associative memory can learn. The complexity of the patterns (the number of components) and the similarity of the input patterns that are associated with significantly different response patterns both play a role. We shall consider a few of these ideas in Section 3.3.4.

3.1 TRAINING ALGORITHMS FOR PATTERN ASSOCIATION

3.1.1 Hebb Rule for Pattern Association

The Hebb rule is the simplest and most common method of determining the weights for an associative memory neural net. It can be used with patterns that are represented as either binary or bipolar vectors. We repeat the algorithm here for input and output training vectors (only a slight extension of that given in the previous chapter) and give the general procedure for finding the weights by outer products. Since we want to consider examples in which the input to the net after training is a pattern that is similar to, but not the same as, one of the training inputs, we denote our training vector pairs as **s**:**t**. We then denote our testing input vector as **x**, which may or may not be the same as one of the training input vectors.

Algorithm

Step 0. Initialize all weights ($i = 1, \ldots, n; j = 1, \ldots, m$):

$$w_{ij} = 0.$$

Step 1. For each input training–target output vector pair **s**:**t**, do Steps 2–4.

Step 2. Set activations for input units to current training input ($i = 1, \ldots, n$):

$$x_i = s_i$$

Step 3. Set activations for output units to current target output ($j = 1, \ldots, m$):

$$y_j = t_j.$$

Step 4. Adjust the weights ($i = 1, \ldots, n; j = 1, \ldots, m$):

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j.$$

The foregoing algorithm is not usually used for this simple form of Hebb learning, since weights can be found immediately, as shown in the next section. However, it illustrates some characteristics of the typical *neural* approach to learning. Algorithms for some modified forms of Hebb learning will be discussed in Chapter 7.

Outer products

The weights found by using the Hebb rule (with all weights initially 0) can also be described in terms of outer products of the input vector–output vector pairs. The outer product of two vectors

$$\mathbf{s} = (s_1, \ldots, s_i, \ldots, s_n)$$

and

$$\mathbf{t} = (t_1, \ldots, t_j, \ldots, t_m)$$

is simply the matrix product of the $n \times 1$ matrix $\mathbf{S} = \mathbf{s}^T$ and the $1 \times m$ matrix $\mathbf{T} = \mathbf{t}$:

$$\mathbf{ST} = \begin{bmatrix} s_1 \\ \vdots \\ s_i \\ \vdots \\ s_n \end{bmatrix} [t_1 \ldots t_j \ldots t_m] = \begin{bmatrix} s_1t_1 & \cdots & s_1t_j & \cdots & s_1t_m \\ \vdots & \cdot & \vdots & \cdot & \vdots \\ s_it_1 & \cdots & s_it_j & \cdots & s_it_m \\ \vdots & \cdot & \vdots & \cdot & \vdots \\ s_nt_1 & \cdots & s_nt_j & \cdots & s_nt_m \end{bmatrix}.$$

This is just the weight matrix to store the association $\mathbf{s}:\mathbf{t}$ found using the Hebb rule.

To store a set of associations $\mathbf{s}(p) : \mathbf{t}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$

and

$$\mathbf{t}(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p)),$$

the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_{p=1}^{P} s_i(p)t_j(p).$$

This is the sum of the outer product matrices required to store each association separately.

In general, we shall use the preceding formula or the more concise vector-matrix form,

$$\mathbf{W} = \sum_{p=1}^{P} \mathbf{s}^T(p)\mathbf{t}(p),$$

to set the weights for a net that uses Hebb learning. This weight matrix is described by a number of authors [see, e.g., Kohonen, 1972, and Anderson, 1972].

Perfect recall versus cross talk

The suitability of the Hebb rule for a particular problem depends on the correlation among the input training vectors. If the input vectors are uncorrelated (orthogonal), the Hebb rule will produce the correct weights, and the response of the

net when tested with one of the training vectors will be perfect recall of the input vector's associated target (scaled by the square of the norm of the input vector in question). If the input vectors are not orthogonal, the response will include a portion of each of their target values. This is commonly called *cross talk*. As shown in some of the examples of Chapter 2, in some cases the cross talk is mild enough that the correct response will still be produced for the stored vectors.

