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.

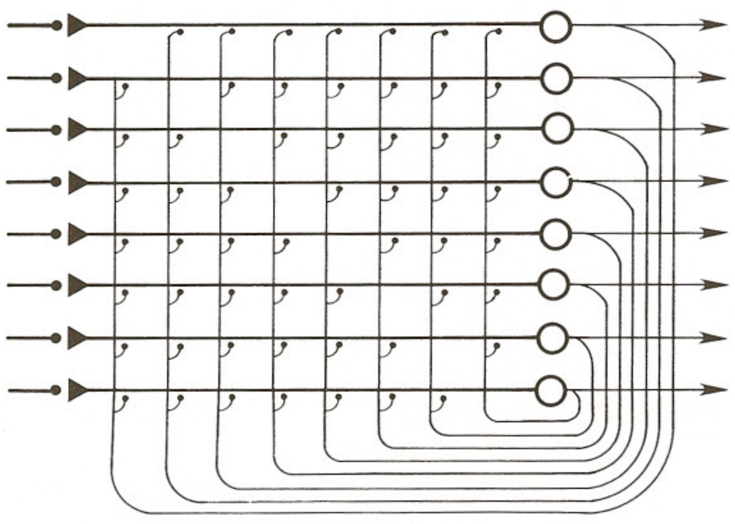


FIGURE 1. A simple eight-unit auto-associative network. (From "Distributed Memory and the Representation of General and Specific Information" by J. L. McClelland and D. E. Rumelhart, 1985, *Journal of Experimental Psychology*, *114*, 159-188. Copyright 1985 by the American Psychological Association.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

$\mathbf{i}_0$: +1.00 +1.00 +1.00 +1.00 −1.00 −1.00 −1.00 −1.00
$\mathbf{i}_1$: +1.00 −1.00 +1.00 −1.00 +1.00 −1.00 +1.00 −1.00

and we test with the following pattern:

$\mathbf{i}_t$: +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

$\mathbf{o}_t$: +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:

$\mathbf{i}_t$: +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 $\mathbf{o}_t$,

$\mathbf{o}_t$: +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:

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

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 $+1$s 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.

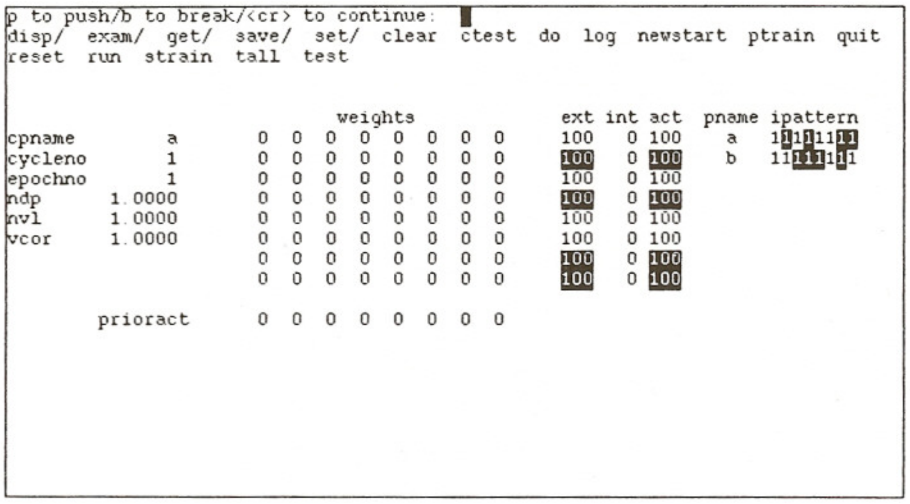
To start you off, we have provided the following relevant files: *lh8.tem, lh8.str*, and *two.pat*. The *lh8.str* file sets up a linear Hebbian auto-associator with eight units. It sets *hebb* mode to 1, sets *linear* mode to 1, and sets *selfconnect* mode to 1. It sets several parameters to values that 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    11111111
cycleno        1         0   0   0   0   0   0   0   0      100   0 100    b    11111111
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.

equal pattern *a* (the vector +−+−++−−) during the first cycle of processing and set the activations of the units based on these external inputs, then paused with the display shown in Figure 2.

The display indicates that the weights are all 0s, that the external input is a pattern of +1s and −1s, matching the first input pattern, that there is no internal input, and that the activations of the units are +1 and −1, matching the external input. The activations are equal to the external inputs since the network is linear and *estr* is equal to 1. So far no internal input has been generated.

To run another cycle, simply type *return*. In this case nothing changes: the external input is still the only thing influencing the activations of the units because the weights are all 0. Since *ncycles* is set to 2, this is the end of processing the first pattern.

At this point, another *return* results in the first pattern being stored in the weights using the Hebb rule. The display that is presented at this point reflects the new values of the weights, as well as the external input that gave rise to them, and the pattern of activation at the end of the preceding processing cycle. Note that the value of the learning rate parameter at this point is 1/*nunits*, or 0.125, as specified in the *lh8.str* file.

Q.6.1.1. Make sure you understand the weight matrix. First, be sure you know which is the receiving unit and which is the sending unit for the weight shown in row 3, column 0. Look at the weights in row 3 of the weight matrix. Why do they have these values? Be sure to explain both the sign and magnitude of the weights. (Remember that the values of the weights are displayed as hundredths and are truncated to only two places, so 12 corresponds to 0.125.)

When the next *return* is entered, the activations are cleared, the second pattern is presented so that its values now appear on the external input, and the first processing cycle is run. At this point the activations reflect the external input alone because there has not yet been a chance for activation to propagate through the connections. After the next *return*, however, the pattern of activation at the end of cycle 1 has had a chance to generate excitatory and inhibitory influences on other units by way of the connections in the network. The reader will note, however, that the internal input to each unit is still 0 at this point.

Q.6.1.2. Explain why the internal input to each unit is 0, even though each unit is producing a nonzero contribution to the net input of every unit.

After another *return*, the second pattern is stored in the weights, and the resulting matrix of weights is displayed. Type one more *return* to get back to the **aa:** prompt.

Q.6.1.3. Describe and explain the weights in row 2 of the weight matrix.

You have now completed training the network once with each of the two patterns in the file *two.pat*, and you are ready to see what happens when you test these two patterns. To do this you should use the *tall* command: when this command is executed, no learning occurs. The patterns are presented in sequential order and processed just as before, but there are no changes made to the weights.

Q.6.1.4. What happens when each of the two learned patterns is processed? Explain.

Now you are ready to try a training experiment of your own. Using the file *two.pat* as your model, generate your own set of four orthogonal patterns; include in the set the two patterns in *two.pat*. Read it into the network using the *get/ patterns* command. (We supply a file called *four.pat* that can be used, but it is better to make up your own.) Test all four patterns using *tall* (see Q.6.1.5 below), based on the weights obtained by training with the patterns in *two.pat*.

You can use the *strain* command to train the network with this new set of patterns on top of the connection strengths obtained by training on the *two.pat* patterns. The new changes to the weights will be added to the changes that are already in place from the first training set. This means that two of the patterns will have been learned twice, whereas the other two will have been learned once each.

Q.6.1.5. Display your set of four orthogonal patterns. Explain what happens when you test each of these four patterns, both before and after learning. Also, describe and explain what happens when you test the network with a vector that is equal to -1.0 times one of the stored vectors. Refer to the facts about eigenvectors in your explanation.

Hints. You can use the *test* command to enter the vector for this last part of the question.

