is set to *pattern*, the screen is updated after each pattern is processed

(i.e., at the end of each trial). When set to *epoch*, the screen is updated after each epoch of processing.

There are only three parameters in *pa*. Most important is *lr*, which is equivalent to the parameter  $\epsilon$  from the "Background" section. The other two parameters are *noise*, which determines the amount of random variability added to elements of input and target patterns, and *temp*, used as the denominator of the logistic function to scale net inputs in *cs* and the default stochastic mode. There are also several new performance measures: the normalized dot product, *ndp*; the normalized vector length measure, *nvl*; the vector correlation measure, *vcor*; the pattern sum of squares, *pss*; and the total sum of squares, *tss*.

### New or Altered Commands

Here follows a more detailed description of the new or altered commands available in *pa*.

#### *newstart*

Seeds the random number with a new random seed, and then returns the program to its initial state before any learning occurred. That is, sets all weights to 0, and sets *nepochs* to 0. Also clears activations and updates the display.

#### *ptrain*

Permuted training command. Runs *nepochs* training epochs, presenting each pattern pair in the pattern list once in each epoch. Order of patterns is randomized for each epoch. Returns early if interrupted or if *tss* is less than *ecrit*. (During training, individual elements of each input and target pattern may be randomly distorted by the model by adding random noise, if desired. See the entry for the *noise* parameter in the variable list.)

#### *reset*

Same as *newstart*, but reseeds the random number generator with the same seed that was used last time the network was initialized.

#### *strain*

Sequential training command. Same as *ptrain*, except that pattern pairs are presented in the same, fixed order in each epoch. The order is simply the order in which the pattern pairs are encountered in the list.

#### *tall*

Temporarily turns learning off and tests all patterns in sequential order. Temporarily sets *single* to 1 and sets *stepsize* to *pattern* so that the program updates the screen and pauses after each pattern is tested.

*test*

Allows testing of the response to an individually specified input pattern against an individually specified target. Prompts first for the input pattern, then for the target. Prompt for input is

**input (#N, ?N, E for enter):**

If the user types

- #N Tests with the *N*th input pattern on the list (*N* is an integer).
- #name Tests with the pattern named *name* (*name* is a character string).
- ?N Tests with a distorted version of pattern *N*. The pattern is distorted by adding uniformly distributed noise to the stored pattern with range as specified by the value of the *noise* parameter.
- ?name Tests with a distorted version of pattern *name*.
- E Prompts for a specific pattern to test. The pattern consists of a sequence of floating-point numbers or "+", "-", and "." characters indicating entries for the pattern. Entries must be separated by spaces or *return* and followed by an extra *return* or *end*.

The prompt for the target is analogous; the user can specify a particular target to use in the same way the input pattern is specified. In this case, #0 indicates target pattern 0.

*get/ patterns*

Prompts for a file containing a set of pattern *pairs*. Each pair consists of an input pattern (called *ipattern*) and a target pattern (called *tpattern*). Each specification begins with a name, followed by *nin-puts* entries specifying the elements of the input pattern and *nout-puts* entries specifying the elements of the corresponding target. Entries may be floating-point numbers or "+" (stands for 1.0), "-" (stands for -1.0) or "." (stands for 0.0). Pattern pairs are numbered internally by the program in the order encountered starting from 0.

## Variables

All of the new or altered variables used in the *pa* program are given in the following list. They are all accessible using the *set* and *exam* commands.

*ecrit*

Error criterion for stopping training. If the *tss* at the end of an epoch of training is less than *ecrit*, training stops.

*lflag*

Learning flag; normally set to 1. When nonzero, learning is enabled. Learning is automatically disabled during the *tall* or *test* commands.

*nepochs*

Number of training epochs run by each call to the *strain* and *ptrain* commands.

*stepsize*

Step size for updating the screen and for pausing if *single* is set to 1 during *strain* and *ptrain*. Values allowed are *cycle*, *pattern*, *epoch*, and *nepochs*. If the value is *cycle*, the screen is updated after processing each pattern and then updated again (if *lflag* is set to 1) after the weights are changed. If the value is *pattern*, the screen is only updated after the weights are changed. If the value is *epoch*, the screen is updated at the end of each epoch; if the value is *nepochs*, the screen is updated only when the *strain* and *ptrain* commands return.