To see why cross talk occurs, recall that two vectors $\mathbf{s}(k)$ and $\mathbf{s}(p)$, $k \neq p$, are orthogonal if their dot product is 0. This can be written in a number of ways, including (if, as we assume, $\mathbf{s}(k)$ and $\mathbf{s}(p)$ are row vectors)

$$\mathbf{s}(k)\mathbf{s}^T(p) = 0,$$

or

$$\sum_{i=1}^{n} s_i(k)s_i(p) = 0.$$

Now consider the weight matrix $\mathbf{W}$, defined as before to store a set of input-target vector pairs. The response of the net (with the identity function for the activation function rather than the threshold function) is $\mathbf{y} = \mathbf{xW}$. If the (testing) input signal is the kth training input vector, i.e., if

$$\mathbf{x} = \mathbf{s}(k),$$

then the response of the net is

$$\begin{aligned}\mathbf{s}(k)\mathbf{W} &= \sum_{p=1}^{P} \mathbf{s}(k)\mathbf{s}^T(p)\mathbf{t}(p) \\ &= \mathbf{s}(k)\mathbf{s}^T(k)\mathbf{t}(k) + \sum_{p \neq k} \mathbf{s}(k)\mathbf{s}^T(p)\mathbf{t}(p).\end{aligned}$$

If $\mathbf{s}(k)$ is orthogonal to $\mathbf{s}(p)$ for $p \neq k$, then there will be no contribution to the response from any of the terms in the summation; the response will then be the target vector $\mathbf{t}(k)$, scaled by the square of the norm of the input vector, i.e., $\mathbf{s}(k)\mathbf{s}^T(k)$.

However, if $\mathbf{s}(k)$ is not orthogonal to the other s-vectors, there will be contributions to the response that involve the target values for each of the vectors to which $\mathbf{s}(k)$ is not orthogonal.

Summary

If a threshold function is applied to the response of a net, as described here, and the cross talk is not too severe, the weights found by the Hebb rule may still be satisfactory. (See Examples 2.7, 2.8, and 3.1.)

Several authors normalize the weights found by the Hebb rule by a factor of $1/n$, where n is the number of units in the system [Hertz, Krogh, & Palmer, 1991; McClelland & Rumelhart, 1988]. As the latter observe, the use of normalization can preserve the interpretation of the weight w_{ij} as representing the cor-

relation between the activation of unit x_i and unit y_j, even when the activations have means that are different from zero.

There are three aspects of a particular association problem that influence whether the Hebb rule will be able to find weights to produce the correct output for all training input patterns. The first is simply whether such weights exist. If the input vectors are linearly independent, they do exist. This is the extension of linear separability to the case of vector outputs. Second, there is the question of correlation between (pairs of) the input training vectors. If the input vectors are uncorrelated (orthogonal), then the weight matrix found by the Hebb rule will give perfect results for each of the training input vectors. Finally, because the weights of the Hebb rule represent the extent to which input units and output units should be "on" or "off" at the same time, patterns in which the input activations are correlated strongly, unit by unit, with the target values will be able to be learned by the Hebb rule. For this reason, the Hebb rule is also known as correlation training or encoding.

3.1.2 Delta Rule for Pattern Association

The delta rule is an iterative learning process introduced by Widrow and Hoff (1960) for the ADALINE neuron (see Chapter 2). The rule can be used for input patterns that are linearly independent but not orthogonal. Mappings in which the input vectors are linearly independent can be solved using single-layer nets as described in this chapter. However, the delta rule is needed to avoid the difficulties of cross talk which may be encountered if a simpler learning rule, such as the Hebb rule, is used. Furthermore, the delta rule will produce the least squares solution when input patterns are not linearly independent [Rumelhart, McClelland, & the PDP Research Group, 1986].

In its original form, as introduced in Chapter 2, the delta rule assumed that the activation function for the output unit was the identity function. A simple extension allows for the use of any differentiable activation function; we shall call this the extended delta rule, since some authors use the term "generalized delta rule" synonymously with "backpropagation" for multilayer nets. The nomenclature we use is as follows:

α	learning rate.
$\mathbf{x}$	training input vector.
$\mathbf{t}$	target output for input vector $\mathbf{x}$.