Q.6.1.6. Set the number of processing cycles (*ncycles*) to 4, and use the *test* command to test one of the new (once-learned) patterns and one of the old (twice-learned) patterns. What happens with each? Explain.

Now construct a set of eight orthogonal patterns of $+1$s and -1s (hint: one of the vectors must be all $+1$s or all -1s). Reset the weights to 0 using the *reset* command, set *ncycles* back to 2, and train the network through one training epoch on all eight patterns, using the *strain* command.

Q.6.1.7. Describe the set of weights that results from this training experience. What will happen at this point if you present an arbitrary vector of +1s and −1s? Explain both in terms of the weights in the network and in terms of the eigenvectors of the network.

Ex. 6.2. The Brain State in the Box

One of the flaws of the linear Hebbian associator is that it can "blow up," as you will have seen in Ex. 6.1. You can overcome this limitation, however, by using the brain-state-in-the-box model; that is, by simply stipulating that units have maximum and minimum activations that cannot be exceeded. Our implementation arbitrarily sets these as +1.0 and −1.0. You can implement this model by using the following command:

aa: *set mode bsb 1*

Under these circumstances, it is more interesting to start with weaker external inputs so that they have some room to grow before they are clipped off at the corners. You can make the external inputs weaker by using the *estr* parameter. For example, setting it to 0.1 will mean that an external input specified as having a magnitude of 1.0 will actually only add 0.1 to the net input of the unit receiving it. Note that the *extinput* column in the display gives the external input specification *before* multiplication by *estr*.

The file *bsb8.str* turns on *bsb* and sets *estr* to 0.1, so you can set up for this exercise by restarting the program with the command:

aa lh8.tem bsb8.str

Completion and rectification. In this part of the exercise, you will see how the BSB model's corner-seeking characteristics lead to the pattern completion and rectification capabilities of this type of auto-associative network. First, train your BSB network with the two patterns in the file *two.pat*. Run the network for two training epochs. (It's easiest to just run the *strain* command twice.) At this point the weights should have values of +0.5, −0.5, and 0.0. You may notice what looks like an anomaly; the external activation is shown as +1.0 (100) and −1.0 (reverse video 100), but the internal input is ±0.1, and the activation is only ±0.2. This is because the *estr* parameter is set to 0.1. Thus, external inputs with an absolute value of 1.0 only add 0.1 to the net input.

At this point, you should set *ncycles* to 8. Now you're ready to do the following tests.

Q.6.2.1. Use the *tall* command to test the two learned patterns. Describe and explain the time course of build-up of activation of the units.

Explain in terms of the eigenvalues of the stored patterns and in terms of the BSB activation assumptions.

Q.6.2.2. Use the *ctest* command to present the pattern +−+− . . . . (This is pattern *a* with elements 4 through 7 cleared.) Describe the time course and build-up of activation of the units in terms of the similarity of the input pattern to the two stored patterns.

Q.6.2.3. Present the pattern +−+−++−+ using the *test* command. This pattern is identical to pattern *a* except for the last element. Describe the time course and build-up of activation of the units. Specifically explain the activations of units 6 and 7 at the end of cycles 1, 2, 3, 4, 5, and 6.

Prototype learning and self-connections. In this exercise, we will see how the BSB model can be used to learn the prototype of a set of training experiences. For this purpose, reset the weights in the network and read in the patterns in the file *bsb8.pat*. This file contains eight patterns, each made by changing a different bit in the pattern +−+−+−+−. Set *ncycles* back to 2, and train the network for one epoch using these patterns. Note that during this training, the network sees eight different distortions of the prototype and never sees the prototype itself. After training, set *ncycles* back to 8 before testing.

Q.6.2.4. Test the network with one of the eight training patterns and with the prototype pattern (using the *test* command). Describe what happens in each case and explain, referring to the values of the weights. Explain the values of the weights in terms of the correlational character of Hebbian learning.

Ex. 6.3. The Delta Rule in a Linear Associator

This exercise allows you to explore the delta rule in a linear auto-associator. First you will get a chance to develop your own set of training patterns that are not orthogonal yet still "solvable" in an auto-associative network. Then you will get a chance to try to break the network by finding a set of patterns that the network cannot solve.

When learning using the delta rule, there is always a trivial solution to any auto-associator problem as long as there are self-connections of the units: The trivial solution is to make the self-connections large enough so that each unit essentially generates its own internal input to match the external input it receives from the outside. In many cases (as you can demonstrate in the exercises) the network will in fact make use of this situation to set the self-connections to 1.0. For this reason, the delta rule is typically used without self-connections in an auto-associator.

Begin this exercise by constructing a set of three nonorthogonal patterns that are learnable by the network. One possibility is to generate such a set at random, using the *get/ rpatterns* command. Of course, if you do this you will have to check to make sure that they are learnable. Alternatively, you can try constructing a set on your own.

For this exercise you will begin by calling the program with the *dr8.tem* and *.str* files:

aa dr8.tem dr8.str

The template file differs from the one you have been using up to now only in that it displays the *pss* (pattern sum of squares) and *tss* (total sum of squares) measures. The *dr8.str* file sets *linear* mode to 1, but all other modes have their default values; in particular, *hebb* mode and *selfconnect* mode are both off. Since training takes several epochs to run to completion, the file also sets *stepsize* to *pattern* and sets *nepochs* to 10. This means that the display is updated only once for each pattern after cycling and changing the weights. The *ncycles* variable is set to 2. The activation at the end of processing reflects the sum of the external input and the internal input generated by the activations produced by external input on cycle 1. When the internal input matches the external input, the error term is 0. Thus, when then network has "solved" a set of training patterns, the activations will equal twice the external input. Finally, the *lrate* parameter is set to 0.05; large values result in overshooting weight changes, which can lead to disaster.

The linear predictability constraint. The first exercise is to generate your set of three nonorthogonal but still learnable patterns, and present them to the network for learning. Study what happens to the *pss* and *tss* measures (you may want to set *single* to 1 to do this). Run the learning process to the point where the *tss* becomes less than 0.001 (*strain* terminates when this occurs since *ecrit* is set to 0.001), and compare the resulting weights to those that you obtain if you teach the network the same set of patterns for 10 epochs using the Hebb rule.

Q.6.3.1. Display your three patterns, and discuss the two weight matrices you obtained. In what ways is the matrix obtained using the delta rule similar to the matrix obtained using Hebbian learning? How are they different? Explain.

Hints. We supply a set of patterns in *dr8.pat*, which you may use if you wish. One of the things you will observe in the delta rule solution is that the weights on each row of the weight matrix add up to 0.99; when the problem is completely solved, and if you could see all the decimal places, they would add up to 1.0. This is not the case in the Hebbian matrix. Your answer should include an explanation of this fact.

An unsolvable set of patterns. This exercise is a little more difficult. The task is to construct a set of patterns such that there is no set of weights that will reduce the *tss* to 0. You will want to test your set of patterns, of course, to be sure that it cannot be solved. Run *strain* until it is clear that the *tss* is not getting any better. We supply a file called *imposs.pat* containing a set of patterns that cannot be learned, but it is better to make up your own.

Q.6.3.2. Display your set of unsolvable patterns, and explain how you arrived at this set. Then show the set of weights obtained by the network and indicate where the problem lies in solving the set of patterns.