*config/ ninputs*

Number of input units. This is crucial both for defining the network and for reading in pattern pairs.

*config/ noutputs*

Number of output units. Also crucial for defining the network and reading pattern pairs.

*config/ nunits*

Number of units; should equal the sum of *ninputs* plus *noutputs*.

*env/ ipattern*

Input pattern array. Prompts for an input pattern number or name, then for an element number, and then displays the value of the specified element in the specified pattern.

*env/ npatterns*

Number of input-target pattern pairs in the program's pattern list. This variable is set implicitly in reading in the input patterns.

*env/ pname*

Pattern name vector. Prompts for a pattern number and gives its name.

*env/ tpattern*

Target pattern array; analogous to input pattern array. Prompts for a target pattern number or name and then for an element number. Element 0 is the first element of each target pattern.

*mode/ cs*

Flag variable (standing for "continuous sigmoid"). When nonzero, each output unit takes on an activation based on the logistic function of its net input.



*mode/ hebb*

Flag variable. When nonzero, the Hebb rule is used in updating the weights; otherwise, the delta rule is used.

*mode/ linear*

Flag variable. When nonzero, a linear activation function is used to set the activations of the output units on the basis of the input units. If this and mode variables *lt* and *cs* are 0, the default stochastic activation function is used.

*mode/ lt*

Flag variable (standing for "linear threshold"). When nonzero, each output unit takes on an activation based on the linear threshold function so that its activation is set to 1 if its input is greater than 0; otherwise, its activation is set to 0.

*param/ lrate*

The learning rate parameter. Scales the size of the changes made to the weights. Generally, if there are  $n$  input units, the learning rate should be less than or equal to  $1/n$ .

*param/ noise*

Range of the random distortion added to each input and target pattern specification value during training and testing. The value added is uniformly distributed in the interval  $[-noise, +noise]$ . See the *strain*, *ptrain*, and *test* commands.

*param/ temp*

Denominator used in the logistic function to scale net inputs in both *cs* (continuous sigmoid) and default (stochastic) modes. Generally, *temp* can be set to 1, except in the simulations of the rule of 78 (see Ex. 4.4). Note that there is only one cycle of processing in *pa*, so there is no annealing.

*state/ cpname*

Name of the current pattern, as given in the pattern file.

*state/ epochno*

Number of the current epoch; updated at the beginning of each epoch.

*state/ error*

Vector of errors, or differences between the current target pattern and the current pattern of activation over the output units.

*state/ input*

Vector of activations of the input units in the network, based on the current input pattern (subject to the effects of noise).

*state/ ndp*

Normalized dot product of the obtained activation vector over the output units and the target vector.

*state/ netinput*

Vector of net inputs to each output unit.

*state/ nvl*

Normalized length of the obtained activation vector over the output units.

*state/ output*

Vector of activations of output units in the network.

*state/ patno*

The number of the current pattern, updated at the beginning of processing the pattern. Note that this is the index of the pattern on the program's pattern list; when *ptrain* is used, it is not the same as the pattern's position within the random training sequence in force for a particular epoch.

*state/ pss*

Pattern sum of squares, equal to the sum over all output units of the squared difference between the target for each unit and the obtained activation of the unit.

*state/ target*

Vector of target values for output units, based on the current target pattern, subject to effects of noise.

*state/ tss*

Total sum of squares, equal to the sum of all patterns so far presented during the current epoch of the pattern sum of squares.

*state/ vcor*

Vector correlation of the obtained activation vector over the output units and the target vector.

## OVERVIEW OF EXERCISES

In these exercises, we will study several basic properties of pattern associator networks, starting with their tendency to generalize what they have learned to do with one input pattern to other similar patterns; we will explore the role of similarity and the learning of responses to unseen prototypes. These first studies will be done using a completely linear Hebbian pattern associator. Then, we will shift to the linear delta rule associator of the kind studied by Kohonen (1977) and analyzed in *PDP:11*. We will study what these models can and cannot learn and how they can be used to learn to get the best estimate of the correct output pattern, given noisy input and outputs. Finally, we will examine the acquisition of a rule and an exception to the rule in a nonlinear (stochastic) pattern associator.

### Ex. 4.1. Generalization and Similarity With Hebbian Learning

In this exercise, you will train a linear Hebbian pattern associator on a single input-output pattern pair, and study how its output, after training, is

affected by the similarity of the input pattern used at test to the input pattern used during training.