Original delta rule

The original delta rule, for several output units, was derived in Section 2.4.4. It is repeated here for comparison with the extended delta rule described in the next section. The original rule assumes that the activation function for the output units

is the identity function; or, equivalently, it minimizes the square of the difference between the net input to the output units and the target values. Thus, using y for the computed output for the input vector x, we have

$$y_j = \sum_i x_i w_{ij},$$

and the weight updates are

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(t_j - y_j)x_i \quad (i = 1, \ldots, n; j = 1, \ldots, m).$$

This is often expressed in terms of the weight change

$$\Delta w_{ij} = \alpha(t_j - y_j)x_i,$$

which explains why this training rule is called the delta rule.

Extended delta rule

This minor modification of the delta rule derived in Chapter 2 allows for an arbitrary, differentiable activation function to be applied to the output units. The update for the weight from the Ith input unit to the Jth output unit is

$$\Delta w_{IJ} = \alpha(t_J - y_J)x_I f'(y\_in_J).$$

Derivation. The derivation of the extended delta rule given here follows the discussion in Section 2.4.4 closely. However, we now wish to change the weights to reduce the difference between the computed output and the target value, rather than between the net input to the output unit(s) and the target(s).

The squared error for a particular training pattern is

$$E = \sum_{j=1}^{m} (t_j - y_j)^2.$$

E is a function of all of the weights. The gradient of E is a vector consisting of the partial derivatives of E with respect to each of the weights. This vector gives the direction of most rapid increase in E; the opposite direction gives the direction of most rapid decrease in E. The error can be reduced most rapidly by adjusting the weight w_{IJ} in the direction of $-\partial E/\partial w_{IJ}$.

We now find an explicit formula for $\partial E/\partial w_{IJ}$ for the arbitrary weight w_{IJ}. First note that

$$\frac{\partial E}{\partial w_{IJ}} = \frac{\partial}{\partial w_{IJ}} \sum_{j=1}^{m} (t_j - y_j)^2$$

$$= \frac{\partial}{\partial w_{IJ}} (t_J - y_J)^2,$$

since the weight w_{IJ} only influences the error at output unit Y_J.

Furthermore, using the facts that

$$y\_in_J = \sum_{i=1}^{n} x_i w_{iJ} \quad \text{and}$$

$$y_J = f(y\_in_J),$$

we have

$$\frac{\partial E}{\partial w_{IJ}} = -2(t_J - y_J)\frac{\partial y\_in_J}{\partial w_{IJ}}$$

$$= -2(t_J - y_J)x_I f'(y\_in_J).$$

Thus the local error will be reduced most rapidly (for a given learning rate α) by adjusting the weights according to the delta rule

$$\Delta w_{IJ} = \alpha(t_J - y_J)x_I f'(y\_in_J).$$

3.2 HETEROASSOCIATIVE MEMORY NEURAL NETWORK

Associative memory neural networks are nets in which the weights are determined in such a way that the net can store a set of P pattern associations. Each association is a pair of vectors $(\mathbf{s}(p), \mathbf{t}(p))$, with $p = 1, 2, \ldots, P$. Each vector $\mathbf{s}(p)$ is an n-tuple (has n components), and each $\mathbf{t}(p)$ is an m-tuple. The weights may be found using the Hebb rule (Sections 3.1.1) or the delta rule (Section 3.1.2). In the examples in this section, the Hebb rule is used. The net will find an appropriate output vector that corresponds to an input vector $\mathbf{x}$ that may be either one of the stored patterns $\mathbf{s}(p)$ or a new pattern (such as one of the training patterns corrupted by noise).

3.2.1 Architecture

The architecture of a heteroassociative memory neural network is shown in Figure 3.1.

3.2.2 Application

Procedure

Step 0. Initialize weights using either the Hebb rule (Section 3.1.1) or the delta rule (Section 3.1.2).

Step 1. For each input vector, do Steps 2–4.