Hints. It is possible to construct unsolvable sets of only two patterns, in which the two patterns are identical except for a single unit; this unit cannot then be predicted from the other units. These cases are somewhat trivial, since they would be unsolvable by any network. More interesting are the cases that involve, say, four patterns. Here, sets can be constructed that could be solved by a learning mechanism that can make use of hidden units, but not by a one-layer associative network. For simplicity, it is worthwhile to focus on constructing a set of patterns in which only one of the units is "unsolvable." It may be necessary to set the *lrate* to 0.01 to avoid wild oscillations of the weights.

Q.6.3.3. What happens if you set up the network so that two units are perfectly correlated with each other but neither can be predicted by any of the other units? Discuss the implications of this for using the auto-associator as a pattern completion device.

Ex. 6.4. Memory for General and Specific Information

Our final exercise with the **aa** program allows you to explore the version of the auto-associator discussed in *PDP:17* and *PDP:25*—the DMA model. You should be able to use the program to set up and replicate all of the simulation experiments described in *PDP:17* and *PDP:25*. We have provided files to allow you to repeat variants of several of the examples discussed in *PDP:17* (pp. 182-192). These examples illustrated the following aspects of the DMA model:

1. The model can extract what appears to be the prototype, or central tendency, of a set of patterns if the patterns are in fact random distortions of the same base or prototype pattern.

2. The model can do this for several different patterns, using the same set of connections to store its knowledge of all of the prototypes.

3. This ability does not depend on the exemplars being presented with labels.

4. Representations of specific repeated exemplars can coexist in the same set of connections with knowledge of the prototype.

The examples discussed in *PDP:17* were formulated on a 24-unit auto-associator. Patterns presented to the network consisted of an eight-unit name field and a 16-unit visual pattern field. Because the weights for a 24-unit associator cannot be displayed conveniently on the screen, the versions of these examples considered here use a 16-unit associator that implements only the visual pattern field from the *PDP:17* examples.

Learning a prototype from exemplars. We have already explored this capability in Ex. 6.2, using a systematic set of distortions. In this instance each distortion is a random variant of the prototype. A single prototype pattern—taken, for concreteness, to represent the typical dog—is distorted randomly on each learning trial by changing the sign of each element of the external input with a probability given by the value of the parameter *pflip*. The weights are adjusted after each pattern according to the delta rule. In *PDP:17*, we imagined that the learning rate parameter was rather large, so that immediately after the weight adjustment the weights reflected the new exemplar, but that each new increment decayed to a fraction of its initial value before the next pattern was presented. Here we simply set the learning rate parameter to a small value so that the weight increments already reflect the assumed decay.

The parameter values used in this simulation are as follows: *estr* = 0.15, *istr* = 0.15, and *decay* = 0.15. The learning rate parameter *lrate* and the distortion rate parameter *pflip* are set initially to 0.01[1] and 0.2, respectively,

[1] Careful reading of the text of *PDP:17* suggests that the example described in the section "Learning a Prototype From Exemplars" would have used a learning rate of $(0.05)(0.85)/(\mathit{nunits}-1)$ which is about 0.002. This is not correct; a value of 0.01 reproduces the results described more closely, so we have used the latter value here. The difference is one of the magnitude of the impact of each distortion to the weights and does not affect the qualitative character of the results.

but the experiments we suggest involve manipulating these parameters. Also, the number of processing cycles run in processing each pattern (*ncycles*) is set to 10 for these examples. Larger values will allow the network to settle closer to asymptotic activation values but do not affect the results materially so we use a smaller value to save computing time.

The example is carried out using the *dma.tem* and *dma.str* files to set up the screen layout and set the parameters to the appropriate values. Then the file *dog.pat* is read in using the *get/ patterns* command. The screen layout is rather cramped so that all of the information displayed previously still fits. To train the network with a series of distorted exemplars to the *dog* prototype, simply enter the *strain* command. You will want to enter it twice to complete 50 epochs. The prototype and an approximate characterization of the weights that should result are shown in Figure 3. Note that the

```
Prototype pattern:

                + − + + − − − − + + + + + − − −

Weights acquired after learning:

  . .   . + . . . .     . . . .     .   . . .
.   . .   .     .   . . . .       . .     . .
. .   − . − . . . . . . . . +     . . . .     .
. . −   . . . .     . . . . − . .   . . . . . .
.   . .   . . . .     .   . .   . . . . . .
. . − . .   . − . . . .     − . . . . . .     .
.   . . . .   . .   . .     . . .   .   . . . .
. .   . . . .   . . .     . . . .   .   .   . .
.   . . . . .     − + . . . . − . + + . . . . .
. . . . .   . . .   − − + . . + − − − − . + + +
    .   . . . . + −   + − . − − + + + + + − − −
  .     . . .   . − .   − − . − + + + + . − − −
.       . .   . . + − −   + + + − − − − . + + +
.   . .   .   . . + . − +   . + − − − − − + . +
. . . .   . . . . + − − + .   + − − − − . + + +
  . .   .   . . − + − − + + +   − − − − − + + +
. . . .     . . . − . + − − . −   + + + . − − −
  . . . . .     . − + + − − − − +   + + . − − −
. . . . . . . . + − . + − − . − + +   + . − − −
  . .   . .     . − + + − . . − + + +   + − . −
. . . . . . . . . − + + − − . − . + + +   − . −
. . . .   . .   . + . − + + . + − − − − −   + +
. . . . . .   .   .   . .   . . . − . .   .   +
  . . . . . .   . + . − + . . + − − − − . + +
```

FIGURE 3. Weights acquired in learning from distorted exemplars of a prototype. (The prototype pattern is shown above the weight matrix. Blank entries correspond to weights with absolute values less than 0.01; dots correspond to absolute values less than 0.06; pluses or minuses are used for weights with larger absolute values.) (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.)

display you will see on your screen corresponds to the lower left 16×16 entries in this figure.

At the end of 50 epochs, test the network using the *test* command. Test it on the last exemplar studied (using the *L* option with *test*), on the prototype (by entering *#dog* to the prompt), and on two or three new distortions (by entering *?dog* to the prompt). Pay particular attention to the *ndp* measure. Then set *nepochs* to 1 and run several more individual epochs of training, testing the model as just described after each epoch.

Q.6.4.1. Compare the weights you obtain to the results shown in Figure 3. Are they qualitatively similar? Also describe the results of the tests. Discuss the sense in which this model is sensitive both to the prototype and to recent, specific exemplars.

Q.6.4.2. Repeat the experiment using larger and smaller values of *lrate* and larger and smaller values of *pflip*. Describe the results and explain. Note that the network will repeat the exact same series of distortions on successive runs in which *pflip* is not changed if reinitialization is done with *reset*; a new series of distortions will occur if *newstart* is used to reinitialize. If *pflip* is decreased (or increased) and *reset* is used, the distortions should be a subset (or superset, respectively) of the distortions from the previous run.

Learning several categories without labels. This example allows you to replicate the dog-cat-bagel example discussed on pages 184-189 of *PDP:17*. Once again the name field will not be used, so that the experiments will be most comparable to those discussed in the section "Category Learning Without Labels" in *PDP:17*.

In the dog-cat-bagel example, there are three prototypes: the same one you have been using as the *dog* and two new ones—the *cat*, similar to *dog*, and the *bagel*, orthogonal to both of these. The network is trained using distortions of each of these, with *pflip* set to 0.1. The prototypes and the weights that resulted from training with both name and visual patterns are shown in Figure 4, and the results of learning the three prototypes without names are reproduced in Table 1.

To run the exercise, increase *nepochs* and set the *lrate* and *pflip* parameters to 0.01 and 0.1, respectively. Use *newstart* to reinitialize the network, and read in the three patterns from the file *dcb.pat* using the *get/ patterns* command. Then run 50 training epochs.