After you have created a working directory for the `pa` program and all of the associated files, you can start up this exercise by entering the command:

```
pa 8x8.tem lin.str
```

The files `8x8.tem` and `lin.str` set up the network to be a linear Hebbian pattern associator with eight input units and eight output units, starting with initial weights that are all 0. The `.str` file sets the value of the learning rate parameter to 0.125, which is equal to 1 divided by the number of units. With this value, the Hebb rule will learn an association between a single input pattern consisting of all +1s and -1s and any desired output pattern perfectly in one trial.

The file `one.pat` contains a single pattern (or, more exactly, a single input-output pattern pair) to use for training the associator. Both the input pattern and the output pattern are eight-element vectors of +1s and -1s. Load this pattern file:

```
get pat one.pat
```

When it is loaded, the single input and target pattern will appear in the upper right corner of the display area.

Now you can train the network on this first pattern pair for one epoch. The `nepochs` variable has been set to 1 in the `lin.str` file, but before you start it might be best to set `single` to 1 so that you can watch things progress:

```
set single 1
```

Now, you can train the network by simply entering

```
strain
```

Because `stepsize` is set to `cycle` in the `.str` file, the program will present the first (and, in this case, only) input pattern, compute the output based on the current weights, and then display the input, output, and target patterns, as well as some summary statistics. If you have set `single` to 1, it will pause, with the display shown in Figure 4. At the top of the display you will see the **p to push / b to break / <cr> to continue:** prompt, under which is the top-level menu. Below this is the display area. In the upper left corner of the display area, you will see some summary information, including the current `epochno` (which is 1, indicating the first epoch of training; note that `epochno` changes at the beginning of each epoch and stays the same until the next epoch begins) and the current `patno` (which is 0, indicating that pattern 0 is being processed). Below these entries are the `ndp`, or normalized dot product, of the output obtained by the network with the target pattern; the `nvl`, or normalized vector length, of the obtained output pattern;

```

p to push/b to break/<cr> to continue:
disp/ exam/ get/ save/ set/ clear do log newstart ptrain quit reset
run strain tall test

epochn      1
cpname      a
ndp         0.0000
nvl         0.0000
vcor        0.0000
pss         8.0000
tss         8.0000
                                out   tar
weights     0 0 0 0 0 0 0 0 0 0   0   100
              0 0 0 0 0 0 0 0 0 0   0   100
              0 0 0 0 0 0 0 0 0 0   0   100
              0 0 0 0 0 0 0 0 0 0   0   100
              0 0 0 0 0 0 0 0 0 0   0   100
              0 0 0 0 0 0 0 0 0 0   0   100
              0 0 0 0 0 0 0 0 0 0   0   100
              0 0 0 0 0 0 0 0 0 0   0   100
input       100100100100100100100100

```

FIGURE 4. Display layout for the first *pa* exercise while processing pattern *a*, before any learning has occurred.

and the *vcor*, or vector correlation, of the output with the target. All of these numbers are 0 because the weights are 0, so the input produces no output at all. Below these numbers are the *pss*, or pattern sum of squares, and the *tss*, or total sum of squares. They are the sum of squared differences between the target and the actual output patterns. The first is summed over all output units for the current pattern, and the second is summed over all patterns so far encountered within this epoch (they are, therefore, identical at this point).

Below these entries you will see the weight matrix on the left, with the input vector that was presented for processing below it and the output and target vectors to the right.

The display presents all of the network variables (inputs, weights, activations, etc.) in *hundredths*, and reverse video means negative, so that 1.0 displays as 100 and  $-1.0$  displays as 100 in reverse video. The *ipattern* and *tpattern* arrays (upper right corner of the display area) show only one digit per pattern element because they are always +1 and  $-1$  in this exercise. You will see that the input pattern shown below the weights matches the single input pattern shown under the label *ipattern* and that the target pattern shown to the right of the weights matches the single target pattern shown under *tpattern*. The output itself is all 0s.

If you enter *return* at this point, the target will first be clamped onto the output units, then the weights will be updated according to the Hebbian learning rule:

$$\Delta w_{ij} = (\text{irate}) o_i i_j. \quad (25)$$

Enter *return* one more time to return from the first epoch of training.