Step 2. Set activations for input layer units equal to the current input vector

$$x_i.$$

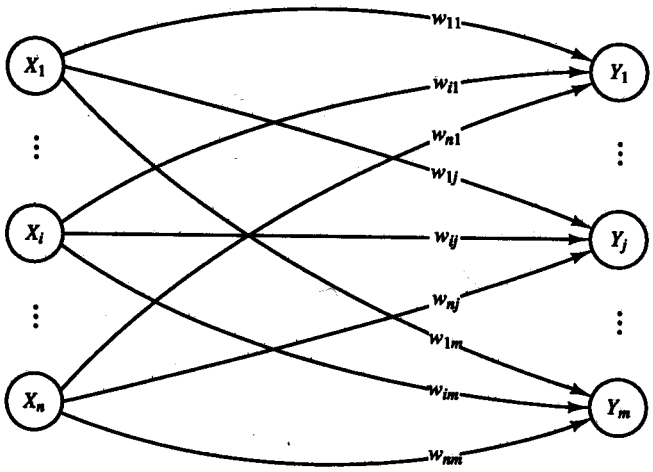

Figure 3.1 Heteroassociative neural net.

Step 3. Compute net input to the output units:

$$y\_in_j = \sum_i x_i w_{ij}.$$

Step 4. Determine the activation of the output units:

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > 0 \\ 0 & \text{if } y\_in_j = 0 \\ -1 & \text{if } y\_in_j < 0 \end{cases},$$

(for bipolar targets).

The output vector **y** gives the pattern associated with the input vector **x**. This heteroassociative memory is not iterative.

Other activation functions can also be used. If the target responses of the net are binary, a suitable activation function is given by

$$f(x) = \begin{cases} 1 & \text{if } x > 0; \\ 0 & \text{if } x \leq 0. \end{cases}$$

A general form of the preceding activation function that includes a threshold θ_j and that is used in the bidirectional associative memory (BAM), an iterative net discussed in Section 3.5, is

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > \theta_j \\ y_j & \text{if } y\_in_j = \theta_j \\ -1 & \text{if } y\_in_j < \theta_j \end{cases}.$$

The choice of the desired response for a neuron if its net input is exactly equal

to the threshold is more or less arbitrary; defining it to be the current activation of the unit Y_j makes more sense for iterative nets; to use it for a feedforward heteroassociative net (as we are discussing here) would require that activations be defined for all units initially (e.g., set to 0). It is also possible to use the perceptron activation function and require that the net input be greater than θ_j for an output of 1 and less than $-\theta_j$ for an output of -1.

If the delta rule is used to set the weights, other activation functions, such as the sigmoids illustrated in Chapter 1, may be appropriate.

Simple examples

Figure 3.2 shows a heteroassociative neural net for a mapping from input vectors with four components to output vectors with two components.

Example 3.1 A Heteroassociative net trained using the Hebb rule: algorithm

Suppose a net is to be trained to store the following mapping from input row vectors $\mathbf{s} = (s_1, s_2, s_3, s_4)$ to output row vectors $\mathbf{t} = (t_1, t_2)$:

		s_1	s_2	s_3	s_4			t_1	t_2
1st	s	(1,	0,	0,	0)	1st	t	(1,	0)
2nd	s	(1,	1,	0,	0)	2nd	t	(1,	0)
3rd	s	(0,	0,	0,	1)	3rd	t	(0,	1)
4th	s	(0,	0,	1,	1)	4th	t	(0,	1)

These target patterns are simple enough that the problem could be considered one in pattern classification; however, the process we describe here does not require that only one of the two output units be "on." Also, the input vectors are not mutually orthogonal. However, because the target values are chosen to be related to the input vectors in a particularly simple manner, the cross talk between the first

