CHAPTER 6

Other Learning Models: Auto-Associators and Competitive Learning

There are a number of learning paradigms in PDP systems—each with a characteristic goal or task. These paradigms include the pattern association paradigm, in which the goal is to learn mappings between specific input and output pairs; the auto-associator paradigm, in which the goal is to store specific patterns for future retrieval; and the regularity detection paradigm, in which the goal is to discover salient features of the ensemble of patterns. Thus far in this book we have focused almost entirely on the pattern association paradigm for learning. Clearly the pattern associator of Chapter 4 and the back propagation model of Chapter 5 are both examples of systems learning input-output mappings. The current chapter focuses on the other two paradigms. We begin with a discussion of several simple auto-associators and then move to a discussion of one of the most studied regularity detection models, *competitive learning.*

THE AUTO-ASSOCIATOR

BACKGROUND

The auto-associator models are a class of related models that share the auto-associative architecture. That is, they all consist of a single set of units that are completely interconnected. In some ways, this architecture is the most general architecture for a connectionist system; all other architectures are more restricted subsets of this architecture. However, given the

learning rules that we will be exploring for training these networks in the present chapter, auto-associators are limited by the fact that they can only train connections between units whose target activations can be specified from outside the network.

In spite of this limitation, auto-associators have several interesting properties. They can learn to do pattern completion and to rectify or restore distorted versions of learned patterns to their original form. They can learn to extract the prototype of a set of patterns from distorted exemplars presented during training. Discussions of these and other aspects of auto-associators may be found in Anderson (1977), Anderson, Silverstein, Ritz, and Jones (1977), Kohonen (1977), and in *PDP:17* and *PDP:25*.

The auto-associator models we will consider in this section are similar to pattern associators, with one major difference: There is only a single set of units, and instead of having connections from input units to output units, each unit serves as both an input unit and an output unit, so that each unit is connected to every other unit. In some versions, it may also be connected to itself. A picture of an auto-associator is shown in Figure 1. In all the versions of the auto-associator that we will consider here, input patterns consist of vectors specifying positive and negative inputs to the units from outside the network. There are no bias terms on the units. Units take on activation values that may be positive or negative, based on these external inputs and on the connections they receive from other units inside the network.

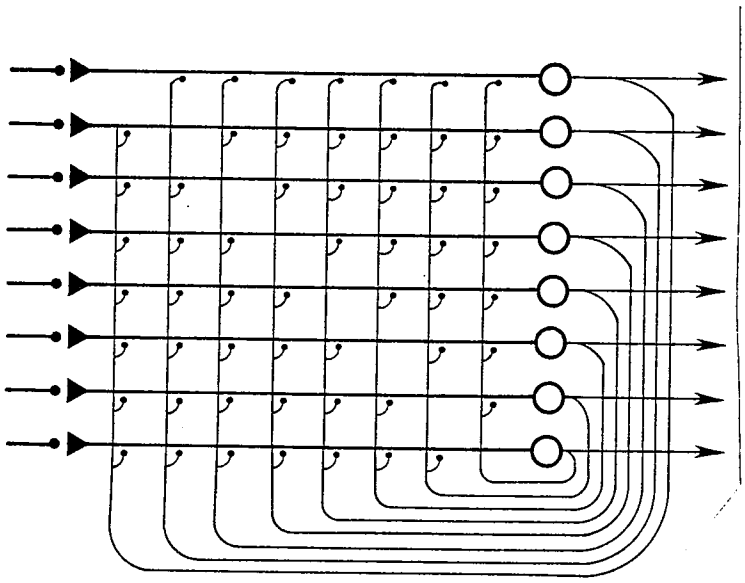

FIGURE 1. A simple eight-unit auto-associative network. (From "Distributed Memory and the Representation of General and Specific Information" by J. L. McClelland and D. E. Rumelhart, 1985, *Journal of Experimental Psychology, 114*, 159-188. Copyright 1985 by the American Psychological Association. Reprinted by permission.)