- Q.4.1.1. Explain the values of the weights in rows 1 and 2 (counting from 0). Note that the display does not indicate the third decimal place, which in this case is a 5; each weight has absolute value 0.125 at this point. Explain the values of the weights in column 7, the last column of the matrix.

Now, with just this one trial of learning, the network will have "mastered" this particular association, so that if you test it at this point, you will find that, given the learned input, it perfectly reproduces the target. You can test the network using the *test* command. Simply enter *test*, and respond to the prompts. The first prompt asks you to specify an input pattern. Your choices are *#N*, *?N*, and *E*, where *N* stands for the numerical index of a pattern. If you enter *#0*, for example, the model will set the input to be *ipattern 0*. The *?N* option is used to specify a distortion of a particular pattern (we will consider this again in a moment); the *E* option allows you to enter a specific pattern of your own choosing. For your first test, enter *#0* for the input. Do the same for the target. The display will now show the actual output computed by the network.

In this particular case the display will not change much because in the previous display the output had been clamped to reflect the very target pattern that the network has now computed. The only thing that actually changes in the display are the *ndp*, *vcor*, and *nvl* fields; these will now reflect the normalized dot product and correlation of the computed output with the target and the normalized length of the output. They should all be equal to 1.0 at this point.

You are now ready to test the generalization performance of the network. To do this, you can enter specific input patterns of your own, instead of *#0*, for the input pattern requested by the test routine. The following exchange indicates the inputting of the pattern *+ - + - + + + +*, followed by a specification that the "target" is still *ipattern 0*:

```
pa: test
input (#N, ?N, E for enter): E
give input elements: + - + - + + + + end
target (#N, ?N, E for enter): #0
```

Note that the elements of the input pattern must be separated by spaces. After this interchange, the network will test the specified input pattern, report the output and the target, and indicate the *ndp*, *vcor*, and *nvl* of the obtained output with the input.

- Q.4.1.2. Try a number of different input patterns, testing each against the *#0* target. Observe the *ndp*, *vcor*, and *nvl* in each case. Relate the obtained output to the specifics of the weights and the input patterns used and to the discussion in the "Background" section about the test output we should get from a linear Hebbian

associator, as a function of the normalized dot product of the input vector used at test and the input vector used during training. Include in your set of patterns one that is orthogonal to the training pattern and one that is perfectly anticorrelated with it, as well as one or two others with positive normalized dot products with the input pattern.

If you understand the results you have obtained in this exercise, you understand the basis of similarity-based generalization in one-layer associative networks. In the process, you should come to develop your intuitions about vector similarity and to clearly be able to distinguish uncorrelated patterns from anticorrelated ones.

#### Ex. 4.2. Orthogonality, Linear Independence, and Learning

This exercise will expose you to the limitation of a Hebbian learning scheme and show how this limitation can be overcome using the delta rule. For this exercise, you are to set up two different sets of training patterns: one in which all the input patterns form an orthogonal set and the other in which they form a linearly independent, but not orthogonal, set. For both cases, choose the output patterns so that they form an orthogonal set, then arbitrarily assign one of these output patterns to go with each input pattern. In both cases, use only three pattern pairs and make sure that both patterns in each pair are eight elements long. The pattern files you construct in each case should contain three lines formatted like the single line in the *one.pat* file:

```
first + - + - + - + -   + + - - + + - -
```

We provide sets of patterns that meet these conditions in the two files *ortho.pat* and *li.pat*. You may use these files if you wish, but you will be more certain of your understanding if you construct your own sets of patterns and make sure that they can be learned.

- Q.4.2.1. For each set of patterns, do the following experiment. Read the patterns into the program using the *get/ patterns* command. Reset the network (this clears the weights to 0s). Then run one epoch of training using the Hebbian learning rule by entering the *strain* command. Following this, execute the *tall* command to test the model's performance. What happens with each pattern? Run three additional epochs of training (one at a time), following each with a *tall*. What happens? In what ways do things get better? In what ways do they stay the same? Why?

- Q.4.2.2. Turn off *hebb* mode in the program (this puts the default, the delta rule, in place) and try the above experiment again. Describe the similarities and differences between the results and explain in terms of the differential characteristics of the Hebbian and delta rule learning schemes.
- Q.4.2.3. With the linearly independent set of patterns, keep running training epochs using the delta rule until the *tss* measure drops below 0.05. Examine and explain the resulting weight matrix, contrasting it with the weight matrix obtained after one cycle of Hebbian learning with the same patterns. What are the similarities between the two matrices? What are the differences? Try to explain rather than just describe the differences.