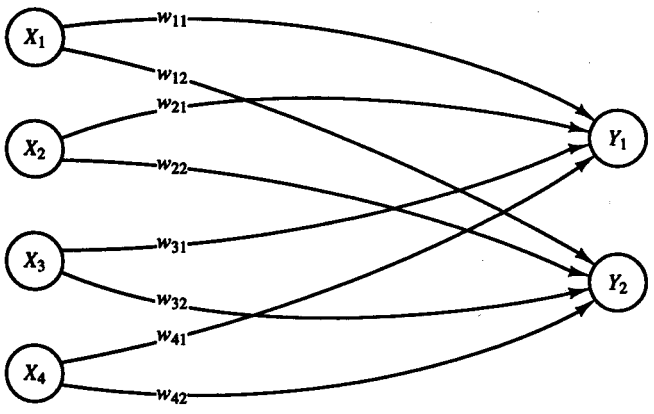

Figure 3.2 Heteroassociative neural net for simple examples.

and second input vectors does not pose any difficulties (since their target values are the same).

The training is accomplished by the Hebb rule, which is defined as

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + s_i t_j; \qquad \text{i.e., } \Delta w_{ij} = s_i t_j.$$

Training

The results of applying the algorithm given in Section 3.1.1 are as follows (only the weights that change at each step of the process are shown):

Step 0. Initialize all weights to 0.

Step 1. For the first **s**:**t** pair (1, 0, 0, 0):(1, 0):

Step 2. $x_1 = 1; \quad x_2 = x_3 = x_4 = 0.$

Step 3. $y_1 = 1; \quad y_2 = 0.$

Step 4. $w_{11}(\text{new}) = w_{11}(\text{old}) + x_1 y_1 = 0 + 1 = 1.$
(All other weights remain 0.)

Step 1. For the second **s**:**t** pair (1, 1, 0, 0):(1, 0):

Step 2. $x_1 = 1; \quad x_2 = 1; \quad x_3 = x_4 = 0.$

Step 3. $y_1 = 1; \quad y_2 = 0.$

Step 4. $w_{11}(\text{new}) = w_{11}(\text{old}) + x_1 y_1 = 1 + 1 = 2;$
$w_{21}(\text{new}) = w_{21}(\text{old}) + x_2 y_1 = 0 + 1 = 1.$
(All other weights remain 0.)

Step 1. For the third **s**:**t** pair (0, 0, 0, 1):(0, 1):

Step 2. $x_1 = x_2 = x_3 = 0; \quad x_4 = 1.$

Step 3. $y_1 = 0; \quad y_2 = 1.$

Step 4. $w_{42}(\text{new}) = w_{42}(\text{old}) + x_4 y_2 = 0 + 1 = 1.$
(All other weights remain unchanged.)

Step 1. For the fourth **s**:**t** pair (0, 0, 1, 1):(0, 1):

Step 2. $x_1 = x_2 = 0; \quad x_3 = 1; \quad x_4 = 1.$

Step 3. $y_1 = 0: \quad y_2 = 1.$

Step 4. $w_{32}(\text{new}) = w_{32}(\text{old}) + x_3 y_2 = 0 + 1 = 1;$
$w_{42}(\text{new}) = w_{42}(\text{old}) + x_4 y_2 = 1 + 1 = 2.$
(All other weights remain unchanged.)

The weight matrix is

$$\mathbf{W} = \begin{bmatrix} 2 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}.$$

Example 3.2 A heteroassociative net trained using the Hebb rule: outer products

This example finds the same weights as in the previous example, but using outer products instead of the algorithm for the Hebb rule. The weight matrix to store the first pattern pair is given by the outer product of the vector

$$\mathbf{s} = (1, 0, 0, 0)$$

and

$$\mathbf{t} = (1, 0).$$

The outer product of a vector pair is simply the matrix product of the training vector written as a column vector (and treated as an $n \times 1$ matrix) and the target vector written as a row vector (and treated as a $1 \times m$ matrix):

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} [1 \quad 0] = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Similarly, to store the second pair,

$$\mathbf{s} = (1, \quad 1, \quad 0, \quad 0)$$

and

$$\mathbf{t} = (1, \quad 0),$$

the weight matrix is

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} [1 \quad 0] = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

To store the third pair,

$$\mathbf{s} = (0, \quad 0, \quad 0, \quad 1)$$

and

$$\mathbf{t} = (0, \quad 1),$$

the weight matrix is

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} [0 \quad 1] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