Q.6.4.3. Compare the weights you obtain to those shown in the lower right portion of Figure 4. Are there any systematic differences? Why might such differences occur?

Q.6.4.4. Use the *test* command to enter the probes shown in Table 1, and compare the pattern of activation you obtain to that shown in the

Prototypes

```
Dog:
+ − + − + − + −     + − + + − − − − + + + + + − − −
Cat:
+ + − − + + − −     + − + + − − − − + − + − + + − +
Bagel:
+ − − + + − − +     + + − + − + + − + − − + + + + −
```

Weights

```
      -1  -1     +2*  -1      -1      +3          +5  -2  -1      -3  +2  -1          +2          -1
          -2      -1 +4*              +1  -1                  -1  -1  +1  -1  -1      -3  -1      -3  +3
  -1  -2      -1          +5                  +1      +1  -1  -1      -1  +4      +1      -5      -1
          -1      -1      -1  +2      -1  +3  -3          +2  +3          -1  -2      -1  +1  +3
  +3  -1  -1                  +1          -1      +3  -3          -2  +2          +1  +3  +1  -1  -1
  -1  +5  -3              -2          -1  -1      -1      -1          -1  -2  +1  -5  -1  +2  -1  +5
  -1      +3  -1  -1  -2                      +1          -1      +1      +4          -1  -2
  -1  -1  -1  +2          -1          -1  +3  -2  -1      +3  +3      -1  -1  -3  +1  -1  +3  +3

  +3      -1  -1  +2  -1  -1                      +3  -2          -3  +2      +1      +3  +1      -1
  -1  -1  -1  +3      -1      +3              -3  -1      +3  +3  +1  -1  -2  -2  +1  -1      +2
  +1  +1  +1  -3              -2      +1  -3          -1  -3  -3          +1  +3  -1      -1  -2  +2
  +2  -1  -1  -1  +3      -1          +3  -1  +1      -2  -1      -2  +2  -1          +2  +1  -1
  -2  +1  +1      -2  +1  +1  +1      -3          -3          +1  +2  -3          -1  -2  -1  +1
      -1      +2  -1          +2      -1  +2  -2      +1      +2  +1  -1      -3      -1      +2
  -1  -1  -1  +2      -1  -1  +2          +2  -3          +3      +1          -3      -1      +3
  -2  +1  +1  +1  -3      +1  +1      -2  +1  -1  -3  +3              -2  +1  -1  -1  -3  -1  +1
  +3      -1      +2      -1          +2          +3  -3          -3      -1  +1      +3  +1  -1  -1
          +5  -1  -1  -1  +4  -1                      +1      -3  +1          +1      -1  -5  -1  -1
      +1      -2      +1  +1  -3      +1  -2  +2  +1  -1  -3  -3  -1      +1      -2          -3  +1
  +1  -5  +2  +1  +1  -5  +3                  -2  +1  -1  +1          +1  +1  -2      +1  -2  +1  -3
  +2  -1  -1      +3  -1              +2      +1  +2  -2          -3  +2      +1
      +3  -4  +1      +3  -4  +1      +1      -1          +1  +1      +1  -5  -1  -3          +1  +2
              +2  -1      -1          -1  +2  -3          +2  +2          -1  -3          +1
  -1  +3  -2  -1  -1  +5  -3              -1  +3      +1  -1          -1      +1  -2      +3  -1
```

FIGURE 4. Weights acquired in learning the three prototype patterns shown. (Blanks in the matrix of weights correspond to weights with absolute values less than or equal to 0.05. Otherwise the actual value of the weight is about 0.05 times the value shown; thus +5 stands for a weight of +0.25. The gap in the horizontal and vertical dimensions is used to separate the name field from the visual pattern field.) (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.)

TABLE 1

RESULTS OF TESTS AFTER LEARNING THE DOG, CAT, AND BAGEL PATTERNS WITHOUT NAMES

Dog visual pattern:	+	−	+	+	−	−	−	−	+	+	+	+	+	−	−	−
Probe:									+	+	+	+				
Response:	+3	−3	+3	+3	−3	−4	−3	−3	+6	+5	+6	+5	+3	−2	−3	−2
Cat visual pattern:	+	−	+	+	−	−	−	−	+	−	+	−	+	+	−	+
Probe:									+	−	+	−				
Response:	+3	−3	+3	+3	−3	−3	−3	−3	+6	−5	+6	−5	+3	+2	−3	+2
Bagel visual pattern:	+	+	−	+	−	+	+	−	+	−	−	+	+	+	+	−
Probe:									+	−	−	+				
Response:	+2	+3	−4	+3	−3	+3	+3	−3	+6	−6	−6	+6	+3	+3	+3	−3

Note. 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.

table. What happens when you increase *ncycles* to 50? Repeat the experiment, using other portions of the patterns as probes. How well do you feel the model does in completion of the prototypes compared to what you might expect from human subjects?

Hints. To do these tests, you must enter *E* to the prompt presented by the *test* routine, and then enter the pattern you wish to test, with dot (.) or 0 in each location that should be blank in the probe. The entries should be separated by spaces and terminated with the word *end* or an extra *return*.

Coexistence of the prototype and repeated exemplars. Our last experiment involves examining the model's ability to retain both prototypes and frequently recurring exemplars. The example is modeled on the one described in *PDP:17* on pages 189-191. In that example, we assumed the model sees several distortions of the same prototype, intermixed with presentations of two repeated examples. The patterns came with names in this example; each new distortion of the prototype was just called *dog*, but the repeated examples were called *Fido* and *Rover*.

For this example, we imagine that the 16-unit network consists of an eight-unit name field and an eight-unit visual pattern field, and we use the name pattern and the first and last groups of four elements from the visual patterns used in *PDP:17*. The original patterns are shown in Table 2. The portion of the visual pattern that is used in the present example is underlined. Note that the elements that differ in the original between the

TABLE 2

RESULTS OF TESTS WITH PROTOTYPE AND SPECIFIC EXEMPLAR PATTERNS

	Name Pattern	Visual Pattern
Pattern for dog prototype	+ − + − + − + −	+ − + + − − − − + + + + + − − −
Response to prototype name		+4 −5 +3 +3 −4 −3 −3 −3 +3 +3 +4 +3 +4 −3 −4 −4
Response to prototype visual pattern	+5 −4 +4 −4 +5 −4 +4 −4	
Pattern for Fido exemplar	+ − − − + − − −	+ − (−) + − − − − + + + + + (+) − −
Response to Fido name		+4 −4 −4 +4 −4 −4 −4 −4 +4 +4 +4 +4 +4 +4 −4 −4
Response to Fido visual pattern	+5 −5 −3 −5 +4 −5 −3 −5	
Pattern for Rover exemplar	+ − − + + + − +	+ (+) + + − − − − + + + + + − − −
Response to Rover name		+4 +5 +4 +4 −4 −4 −4 −4 +4 +4 +4 +4 +4 −4 −4 −4
Response to Rover visual pattern	+4 −4 −2 +4 +4 +4 −2 +4	

Note. Underlined portions of vectors are those used in the patterns stored in the file *dfr.pat*. Elements in parentheses are those that distinguish the visual pattern for each repeated exemplar (Fido and Rover) from the prototype. (Adapted 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.)

prototype and the two exemplars are retained in the shorter versions used here. In the version of this example we consider here, distortions apply to both parts of the pattern (name and visual parts) and occur on the repeated exemplars, as well as the prototype itself. This changes the results in some details, mainly increasing the number of epochs required for adequate learning, but the basic qualitative results are the same as reported in *PDP:17*.