For the next question, reinitialize your network and train the network with the pattern set in the file *li.pat*. (It makes no difference whether you use *reset* or *newstart* to reinitialize since randomness plays no role in this exercise.) Run one epoch of training at a time, and examine performance at the end of each epoch using the *tall* command.

- Q.4.2.4. In *li.pat*, one of the input patterns is orthogonal to both of the others, which are partially correlated with each other. Which input-output pair is mastered first? Why?
- Q.4.2.5. As the final exercise in this set, construct sets of two or more pattern pairs that cannot be effectively mastered, either by Hebbian or delta rule learning. Explain why they cannot be learned, and describe what happens when the network tries to learn them, both in terms of the course of learning and in terms of the weights that result.

*Hints.* We provide a set of impossible pattern pairs in the file *imposs.pat*, but it is preferable for you to try to construct your own. You will probably want to use a small value of the learning rate; this affects the size of the oscillations that you will probably observe in the weights. A learning rate of about 0.0125 is probably good. Keep running more training epochs until the *tss* at the end of each epoch stabilizes.

### Ex. 4.3. Learning Central Tendencies

One of the positive features of associator models is their ability to filter out noise in their environments. In this exercise we invite you to explore this aspect of pattern associator networks. For this exercise, you will still

be using linear units but with the delta rule and with a relatively small learning rate. You will also be introducing noise into your training patterns.

A start-up file that sets things up nicely for this exercise is provided, called *ct.str* (*ct* is for "central tendency"). This file sets the learning rate to 0.0125 and makes sure *hebb* mode is off. It also sets the *noise* variable to 0.5. This means that each element in each input pattern and in each target pattern will have its activation distorted by a random amount uniformly distributed between +0.5 and -0.5.

Start up the program with the command

```
pa 8x8.tem ct.str
```

Then load in a set of patterns (your orthogonal set from Ex. 4.2 or the patterns in *ortho.pat*). Then you can see how well the model can do at pulling out the "signals" from the "noise." The clearest way to see this is by studying the weights themselves and comparing them to the weights acquired with the same patterns without noise added.

Q.4.3.1. Compare learning of the three orthogonal patterns you used in Ex. 4.2, without noise, to the learning that occurs in this exercise, with noise added. Compare the weight matrix acquired after "noiseless" learning with the matrix that evolves given the noisy input-target pairs that occur in the current situation. Run about 60 epochs of training to get an impression of the evolution of the weights through the course of training and compare the results to what happens with errorless training patterns (and a higher learning rate). What effect does changing the learning rate have when there is noise? Try higher and lower values.

*Hints.* You may find it useful to rerun the relevant part of Ex. 4.2 (Q.4.2.2) and to use the *save/ screen* command to store the weight matrices you obtain in the different runs. For longer runs, remember that you can set *nepochs* to a number larger than the default value of 1. Each time *strain* is entered, *nepochs* of training are run.

The results of this simulation are relevant to the theoretical analyses described in *PDP:11* and are very similar to those described under "central tendency learning" in *PDP:25*, where the effects of amnesia (taken as a reduction in connection strength) are considered.

#### Ex. 4.4. Lawful Behavior

We now turn to one of the principle characteristics of pattern associator models that has made us take interest in them: their ability to pick up



regularities in a set of input-output pattern pairs. The ability of pattern associator models to do this is illustrated in the past-tense learning model, discussed in *PDP:18*. Here we provide the opportunity to explore this aspect of pattern associator models, using the example discussed in that chapter, namely, the *rule of 78* (see *PDP:18*, pp. 226-234). We briefly review this example here.

The rule of 78 is a simple rule we invented for the sake of illustration. The rule first defines a set of eight-element input patterns. In each input pattern, one of units 1, 2, and 3 must be on; one of units 4, 5, and 6 must be on; and one of units 7 and 8 must be on. For the sake of consistency with *PDP:18*, we adopt the convention for this example only of numbering units starting from 1. The rule of 78 also defines a mapping from input to output patterns. For each input pattern, the output pattern that goes with it is the same as the input pattern, except that if unit 7 is on in the input pattern, unit 8 is on in the output and vice versa. Table 1 shows this rule.