A basic understanding of the essential properties of the auto-associator can best be achieved by considering a linear, Hebbian version of a pattern associator in which the input patterns and the output patterns happen to be the same. For this case we can use what we learned in Chapter 4 about the pattern associator, noting that the associations are now between a pattern and itself. Specifically, we recall that the output produced in response to test input pattern $\mathbf{i}_t$ is proportional to the sum of the output patterns experienced during learning, each weighted by the similarity of the corresponding input pattern to the test input pattern:

$$\mathbf{o}_t = k\sum_l \mathbf{o}_l(\mathbf{i}_l \cdot \mathbf{i}_t)_n. \tag{1}$$

The constant of proportionality, k, is equal to the learning rate parameter, ϵ, times the number of units in the network. Since we are considering the case in which the training consists of associating each input vector with itself, the training output vectors $\mathbf{o}_l$ can be replaced with the training input vectors $\mathbf{i}_l$. In this case, the output at test is equal to the sum of the input patterns used during training, each weighted by its similarity to the input pattern used at test. Given this equation, we can immediately observe the following points:

- If a test input pattern is orthogonal to all of the input patterns used during training, then the network will produce a null output.
- If a test pattern is orthogonal to all but one of the training patterns and is identical to this other training pattern, then the output will be equal to the test pattern scaled by the value of k.
- If the same pattern is presented m times during learning, then it will be as though there are m patterns "stored" in the network that are identical to it. Therefore if this same pattern is presented as a test input, the output will be equal to m times k times the test pattern.

More succinctly, we can say that if we associate a set of orthogonal patterns, each with itself, in a linear Hebbian associator, and if we test with one of these stored patterns, then the output will be equal to a scaled version of the input, and the scale factor will be proportional to the number of times we have experienced the pattern during learning.

Patterns that are scaled by a network are called *eigenvectors*; eigenvector simply means "same vector." The magnitude of the eigenvector, as it is processed by the network, is called its *eigenvalue*. For our linear Hebbian auto-associator trained with an orthogonal set of learning patterns, the learned patterns form a set of eigenvectors. Their eigenvalues are km_l, where m_l is the number of presentations of learning pattern l.

Now, however, suppose that we present a pattern that has some similarity to each of several different stored patterns. Then we find that the output produced is a blend of these stored patterns, with the contribution of each weighted by its similarity to the test pattern times its eigenvalue. As an example, suppose we have stored these two patterns:

$$\mathbf{i}_0\text{: } +1.00\ +1.00\ +1.00\ +1.00\ -1.00\ -1.00\ -1.00\ -1.00$$
$$\mathbf{i}_1\text{: } +1.00\ -1.00\ +1.00\ -1.00\ +1.00\ -1.00\ +1.00\ -1.00$$

and we test with the following pattern:

$$\mathbf{i}_t\text{: } +1.00\ +1.00\ +1.00\ +1.00\ -1.00\ -1.00\ +1.00\ -1.00$$

We find that the normalized dot product of pattern $\mathbf{i}_0$ with pattern $\mathbf{i}_t$ $(\mathbf{i}_0 \cdot \mathbf{i}_t)_n$ is 0.75, and the normalized dot product of pattern $\mathbf{i}_1$ with pattern $\mathbf{i}_t$ $(\mathbf{i}_1 \cdot \mathbf{i}_t)_n$ is 0.25. If each has been stored exactly once and k is equal to 1.0, then we will get as our output 0.75 times $\mathbf{i}_1$ plus 0.25 times $\mathbf{i}_2$, so the resulting output pattern is

$$\mathbf{o}_t\text{: } +1.00\ +0.50\ +1.00\ +0.50\ -0.50\ +1.00\ -0.50\ -1.00$$

This vector is not the same as the input vector $\mathbf{i}_t$, so $\mathbf{i}_t$ is not an eigenvector of this network. The response is a weighted sum of the stored vectors, with the weights depending both on the similarity of the input to each stored vector and on the eigenvalues of these vectors. We will see in the exercises that when the output of the auto-associator is fed back into itself and nonlinearities are introduced, the output can often end up exactly matching the most similar pattern used during learning. We call this process the *pattern rectification process.*

A special case of pattern rectification is what is called the *pattern completion process.* This is what happens when we present an incomplete vector in which some of the +1s and −1s have been replaced by 0s. Thus, if we have previously stored patterns $\mathbf{i}_0$ and $\mathbf{i}_1$ as above, we can present an incomplete version of one of these patterns as a test input pattern and the network will fill in or complete the remainder. Thus suppose we present the following test pattern:

$$\mathbf{i}_t\text{: } +1.00\ -1.00\ +1.00\ -1.00\ \ 0.00\ \ 0.00\ \ 0.00\ \ 0.00$$

In this case, $\mathbf{i}_0 \cdot \mathbf{i}_t$ is 0.0 and $\mathbf{i}_1 \cdot \mathbf{i}_t$ is 0.5. The network will produce the output vector $\mathbf{o}_t$,

$$\mathbf{o}_t\text{: } +0.50\ -0.50\ +0.50\ -0.50\ +0.50\ -0.50\ +0.50\ -0.50$$

in response to this input. Note that this vector points in the same direction as the stored vector $\mathbf{i}_1$, but it is of lesser magnitude.