This example is run with the same parameters as the previous one, so all you have to do is reinitialize the network and read in the *dog*, *Fido*, and *Rover* patterns from the file *dfr.pat*. (If you start again from scratch, do not forget to set *pflip* to 0.1.) Use *strain* (or *ptrain*, if you prefer) to run 50 epochs of training, and then test the network using the *ctest* routine, testing first for the network's ability to fill in the visual pattern from the name and then testing the network's ability to fill in the name pattern from the visual pattern. (Set *ncycles* to 50 for these tests.) These tests require clearing elements 8 through 15 and 0 through 7, respectively. For example, to test for the network's ability to fill in the visual pattern for *dog*, you would enter:

```
ctest dog 8 15
```

Set *ncycles* back to 10, run another 50 epochs, set *ncycles* to 50 again, and test again.

Q.6.4.5. How close do you come to reproducing the results shown in Table 2 at the end of 50 epochs? At the end of 100 epochs? In general, what do you see as the strengths and weaknesses of this approach to storing and retrieving representations of categories and exemplars?

COMPETITIVE LEARNING

In Chapter 5 we showed that multilayer, nonlinear networks are essential for the solution of many problems. We showed one way, the back propagation of error, that a system can learn appropriate features for the solution of these difficult problems. This represents the basic strategy of pattern association—to search out a representation that will allow the computation of a specified function. There is a second way to find useful internal features: through the use of a *regularity detector,* a device that discovers useful features based on the stimulus ensemble and some a priori notion of what is important. The competitive learning mechanism described in *PDP:5* is one such regularity detector. In this section we describe the basic concept of competitive learning, show how it is implemented in the **cl** program, describe the basic operations of the program, and give a few exercises designed to familiarize the reader with these ideas.

BACKGROUND

The basic architecture of a competitive learning system (illustrated in Figure 5) is a common one. It consists of a set of hierarchically layered units in which each layer connects, via excitatory connections, with the layer immediately above it, and has inhibitory connections to units in its own layer. In the most general case, each unit in a layer receives an input from each unit in the layer immediately below it and projects to each unit in the layer immediately above it. Moreover, within a layer, the units are broken into a set of inhibitory clusters in which all elements within a cluster inhibit all other elements in the cluster. Thus the elements within a cluster at one level compete with one another to respond to the pattern appearing on the layer below. The more strongly any particular unit responds to an incoming stimulus, the more it shuts down the other members of its cluster.

There are many variants to the basic competitive learning model. Von der Malsburg (1973), Fukushima (1975), and Grossberg (1976), among others, have developed competitive learning models. In this section we describe the simplest of the many variations. The version we describe was first proposed by Grossberg (1976) and is the one studied by Rumelhart and Zipser in *PDP:5*. This version of competitive learning has the following properties:

- The units in a given layer are broken into several sets of nonoverlapping clusters. Each unit within a cluster inhibits every other unit within a cluster. Within each cluster, the unit receiving the largest input achieves its maximum value while all other units in the cluster are pushed to their minimum value.[2] We have arbitrarily set the maximum value to 1 and the minimum value to 0.

- Every unit in every cluster receives inputs from all members of the same set of input units.

- A unit learns if and only if it wins the competition with other units in its cluster.

[2] A simple circuit, employed by Grossberg (1976) for achieving this result, is attained by having each unit activate itself and inhibit its neighbors. Such a network can readily be employed to *choose* the maximum value of a set of units. In our simulations, we do not use this mechanism. We simply compute the maximum value directly.

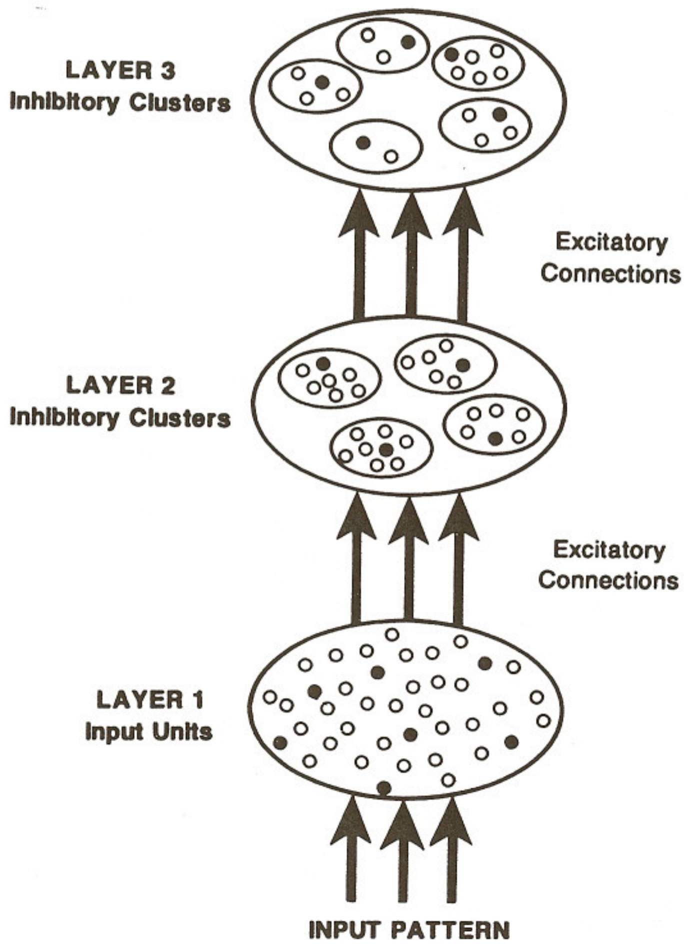


FIGURE 5. The architecture of the competitive learning mechanism. Competitive learning takes place in a context of sets of hierarchically layered units. Units are represented in the diagram as dots. Units may be active or inactive. Active units are represented by filled dots, inactive ones by open dots. In general, a unit in a given layer can receive inputs from all of the units in the next lower layer and can project outputs to all of the units in the next higher layer. Connections between layers are excitatory and connections within layers are inhibitory. Each layer consists of a set of clusters of mutually inhibitory units. The units within a cluster inhibit one another in such a way that only one unit per cluster may be active. We think of the configuration of active units on any given layer as representing the input pattern for the next higher level. There can be an arbitrary number of such layers. A given cluster contains a fixed number of units, but different clusters can have different numbers of units. (From "Feature Discovery by Competitive Learning" by D. E. Rumelhart and D. Zipser, 1985, *Cognitive Science*, 9, 75-112. Copyright 1985 by Ablex Publishing. Reprinted by permission.)

- A stimulus pattern S_j consists of a binary pattern in which each element of the pattern is either *active* or *inactive*. An active element is assigned the value 1 and an inactive element is assigned the value 0.