The rule of 78 defines 18 input-output pattern pairs. Eighteen *arbitrary* input-output pattern pairs would exceed the capacity of an eight-by-eight pattern associator, but as we shall see, the patterns that exemplify the rule of 78 can easily be learned by the network.

The version of the pattern associator used for this example follows the assumptions we adopted in *PDP:18* for the past-tense learning model. Input units are binary and are set to 1 or 0 according to the input pattern. The output units are binary, stochastic units and take on activation values of 0 or 1 with probability given by the logistic function:

$$p(\text{act}_i = 1) = \frac{1}{1 + e^{-\text{net}_i/T}} \quad (26)$$

TABLE 1  
THE RULE OF 78

Input patterns consist of one active unit from each of the following sets:	(1 2 3) (4 5 6) (7 8)
The output pattern paired with a given input pattern consists of:	The same unit from (1 2 3) The same unit from (4 5 6) The other unit from (7 8)
Examples:	2 4 7 → 2 4 8 1 6 8 → 1 6 7 3 5 7 → 3 5 8
An exception:	1 4 7 → 1 4 7

(From *PDP:18*, p. 229.)

where  $T$  is equivalent to the *temp* parameter. Note that, although this function is the same as for the Boltzmann machine, the calculation of the output is only done once, as in other versions of the pattern associator; there is no annealing, so *temp* is just a scaling factor.

Learning occurs according to the delta rule, which in this case is equivalent to the perceptron convergence procedure because the units are binary. Thus, when an output unit should be on (target is 1) but is not (activation is 0), an increment of size *lrate* is added to the weight coming into that unit from each input unit that is on. When an output unit should be off (target is 0) but is not (activation is 1), an increment of size *lrate* is subtracted from the weight coming into that unit from each input unit that is on.

For this example, we follow *PDP:18* and use *temp* of 15. This means that the net input to a unit needs to be about 45 for it to come on with probability .95. Thus, learning will be gradual, even with an *lrate* of 2.0, which is what we use in these examples. Note, though, that you can speed things up by using a larger value of the learning rate. (The simulations that you will do here will not conform to the example in *PDP:18* in all details, since in that example an approximation to the logistic function was used. The basic features of the results are the same, however.)

To run this example, you will need to start up the *pa* program with the *78.tem* and *78.str* files:

```
pa 78.tem 78.str
```

The *78.str* file will read in the appropriate network specification file (in *8x8.net*) and the 18 patterns that exemplify the rule of 78, then display these on the screen to the right of the weight matrix. Since the units are binary, there is only a single digit of precision for both the input, output, and target units. Given the *lrate* of 2.0, the weights take on integer values divisible by 2. Their actual values are displayed in the cells of the weight matrix; they are not multiplied first by 100, as in the previous exercises.

You should now be ready to run the exercise. The variable *nepochs* is initialized to 10, so if you enter one of the training commands, 10 epochs of training will be run. We recommend using *ptrain* because it does not result in a consistent bias in the weights favoring the patterns later in the pattern list. The screen is updated once per pattern after the weights have been adjusted, so you should see the weights and the input, output, and target bits changing. The *pss* and *tss* (which in this case indicate the number of incorrect output bits) will also be displayed once per pattern.

Q.4.4.1. At the end of the 10th epoch, the *tss* should be in the vicinity of 20, or about one error per pattern; this means that the model is getting each output bit correct with a probability of about .9. Given the values of the weights and the fact that *temp* is set to 15, calculate the net input to the last output unit for the first two

input patterns, and (using Figure 7 of Chapter 3 or your calculator) calculate the approximate probability that this last output unit will receive the correct activation in each of these two patterns. The probabilities should be in the upper .80s or low .90s.

At this point you should be able to see the solution to the rule of 78 patterns emerging. Generally, there are large positive weights between input units and corresponding output units, with unit 7 exciting unit 8 and unit 8 exciting unit 7. You'll also see rather large inhibitory weights from each input unit to each other unit within the same subgroup (i.e., 1, 2, and 3; 4, 5, and 6; and 7 and 8). Run another 20 or so epochs, and a subtler pattern will begin to emerge.

Q.4.4.2. Generally there will be slightly negative weights from input units to output units in other subgroups. See if you can understand why this happens. Note that this does not happen reliably for weights coming into output units 7 and 8. Your explanation should explain this too.