In general, in completion with orthogonal input patterns and linear units we obtain a scaled version of the incomplete stored vector that is probed, where the scale factor is equal to the normalized dot product of the stored vector and the incomplete version of it that is used as the probe.

The pattern completion and rectification processes we have been describing are general characteristics of auto-associator models. Another general characteristic is their tendency to learn to respond better to the prototype, or central tendency, of a set of distorted exemplars of a category than to any of the individual distortions themselves. This characteristic arises from the fact that each new distortion learned is superimposed in the connection strengths; the characteristics of the individual exemplars tend to average out as more and more exemplars are presented. This characteristic of auto-associators is discussed at length in Anderson et al. (1977) and in *PDP:17*, and is explored extensively in the exercises.

So far we have been treating the auto-associator as if it were a pattern associator in which the input and output patterns just happen to be the same. In fact, though, the input and output patterns happen to be the same because the input and output units are really the same units. This gives the auto-associator the capability of multiple processing cycles in which the initial pattern of activation is produced by some external input, and each successive cycle involves updating the activations of the units, based on the continuing external input, plus what we call the *internal input*—the input to each unit via the connections internal to the net. The internal input to unit i, $intinput_i$, is given by

$$intinput_i = \sum_j w_{ij} a_j.$$

This internal input is equivalent to the output that would be produced by a linear pattern associator. In the auto-associator, it is combined with the continuing external input to each unit, and is then treated in different ways in the different variants of the auto-associator model, which are described later.

Learning Regimes for Auto-Associators

Both the Hebb rule and the delta rule are available for use in auto-associator models. When the Hebb rule is used, the external input is assumed to be clamped onto the units for the purpose of training. In this case the formula for updating the weights is

$$\Delta w_{ij} = \epsilon\,(extinput_i)\,(extinput_j).$$

When the delta rule is used, the external input pattern is applied at the beginning of time cycle 1 and is left on. Processing goes on for *ncycles*. At the end of *ncycles*, a variant of the delta rule is used to adjust the strengths

of the connections in the network. In this variant, the goal of learning is to have the internal input to each unit match the external input. In this case, the error measure for each unit, $error_i$, is defined to be

$$error_i = extinput_i - intinput_i$$

where the $intinput_i$ is the value at the end of *ncycles* of processing, based on the activations at the end of the preceding cycle.

In the general formulation of the auto-associator, each unit is assumed to be connected to every other unit, including itself. In networks with large numbers of units, these self-connections are unimportant, but in smaller networks trained with the delta rule, where the goal is to learn connections that foster pattern completion and rectification, strong self-connections can tend to defeat learning. This is because self-connections allow units to predict their own activation, thus reducing the error and preventing the network from learning strong between-unit connections that can perform the completion and rectification processes. Thus, when the delta rule is used in an auto-associator, it is best to force the connection from each unit to itself to remain fixed at 0.

Limitations of the Auto-Associator

The limitations of the auto-associator are similar to the limitations of the pattern associator. When trained using the Hebb rule, perfect reproduction of learned patterns can only be obtained if orthogonal patterns are used; with nonorthogonal patterns there is always some cross-talk between the patterns. When trained using the delta rule, the learning process converges only if the following linear predictability constraint can be met:

> Over the entire set of patterns, the external input to each unit must be predictable from a linear combination of the activations of each unit that projects to it.

This constraint, for example, prevents the auto-associator without hidden units from learning to turn on a unit when two other units are both on or both off, while at the same time turning the unit off when one of the two other units is on and one is off.

Auto-associators can be constructed in which there are hidden units, of course; a simple example is described at the end of *PDP:17*. More generally, encoder networks as described in *PDP:5* are examples of auto-associators with hidden units. The auto-associator models used in the present chapter, however, do not contain hidden units.