- Each unit has a fixed amount of weight (all weights are positive) that is distributed among its input lines. The weight on the line connecting to unit i on the upper layer from unit j on the lower layer is designated w_{ij}. The fixed total amount of weight for unit j is designated $\sum_j w_{ij} = 1$. A unit learns by shifting weight from its inactive to its active input lines. If a unit does not respond to a particular pattern, no learning takes place in that unit. If a unit wins the competition, then each of its input lines gives up some portion ϵ of its weight and that weight is then distributed equally among the active input lines. Mathematically, this learning rule can be stated

$$\Delta w_{ij} = \begin{cases} 0 & \text{if unit } i \text{ loses on stimulus } k \\ \epsilon \dfrac{active_{jk}}{nactive_k} - \epsilon w_{ij} & \text{if unit } i \text{ wins on stimulus } k \end{cases}$$

where $active_{jk}$ is equal to 1 if in stimulus pattern S_k, unit j in the lower layer is active and is zero otherwise, and $nactive_k$ is the number of active units in pattern S_k (thus $nactive_k = \sum_j active_{jk}$).[3]

Figure 6 illustrates a useful geometric analogy to this system. We can consider each stimulus pattern as a vector. If all patterns contain the same number of active lines, then all vectors are the same length and each can be viewed as a point on an N-dimensional hypersphere, where N is the number of units in the lower level, and therefore, also the number of input lines received by each unit in the upper level. Each $\times$ in Figure 6A represents a particular pattern. Those patterns that are very similar are near one another on the sphere, and those that are very different are far from one another on the sphere. Note that since there are N input lines to each unit in the upper layer, its weights can also be considered a vector in N-dimensional space. Since all units have the same total quantity of weight, we have N-dimensional vectors of approximately fixed length for each unit

[3] Note that for consistency with the other chapters in this book we have adopted terminology here that is different from that used in the *PDP:5*. Here we use ϵ where g was used in *PDP:5*. Also, here the weight to unit i from unit j is designated w_{ij}. In *PDP:5*, i indexed the sender not the receiver, so w_{ij} referred to the weight from unit i to unit j.

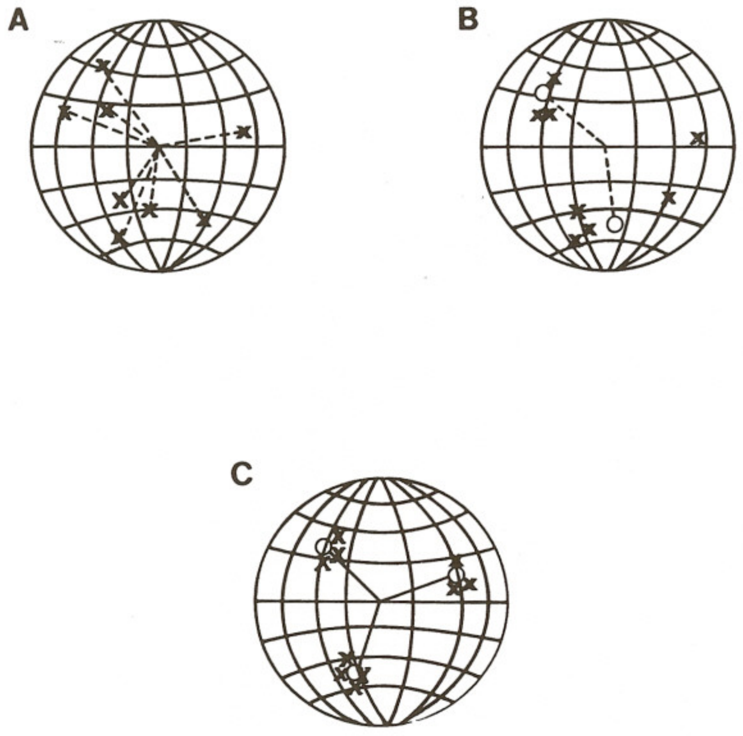


FIGURE 6. A geometric interpretation of competitive learning. *A:* It is useful to conceptualize stimulus patterns as vectors whose tips all lie on the surface of a hypersphere. We can then directly see the similarity among stimulus patterns as distance between the points on the sphere. In the figure, a stimulus pattern is represented as an ×. The figure represents a population of eight stimulus patterns. There are two clusters of three patterns and two stimulus patterns that are rather distinct from the others. *B:* It is also useful to represent the weights of units as vectors falling on the surface of the same hypersphere. Weight vectors are represented in the figure as ○'s. The figure illustrates the weights of two units falling on rather different parts of the sphere. The response rule of this model is equivalent to the rule that whenever a stimulus pattern is presented, the unit whose weight vector is closest to that stimulus pattern on the sphere wins the competition. In the figure, one unit would respond to the cluster in the northern hemisphere and the other unit would respond to the rest of the stimulus patterns. *C:* The learning rule of this model is roughly equivalent to the rule that whenever a unit wins the competition (i.e., is closest to the stimulus pattern), that weight vector is moved toward the presented stimulus. The figure shows a case in which there are three units in the cluster and three natural groupings of the stimulus patterns. In this case, the weight vectors for the three units will each migrate toward one of the stimulus groups. (From "Feature Discovery by Competitive Learning" by D. E. Rumelhart and D. Zipser, 1985, *Cognitive Science*, 9, 75-112. Copyright 1985 by Ablex Publishing. Reprinted by permission.)

in the cluster.[4] Thus, properly scaled, the weights themselves form a set of vectors that (approximately) fall on the surface of the same hypersphere. In Figure 6B, the ○'s represent the weights of two units superimposed on the same sphere with the stimulus patterns. Whenever a stimulus pattern is presented, the unit that responds most strongly is simply the one whose weight vector is nearest that for the stimulus. The learning rule specifies that whenever a unit wins a competition for a stimulus pattern, it moves a fraction ϵ of the way from its current location toward the location of the stimulus pattern on the hypersphere. Suppose that the input patterns fell into some number, M, of "natural" groupings. Further, suppose that an inhibitory cluster receiving inputs from these stimuli contained exactly M units (as in Figure 6C). After sufficient training, and assuming that the stimulus groupings are sufficiently distinct, we expect to find one of the vectors for the M units placed roughly in the center of each of the stimulus groupings. In this case, the units have come to detect the grouping to which the input patterns belong. In this sense, they have "discovered" the structure of the input pattern sets.

Some Features of Competitive Learning

There are several characteristics of a competitive learning mechanism that make it an interesting candidate for study, for example:

- Each cluster classifies the stimulus set into M groups, one for each unit in the cluster. Each of the units captures roughly an equal number of stimulus patterns. It is possible to consider a cluster as forming an M-valued feature in which every stimulus pattern is classified as having exactly one of the M possible values of this feature. Thus, a cluster containing two units acts as a binary feature detector. One element of the cluster responds when a particular feature is present in the stimulus pattern, otherwise the other element responds.

- If there is structure in the stimulus patterns, the units will break up the patterns along structurally relevant lines. Roughly speaking, this means that the system will find clusters if they are there.

[4] It should be noted that this geometric interpretation is only approximate. We have used the constraint that $\sum_j w_{ij} = 1$ rather than the constraint that $\sum_j w_{ij}^2 = 1$. This latter constraint would ensure that all vectors are in fact the same length. Our assumption only assures that they will be approximately the same length.

- If the stimuli are highly structured, the classifications are highly stable. If the stimuli are less well structured, the classifications are more variable, and a given stimulus pattern will be responded to first by one and then by another member of the cluster. In our experiments, we started the weight vectors in random directions and presented the stimuli randomly. In this case, there is rapid movement as the system reaches a relatively stable configuration (such as one with a unit roughly in the center of each cluster of stimulus patterns). These configurations can be more or less stable. For example, if the stimulus points do not actually fall into nice clusters, then the configurations will be relatively unstable and the presentation of each stimulus will modify the pattern of responding so that the system will undergo continual evolution. On the other hand, if the stimulus patterns fall rather nicely into clusters, then the system will become very stable in the sense that the same units will always respond to the same stimuli.[5]

- The particular grouping done by a particular cluster depends on the starting value of the weights and the sequence of stimulus patterns actually presented. A large number of clusters, each receiving inputs from the same input lines can, in general, classify the inputs into a large number of different groupings or, alternatively, discover a variety of independent features present in the stimulus population. This can provide a kind of distributed representation of the stimulus patterns.