At this point, you have watched a simple PDP network learn to behave in accordance with a simple rule, using a simple, local learning scheme; that is, it adjusts the strength of each connection in response to its errors on each particular learning experience, and the result is a system that exhibits lawful behavior in the sense that it conforms to the rule.

For the next part of the exercise, you can explore the way in which this kind of pattern associator model captures the three-stage learning phenomenon exhibited by young children learning the past tense in the course of learning English as their first language. To briefly summarize this phenomenon: Early on, children know only a few words in the past tense. Many of these words happen to be exceptions, but at this point children tend to get these words correct. Later in development, children begin to use a much larger number of words in the past tense, and these are predominantly regular. At this stage, they tend to overregularize exceptions. Gradually, over the course of many years, these exceptions become less frequent, but adults have been known to say things like *ringed* or *taked*, and lower-frequency exceptions tend to lose their exceptionality (i.e., to become regularized) over time.

The 78 model can capture this pattern of results; it is interesting to see it do this and understand how and why this happens. For this part of the exercise, you will want to reset the weights, and read in the file *hf.pat*, which contains an exception pattern ( $147 \rightarrow 147$ ) and one regular pattern ( $258 \rightarrow 257$ ). If we imagine that the early experience of the child consists mostly of exposure to high-frequency words, a large fraction of which are irregular (8 of the 10 most frequent verbs are irregular), this approximates the early experience the child might have with regular and irregular past-tense forms. If you run 20 epochs of training using *ptrain* with these two

patterns, you will see a set of weights that allows the model to set each output bit correctly in each pattern about 85% of the time. At this point, you can read in the file *all.pat*, which contains these two pattern pairs, plus all of the other pairs that are consistent with the rule of 78. This file differs from the *78.pat* file only in that the input pattern 147 is associated with the "exceptional" output pattern 147 instead of what would be the "regular" corresponding pattern 148. Save the screen displaying the weights that resulted from learning *hf.pat*. Then read in *all.pat* and run 10 more epochs.

Q.4.4.3. Given the weights that you see at this point, what is the network's most probable response to 147? Can you explain why the network has lost the ability to produce 147 as its response to this input pattern? What has happened to the weights that were previously involved in producing 147 from 147?

One way to think about what has happened in learning the *all.pat* stimuli is that the 17 regular patterns are driving the weights in one direction and the single exception pattern is fighting a lonely battle to try to drive the weights in a different direction, at least with respect to the activation of units 7 and 8. Since eight of the input patterns have unit 7 on and "want" output unit 8 to be on and unit 7 to be off and only one input pattern has input unit 7 on and wants output unit 7 on and output unit 8 off, it is hardly a fair fight.

If you run more epochs, though, you will find that the network eventually finds a compromise solution that satisfies all of the patterns.

Q.4.4.4. If you have the patience, run the model until it finds a set of weights that gets each output unit correct about 90% of the time for each input pattern (90% correct corresponds to a net input of +30 or so for units that should be on and -30 for units that should be off). Explain why it takes so long to get to this point.

### Further Suggested Exercises

In the exercise just described, there was only one exception pattern, and when vocabulary size increased, the ratio of regular to exception patterns increased from 1:1 to 17:1. Pinker and Prince (1987) have shown that, in fact, as vocabulary size increases, the ratio of regular to exception verbs stays roughly constant at 1:1. One interesting exercise is to set up an analog of this situation. Start training the network with one regular and one exception pattern, then increase the "vocabulary" by introducing new regular patterns and new exceptions. Note that each exception should be idiosyncratic; if all the exceptions were consistent with each other, they would simply exemplify a different rule. You might try an exercise of this

form, setting up your own correspondence rules, your own exceptions, and your own regime for training.

You can also explore other variants of the pattern associator with other kinds of learning problems. One thing you can do easily is see whether the model can learn to associate each of the individuals from the Jets and Sharks example in Chapter 2 with the appropriate gang (relying only on their properties, not their names; the files *jets.tem*, *jets.str*, *jets.net*, and *jets.pat* are available for this purpose). Also, you can play with the continuous sigmoid (or logistic) activation function by setting the *cs* mode flag to 1 (for the rule of 78 example, it is best to use the *8x8.tem* file since the weights have real values and to cut *lrate* to about 0.12 and *temp* to 1.0 so that the weights stay in range; remember weights and activations are displayed as hundredths in this template).