In the sections that follow, we describe three main variants of the auto-associator. All of these will be considered in the exercises.

The Linear Auto-Associator

Perhaps the simplest variant of the auto-associator is what we will call the linear auto-associator. In this model, the change in activation of each unit on each processing cycle is a weighted sum of the external and internal inputs to the unit, less a decay term that tends to restore activation to a resting level of 0:

$$\Delta a_i = (estr)extinput_i + (istr)intinput_i - (decay)a_i. \quad (2)$$

Note that the $extinput_i$ and $intinput_i$ together make up the net input to unit i (there is no bias term). The parameters *estr* and *istr* scale the contributions of the external and internal input to each unit, as in the constraint satisfaction models considered in Chapter 3.

This model is mathematically very simple, and it is typically used in the following way. At some time $t = 0$, activations of all units are set to 0. At the beginning of cycle 1, a pattern of +1s and −1s is supplied as the external input and is left on until the end of *ncycles* of processing. On the first cycle of processing, since the prior activations of all the units are all 0, each unit takes on an activation equal to *estr* times the external input pattern. After that, processing proceeds in accordance with Equation 2. For simplicity, we will study the case in which the decay parameter is set to 1.0. In this case, the activation of each unit at time t ($a_i(t)$) is given by

$$a_i(t) = (estr)extinput_i(t) + (istr)intinput_i(t).$$

Here $intinput_i(t)$ is based on the activations of the units at time $t-1$.

A Difficulty with the Linear Auto-Associator

The linear auto-associator model is very useful for illustrating the basic pattern completion and regularization processes described above. A difficulty, however, is that the network can "blow up"; that is, activations can become very large, very quickly as a result of the self-reinforcing feedback characteristic of the network. When the model is run with *ncycles* equal to 2, this is not a problem. With larger values of *ncycles*, some form of nonlinearity must be introduced. The next two variants of the auto-associator involve introducing different types of nonlinearity into the basic model.

The Brain-State-in-the-Box Model

One form of nonlinearity that keeps activations from growing without bound is introduced in the "brain state in the box" or BSB model proposed

by Anderson et al. (1977). In this model, activations are prevented from growing larger than $+C$ or smaller than $-C$. In our version of this model, we will use $C = 1.0$.

The effect of this "clipping" operation, of course, is to prevent activations of units from growing without bound; instead it keeps them in a hypercube, or box, bounded by +1.0 and −1.0 on each dimension. Small inputs may still be amplified by the network, but when the activations of the units reach +1.0 or −1.0, they are simply cut off. This has an interesting side effect: It means that processing tends to result in patterns of activation that correspond to corners of the hypercube, that is, states that consist of all +1s and −1s. The corners tend to correspond to the patterns that had previously been learned. In this case, as we shall see in the exercises, the auto-associative process tends to drive incomplete or distorted versions of stored patterns toward the stored patterns, producing perfect rectification and completion.

In the version of the BSB model that we shall consider in the exercises, the learning rule is the same as in the linear model already described. It is also possible to use the variant of the delta rule described earlier with the BSB model.

The DMA Model

The final auto-associator model we will consider is the model of distributed memory and amnesia described in *PDP:17* and *PDP:25*. Here we call this model the *DMA model.* This model grew out of our work with the interactive activation and competition scheme described in Chapter 2. In this model, we think of the combined external and internal input to each unit as driving the activation of the unit upward or downward, depending on whether it is excitatory or inhibitory. The magnitude of the effect of the input is dependent on the distance to the maximum or minimum activation value. First we define the net input to unit i:

$$netinput_i = (estr)extinput_i + (istr)intinput_i.$$

If the net input is positive,

$$\Delta a_i = netinput\,(max - a_i) - (decay)a_i,$$

and, if it is negative,

$$\Delta a_i = netinput\,(a_i - min) - (decay)a_i.$$

This model is similar to the BSB model in that activations are kept between the values of *max* and *min*, which are set to +1.0 and −1.0. The main difference is that activations always level off at less extreme values,

since at some point the "restoring" force of the decay term will match the "perturbing" force of the net input term.