- To a first approximation, the system develops clusters that minimize within-cluster distance, maximize between-cluster distance, and balance the number of patterns captured by each cluster. In general, tradeoffs must be made among these various forces and the system selects one of these tradeoffs.

IMPLEMENTATION

The competitive learning model is implemented in the **cl** program. The model implements a single input (or lower level) layer of units, each connected to all members of a single output (or upper level) layer of units. The basic strategy for the **cl** program is the same as for **bp** and the other learning programs. Learning occurs as follows: A pattern is chosen and the

[5] Grossberg (1976) has addressed this problem in his very similar system. He has proved that if the patterns are sufficiently sparse and/or when there are enough units in the cluster, then a system such as this will find a perfectly stable classification. He also points out that when these conditions do not hold, the classification can be unstable. Most of our work is with cases in which there is no perfectly stable classification and the number of patterns is much larger than the number of units in the inhibitory clusters.

pattern of activation specified by the input pattern is clamped on the input units. Next, the net input into each of the output units is computed. The output unit with the largest input is determined to be the winner and its activation value is set to 1. All other units have their activation values set to 0. The routine that carries out this computation is

```
compute_output() {

/* initialize all output units */
  for (i = ninputs; i < nunits; i++) {
    netinput[i] = 0.0;
    activation[i] = 0.0;
  }

/* compute the netinput for each output unit i */
  for (j = 0; j < ninputs; j++) {
    if (activation[j]) {
      for (i = ninputs; i < nunits; i++) {
        netinput[i] += weight[i][j];
      }
    }
  }

/* find the winner */
  for (winner = ninputs, i = ninputs; i < nunits; i++) {
    if (netinput[winner] < netinput[i]) {
      winner = i;
    }
  }

/* set the winner's activation to 1.0 */
  activation[winner] = 1.0;
}
```

After the activation values are determined for each of the output units, the weights must be adjusted according to the learning rule. This involves increasing the weights from the active input lines to the winner and decreasing the weights from the inactive lines to the winner in such a way that the total amount of weight is kept equal to 1.0. This is done by the following routine:

```
change_weights()
{

/* first we determine how many input lines are on */
  for (j = 0; j < ninputs; j++) {
    if (activation[j])
      nactive += 1;
  }
/* if no input lines are on no learning takes place */
  if(nactive == 0) return;

/* otherwise, we adjust the winner's weights */
  for (j = 0; j < ninputs; j++) {
    weight[winner][j] +=
      lrate*(activation[j]/nactive) -
        lrate * weight[winner][j];
  }
}
```

RUNNING THE PROGRAM

The **cl** program is run in much the same way as the programs already described. In general, the program is called with a *.tem* file and a *.str* file. Because of the simplicity of the **cl** architecture (each input unit is connected to each output unit and the output units form a single inhibitory cluster), a *.net* file is not needed; instead, *ninputs* and *noutputs* are defined near the top of the *.str* file. This leads the program to create a network of *ninputs* input units connected to a cluster of *noutputs* output units. The connections are all positive and sum to 1. Generally, a *.pat* file is used to specify a list of patterns for use in training and testing.

The facilities for training and testing are the same as those used in the other learning 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. Since there is no teacher, there is no total sum of squares or error criterion.

Commands

The commands in **cl** are a subset of those for **bp** and therefore need no further explication.

Variables

The following list mentions only those variables that are new or changed in the **cl** program.

stepsize

The default *stepsize* in **cl** is *epoch*; this means that a step consists of going through each of the input patterns, presenting the pattern, computing the activations, and determining the winner, and, if *lflag* is set, changing the weights on the winner; then, after this has been done for each of the patterns, displaying the relevant variables on the screen. Other possible values of stepsize are *pattern*, which causes updating/pausing to occur after each pattern presentation, and *nepochs*, which causes updating/pausing to occur only at the end of *nepochs*.

param/ lrate

Determines the percentage of the winner's weight that is redistributed on each learning trial.

OVERVIEW OF EXERCISES

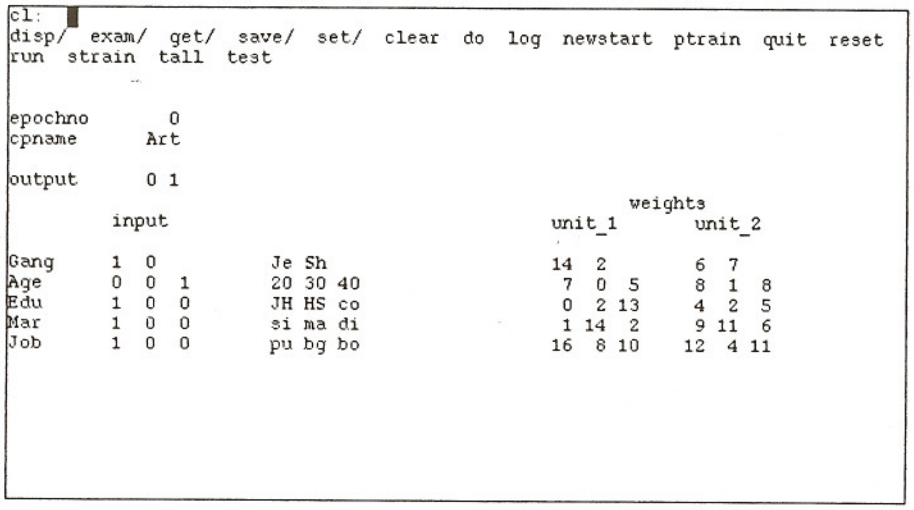
We provide two exercises for the **cl** program. The first uses the Jets and Sharks data base to explore the basic characteristics of competitive learning. The second applies competitive learning to the difficult problem of graph partitioning. A special case of this is the *dipole* problem, considered at the end of Ex. 6.6.

Ex. 6.5. Clustering the Jets and Sharks

The Jets and Sharks data base provides a useful context for studying the clustering features of competitive learning. We have prepared the files *2jets.tem*, *2jets.str*, *jets.pat*, and a couple of *.loo* files for this example. The file *jets.pat* contains the feature specifications for the 27 gang members. (The *2* in the name *2jets.tem* indicates that the network has an output cluster of two units.) The pattern file is set up as follows: The first column contains the name of each individual. The next two tell whether the individual is a Shark or a Jet, the next three columns correspond to the age of the individual, and so on. Note that there are no inputs corresponding to name units; the name only serves as a label for the convenience of the user. To run the program type

```
cl 2jets.tem 2jets.str
```

The resulting screen display (shown in Figure 7) shows the epoch number, the name of the current pattern, the output vector, the inputs, and the weights from the input units to each of the output units. Between the inputs and the weights is a display indicating the labels of each feature.



```
cl:
disp/  exam/  get/  save/  set/  clear  do  log  newstart  ptrain  quit  reset
run  strain  tall  test


epochno       0
cpname       Art

output       0 1
                                                                weights
         input                                            unit_1      unit_2

Gang     1  0             Je Sh                           14  2        6  7
Age      0  0  1          20 30 40                         7  0  5     8  1  8
Edu      1  0  0          JH HS co                         0  2 13     4  2  5
Mar      1  0  0          si ma di                         1 14  2     9 11  6
Job      1  0  0          pu bg bo                        16  8 10    12  4 11
```

FIGURE 7. Initial screen display for the **cl** program running the Jets and Sharks example with two output units.

The inputs and weights are configured in a manner that mirrors the structure of the features. In this case, the pattern for Art is the current pattern. The first row of inputs indicate the gang to which the individual belongs. In the case of Art, we have a 1 on the left and a 0 on the right. This represents the fact that Art is a Jet and not a Shark. Note that there is at most one 1 in each row. This results from the fact that the values on the various dimensions are mutually exclusive. Art has a 1 for the third value of the *Age* row, indicating that Art is in his 40s. The rest of the values are similarly interpreted. The weights are in the same configuration as the inputs. The corresponding weight value is written below each of the two output unit labels (*unit_1* and *unit_2*). Each cell contains the weight from the corresponding input unit to that output unit. Thus the upper left-hand value for the weights is the initial weight from the *Jet* unit to output unit 1. Similarly, the lower right-hand value of the weight matrix is the initial weight from *bookie* to unit 2. The initial values of the weights are random, with the constraint that the weights for each unit sum to 1.0. (Due to scaling and roundoff, the actual values displayed should sum to a value somewhat less than 100.) The *lrate* parameter is set to 0.05. This means that on any trial 5% of the winner's weight is redistributed to the active lines. It should be noted that the *2jets.str* file has already read in the *jets.pat* pattern file.

Now try running the program using the *ptrain* command. (Note that *ptrain* is better than *strain* for the competitive learning procedure since the order can have a large effect on exactly what is learned.) Since *nepochs* is set to 20, the system will stop after 20 epochs. Look at the new values of the weights. Try several more runs, using the *newstart* command to reinitialize the system each time. In each case, note the configuration of the weights. You should find that usually one unit gets about 20% of its weight on the *jets* line and none on the *sharks* line, while the other unit shows the opposite pattern.

Q.6.5.1. What does this pattern mean in terms of the system's response to each of the separate patterns? Explain why the system usually falls into this pattern.

Hints. You can find out how the system responds to each subpattern by using the *tall* command and stepping through the set of patterns—noting each time which unit wins on that pattern (this is indicated by the output activation values displayed on the screen).

Q.6.5.2. Examine the values of the weights in the other rows of the weight matrix. Explain the pattern of weights in each row. Explain, for example, why the unit with a large value on the *Jet* input line has the largest weight for the 20s value of age, whereas the unit with a large value on the *Shark* input line has its largest weight for the 30s value of the age row.

Now repeat the problem and run it several more times until it reaches a rather different weight configuration. (This may take several tries.) You might be able to find such a state faster by reducing *lrate* to a smaller value, perhaps 0.02.

Q.5.3. Explain this configuration of weights. What principle is the system now using to classify the input patterns? Why do you suppose reducing the learning rate makes it easier to find an unusual weight pattern?

We have prepared a pattern file, called *ajets.pat*, in which we have deleted explicit information about which gang the individuals represent. Load this file by typing

```
get patterns ajets.pat
```

Q.5.4. Repeat the previous experiments using these patterns. Describe and discuss the differences and similarities.

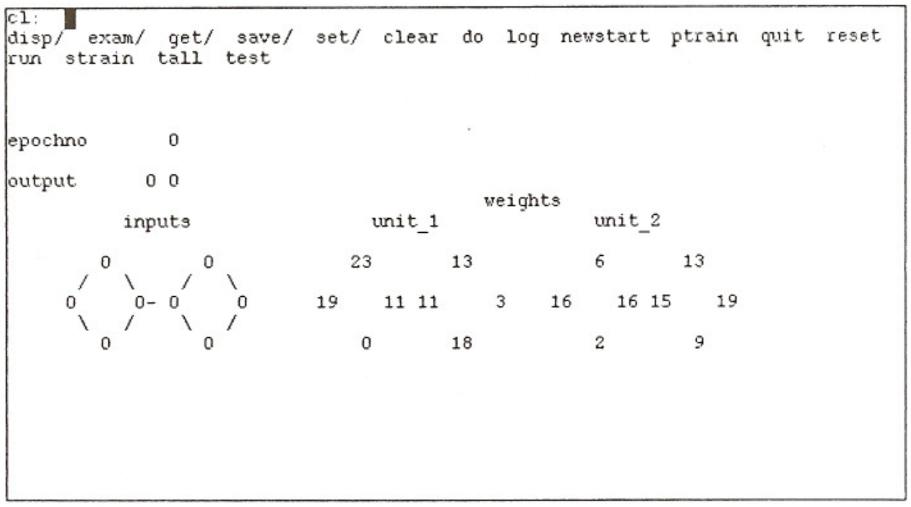
Thus far the system has used two output units and it therefore classified the patterns into two classes. We have prepared a version with three output units. This version can be accessed by the command:

```
cl 3jets.tem 3jets.str
```

Q.6.5.5. Repeat the previous experiments using three output units. Describe and discuss differences and similarities.

Ex. 6.6. Graph Partitioning

Recall that the competitive learning mechanism with n output units has a propensity to put the stimulus patterns in n classes with the classes maximally distinct and the numbers of patterns per class approximately equal. It turns out that there is an interesting and difficult problem, the graph partitioning problem, which requires just that sort of solution. The problem, roughly stated, is this: Given a connected graph of n nodes, each of which is connected to one or more other nodes in the network, divide the graph into two parts with half of the nodes in each while minimizing the number of links that connect nodes from the two different classes. We can map this problem into a problem that competitive learning can work on in the following way. We have two output units, one for each of the two groups into which we are to classify the nodes. There must be n input units, one for each of the nodes in the graph. There is a stimulus pattern for each of the links of the graph. Each stimulus pattern consists of two units on and the rest off. If there is a link from unit i to unit j in the graph, then there is a pattern with units i and j both turned on and the rest turned off. We have prepared a very simple example of this. Files *graph.tem, graph.str, graph.pat*, and so on contain the relevant information. Figure 8 shows the initial screen layout and the graph in question. In this case, the graph



```
cl:
disp/  exam/  get/  save/  set/  clear  do  log  newstart  ptrain  quit  reset
run  strain  tall  test


epochno        0

output        0 0
                                                   weights
          inputs                 unit_1                      unit_2

       0        0               23        13              6        13
     /   \    /   \
    0     0- 0     0       19     11 11      3      16     16 15      19
     \   /    \   /
       0        0                0        18              2         9
```

FIGURE 8. Initial screen display for the graph problem.

consists of two clusters of nodes with a single link between them. The best solution is obviously to separate the two lobes and cut the single link between. Now run the program with the command

cl graph.tem graph.str

Q.6.6.1. Using *ptrain*, run the program several times and note the solutions the system finds. To what degree do these solutions solve the graph partitioning problem? Explain the observed results. Try the same thing with various values for *lrate*. How does this change the results? Why?

Q.6.6.2. Create your own graph and evaluate the results of the network with respect to the graph partitioning problem.

It might be noted that the dipole problem, discussed at length in *PDP:5* (pp. 170-177) is an example of a rather simple graph partitioning problem. In this problem, the patterns consist of pairs of adjacent points on a two-dimensional grid (adjacent points are points that are next to each other on the same row or column). This is equivalent to the graph partioning problem in which each point is connected to every adjacent point. A set of files called *16.tem, 16.str*, and *16.pat* (together with associated *.loo* files) are provided for this example, if you choose to explore it. Just start up the **cl** program with the files *16.tem* and *16.str* and enter *ptrain* to train the network, as in all of the examples already discussed.