Learning in the DMA model takes place using the variant of the delta rule we described earlier. In this rule, when the error is 0, the internal input to a unit matches the external input, and the total net input to a unit is the sum of the *istr* and *estr* parameters times the external input:

$$netinput_i = (estr + istr)extinput_i.$$

In contrast, before learning, when the internal input is 0, we find that the net input is simply

$$netinput_i = (estr)extinput_i.$$

The effect of this difference is to change the asymptotic activation values of the units. From our consideration of the interactive activation and competition model of Chapter 2, we recall that at asymptote, the activation of a unit is given by

$$a_i = \frac{netinput_i}{netinput_i + decay}.$$

Before learning, then,

$$a_i = \frac{(estr)extinput_i}{(estr)extinput_i + decay},$$

while after learning,

$$a_i = \frac{(estr + istr)extinput_i}{(estr + istr)extinput_i + decay}.$$

In most of the simulations reported in *PDP:17* and *PDP:25*, *estr*, *istr*, and *decay* were all set to 0.15, and external inputs used in training patterns are always patterns of +1s and −1s. This means that before learning, units take on activations of 0.50 times the sign of the external input; after learning, this value grows to 0.67. These values, of course, can be moved around at will by changing the values of *estr*, *istr*, and *decay*. The basic point is that the network is more strongly activated by familiar patterns than by unfamiliar ones. It also exhibits pattern completion and rectification, as in the other variants of the auto-associator.

IMPLEMENTATION

The auto-associative models are implemented in the **aa** program. In this program, processing is implemented much as it is in the **iac** program described in Chapter 2. The main difference is that in **aa** the output of a

unit is identical with its activation; there is no check to see that the activation exceeds threshold. Both positive and negative activation values exert influences on other units.

What the **aa** program adds to **iac** is an outer loop that runs epochs of training trials. In each trial, after *ncycles* of processing, the error measure is computed and the connection strengths are modified. The routines for doing this are analogous to those used in the pattern associator.

RUNNING THE PROGRAM

The **aa** program is run in much the same way as the programs already described. The program is called with a *.tem* file and a *.str* file. Because of the simplicity of the **aa** architecture (each unit connected to every other unit), a *.net* file is not needed; instead, *nunits* is defined near the top of the *.str* file. This leads the program to create a network of *nunits* units, with a connection from each unit to itself and every other unit. Generally, a *.pat* file is used to specify a list of patterns for use in training and testing.

The *.str* file generally specifies the size of the net *(nunits)* and specifies which of several possible modes should be on or off. The DMA model is the default. The linear Hebbian model can be studied by setting *linear* mode to 1 and by setting the *hebb* mode to 1. You can study the BSB model by setting the *bsb* mode to 1. There is also a *selfconnect* mode, which is set to 0 by default; in this mode the weight from a unit to itself is forced to remain at 0.0. To study the effects of allowing nonzero self-connections this mode can be set to 1.

The facilities for training and testing are the same as those used in the **pa** and **bp** programs. The *strain* command is used to train the network using a fixed sequential order of training in each epoch. The *ptrain* command is used to train the network using a permuted order of presentations in each epoch. Both commands run *nepochs* of training, ending when interrupted or when the total sum of squares *tss* becomes smaller than the criterion *ecrit*.

During training, it is possible to specify that the training patterns should be randomly distorted. In **aa**, distortion is done by independently changing the sign of each bit (from + to − or from − to +) in each training pattern with probability *pflip* before it is presented to the network for training. A *pflip* of 0 produces no changing; a *pflip* of .5 produces totally random patterns. Note that this method of distortion is different from the one provided in the **pa** program.

To test the network, the *test* command allows testing using either one of the stored patterns, a distortion of one of these patterns, or any pattern entered directly as a sequence of +'s and −'s. At the end of *ncycles* of processing, the normalized dot product of the input with the output, the normalized length of the activation vector produced, and the correlation of the

output with the external input are displayed. The *ctest* command is used for testing the pattern completion capability of the model. It allows the user to specify a part of a pattern to clear to see how well the model can do in filling it back in again. In this case the *ndp*, *nvl*, and *vcor* measures apply to the subpattern of activation filled in by the network on the cleared units rather than to the overall pattern of activation.

New or Altered Commands

The following list mentions only those commands in the **aa** program that are not the same as commands in the **pa** program.

ctest

Allows the user to perform a completion test on an individual pattern. The user specifies which input units to clear, (that is, to set to 0) for completion testing. The *ctest* command prompts for a pattern name or number to test, then asks for a first element to clear (a number from 0 to *nunits* − 1), and then asks for a last element to clear (the last element must be greater than the first and less than *nunits*). Both the beginning and the end elements given are cleared, as well as all the units in between. The statistics computed (*ndp*, *nvl*, and *vcor*) will apply to the cleared portion of the pattern, assessed against the pattern that would have been present had these bits not been cleared.

test

Allows testing of an individual pattern. The following arguments can be given:

- *#N* Instructs *test* to use the corresponding pattern from the pattern list (*N* is a pattern name or number).
- *?N* Instructs *test* to use a distorted version of the corresponding pattern (*N* is as above). Each element has its sign flipped with probability equal to the value of the *pflip* parameter.
- *L* Instructs *test* to use the last pattern tested; this pattern is left in place.
- *E* Instructs *test* to accept a pattern entered by the user. Pattern elements are floating-point numbers or ".", "+", or "−", corresponding to 0.0, +1.0, and −1.0. Elements must be separated by spaces and the list of elements must be terminated by *end* or an extra *return*.

get/ patterns

Reads in a pattern file containing a list of pattern specifications. Each pattern specification consists of a pattern name followed by *nunits* entries indicating the values of each element of the pattern.

Entries can be floating-point numbers or "+" (for 1.0), "−" (for −1.0), or "." (for 0.0).

get/ rpatterns

Causes the program to construct a set of random patterns (vectors of +1s and −1s) with a specified probability that each unit will be +1. Prompts for two arguments as follows:

How many patterns?
(give desired number of patterns to construct)
make input + with probability:
(give desired probability for elements to be positive)

This list is stored in the program's internal *ipattern* list and can be saved using the *save/ patterns* command. Patterns are assigned names of the form *rN* where *N* is the pattern number.

save/ patterns

Allows the user to save the patterns in the program's pattern list in a file.

The **aa** program does not provide a *cycle* command to continue cycling if you wish to run more cycles with the *test* or *ctest* commands. Instead you must set *ncycles* to a larger number and run the *test* or *ctest* command again. With *test* you can enter *L* as the argument to exactly repeat the previous test.

Variables

The following list mentions only those variables that are new or different in the **aa** program. As usual, all of the variables are accessed via the *set/* and *exam/* commands.

stepsize

The default *stepsize* in **aa** is *pattern*. This means that a step consists of presenting an input pattern as the external input, resetting all the activations in the network, running *ncycles* of processing, computing error information and summary statistics, and changing weights if *lflag* is set. Other possible values of *stepsize* are *cycle*, which causes updating/pausing to occur after each cycle; *epoch*, which causes updating/pausing to occur only at the end of an entire processing epoch; and *nepochs*, which causes updating/pausing to occur only at the end of *nepochs*.

mode/ bsb

When *bsb* is set to 1, activations are clipped at +1 and −1. This mode has no effect unless the *linear* mode is also in force since

activations are otherwise restricted to the [1, −1] interval by the DMA activation equations.

mode/ hebb

When *hebb* is set to 1, the program uses the Hebbian learning rule. When *hebb* is 0 (the default), the delta rule is used.

mode/ linear

By default, the activations are updated according to the DMA activation equations. When *linear* is set to 1, the activation process is linear, subject to clipping at +1 and −1 if *bsb* mode is also set.

mode/ selfconnect

By default, when *selfconnect* is 0, the weight from each unit to itself is fixed at 0.0. When *selfconnect* is set to 1, self-connections are trained just like all other connections in the network.

param/ estr

Scales the magnitude of the external input to each unit. The scaling is applied in determining the net inputs to the units but is not applied in computing errors.

param/ istr

Scales the magnitude of the internal input to each unit. Scaling is applied as with *estr*.

param/ lrate

The learning rate parameter. Generally, its value should be less than 1/*nunits*.

param/ pflip

The probability that pattern elements have their signs flipped during training and when flipping is requested in using the *test* command.

state/ error

Vector of errors for each unit. Each element is the difference between the unit's external input and its internal input.

state/ extinput

Vector of external inputs to units. Note that this is displayed before the effects of scaling the external input by the *estr* parameter are applied.

state/ intinput

Vector of internal inputs to units from other units. Note that this vector is displayed before the effects of scaling the external inputs by the *istr* parameter are applied.

state/ ndp

Normalized dot product of the current external input pattern with the current activation pattern. Updated at the end of every cycle when *stepsize* = *cycle* or at the end of every epoch otherwise.

state/ nvl

The normalized length or strength of the activation vector. Updated like *ndp*.

state/ prioract
: Vector of activations from the preceding processing cycle.

state/ vcor
: The vector correlation of the present pattern of activation with the external input. Updated like *ndp*.

OVERVIEW OF EXERCISES

We provide four exercises for use with the different auto-associator models. Ex. 6.1 explores the linear Hebbian associator and examines its handling of sets of orthogonal patterns. Ex. 6.2 explores the BSB model and its pattern completion and reactivation capabilities. Ex. 6.3 examines the linear auto-associator with delta rule learning, focusing on exploring the characteristics of ensembles of patterns that influence whether they can be learned in a one-layer auto-associative network. Finally, Ex. 6.4 examines some of the psychological characteristics of auto-associator models, and allows the user to run variants of several of the examples discussed in *PDP:17* (pp. 182-192).

Ex. 6.1. The Linear Hebbian Associator

In this first exercise, you can familiarize yourself with the use of the **aa** program and study the effects of learning sets of patterns in the linear Hebbian auto-associator.

To start you off, we have provided the following relevant files: *lh8.tem, lh8.str*, and *two.pat*. The *lh8.str* file sets up a linear Hebbian auto-associator with eight units. It sets *hebb* mode to 1, sets *linear* mode to 1, and sets *selfconnect* mode to 1. It sets several parameters to values that make the behavior of the auto-associator particularly transparent. The *decay* variable is set to 1.0. This means that the activation on each trial is simply the sum of the external input (which is turned on and left on throughout processing) and the internal input from the units in the network, based on the pattern of activation achieved at the end of the previous cycle of processing. The *istr* and *estr* parameters are set to 1.0, so the net input is equal to the sum of the external input plus the internal input. The *lrate* parameter is set to 1/*nunits*, or 0.125. This means that in one learning trial, weights that give each of several orthogonal patterns an eigenvalue of 1.0 will be stored in the network. For initial testing, the file *two.pat* is supplied with two orthogonal patterns named *a* and *b*.

To run the program, you type:

aa lh8.tem lh8.str

The resulting screen display is similar to the display for the **pa** program. The left column displays some of the prominent variables relevant to the Hebbian auto-associator. The first three are the pattern number, the cycle number, and the epoch number. Below these are the normalized dot product of the activation vector with the external input, the normalized length of the pattern of activation, and the correlation of the pattern of activation with the external input vector. To the right of these variables is the weight matrix. This matrix shows the value of the weight from the unit in each column to the unit in each row. These values are multiplied by 100, so 10 means 0.10 and 100 means 1.0. Of course, reverse video indicates negative numbers as in other programs. To the right of the weights are the external input pattern, the internal input pattern, and the activation pattern that results from these inputs. All these are scaled by 100 as well, so that 100 stands for an actual value of 1.0. Below the weight matrix, the *prioract* vector is displayed. This vector represents the pattern of activation that was present at the end of the previous processing cycle. Like the *activation* vector, this vector is initialized to 0.0 at the beginning of processing each input pattern.

The display shown in Figure 2 shows the results of the first cycle of processing pattern *a* from the file *two.pat*. The file *two.pat* was read in by entering *get/ pat/ two.pat*, and then *single* was set to 1. Following this, the command *strain* was entered. This command runs *nepochs* of learning, but for the example *nepochs* is set to 1, so each pattern will be presented for learning only once as a result of entering this command at this point. After the *strain* command was entered, the program set the external input to

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


                                    weights                    ext int act  pname ipattern
cpname          a          0  0  0  0  0  0  0  0              100   0 100    a   1 1 1 1 1 1 1 1
cycleno         1          0  0  0  0  0  0  0  0              100   0 100    b   1 1 1 1 1 1 1 1
epochno         1          0  0  0  0  0  0  0  0              100   0 100
ndp        1.0000          0  0  0  0  0  0  0  0              100   0 100
nvl        1.0000          0  0  0  0  0  0  0  0              100   0 100
vcor       1.0000          0  0  0  0  0  0  0  0              100   0 100
                           0  0  0  0  0  0  0  0              100   0 100
                           0  0  0  0  0  0  0  0              100   0 100

           prioract        0  0  0  0  0  0  0  0
```

FIGURE 2. The display produced by **aa** with an eight-unit network while processing an input pattern before any learning has taken place.