And to store the fourth pair,

$$\mathbf{s} = (0, \quad 0, \quad 1, \quad 1)$$

and

$$\mathbf{t} = (0, \quad 1),$$

the weight matrix is

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} [0 \quad 1] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}.$$

The weight matrix to store all four pattern pairs is the sum of the weight matrices to store each pattern pair separately, namely,

$$\mathbf{W} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}.$$

Example 3.3 Testing a heteroassociative net using the training input

We now test the ability of the net to produce the correct output for each of the training inputs. The steps are as given in the application procedure at the beginning of this section, using the activation function

$$f(x) = \begin{cases} 1 & \text{if } x > 0; \\ 0 & \text{if } x \leq 0. \end{cases}$$

The weights are as found in Examples 3.1 and 3.2.

Step 0. $\mathbf{W} = \begin{bmatrix} 2 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}$.

Step 1. For the first input pattern, do Steps 2–4.

Step 2. $\mathbf{x} = (1, 0, 0, 0)$.

Step 3.
$$y\_in_1 = x_1w_{11} + x_2w_{21} + x_3w_{31} + x_4w_{41}$$
$$= 1(2) + 0(1) + 0(0) + 0(0)$$
$$= 2;$$
$$y\_in_2 = x_1w_{12} + x_2w_{22} + x_3w_{32} + x_4w_{42}$$
$$= 1(0) + 0(0) + 0(1) + 0(2)$$
$$= 0.$$

Step 4.
$$y_1 = f(y\_in_1) = f(2) = 1;$$
$$y_2 = f(y\_in_2) = f(0) = 0.$$

(This is the correct response for the first training pattern.)

Step 1. For the second input pattern, do Steps 2–4.

Step 2. $\mathbf{x} = (1, 1, 0, 0)$.

Step 3.
$$y\_in_1 = x_1w_{11} + x_2w_{21} + x_3w_{31} + x_4w_{41}$$
$$= 1(2) + 1(1) + 0(0) + 0(0)$$
$$= 3;$$
$$y\_in_2 = x_1w_{12} + x_2w_{22} + x_3w_{32} + x_4w_{42}$$
$$= 1(0) + 1(0) + 0(1) + 0(2)$$
$$= 0.$$

Step 4.
$$y_1 = f(y\_in_1) = f(3) = 1;$$
$$y_2 = f(y\_in_2) = f(0) = 0.$$

(This is the correct response for the second training pattern.)

Step 1. For the third input pattern, do Steps 2–4.

Step 2. $\mathbf{x} = (0, 0, 0, 1)$.

Step 3. $y\_in_1 = x_1w_{11} + x_2w_{21} + x_3w_{31} + x_4w_{41}$

$= 0(2) + 0(1) + 0(0) + 1(0)$

$= 0;$

$y\_in_2 = x_1w_{12} + x_2w_{22} + x_3w_{32} + x_4w_{42}$

$= 0(0) + 0(0) + 0(1) + 1(2)$

$= 2.$

Step 4. $y_1 = f(y\_in_1) = f(0) = 0;$

$y_2 = f(y\_in_2) = f(2) = 1.$

(This is the correct response for the third training pattern.)

Step 1. For the fourth input pattern, do Steps 2–4.

Step 2. $\mathbf{x} = (0, 0, 1, 1)$.

Step 3. $y\_in_1 = x_1w_{11} + x_2w_{21} + x_3w_{31} + x_4w_{41}$

$= 0(2) + 0(1) + 1(0) + 1(0)$

$= 0;$

$y\_in_2 = x_1w_{12} + x_2w_{22} + x_3w_{32} + x_4w_{42}$

$= 0(0) + 0(0) + 1(1) + 1(2)$

$= 3.$

Step 4. $y_1 = f(y\_in_1) = f(0) = 0;$

$y_2 = f(y\_in_2) = f(2) = 1.$

(This is the correct response for the fourth training pattern.)

The process we have just illustrated can be represented much more succinctly using vector-matrix notation. Note first, that the net input to any particular output unit is the (dot) product of the input (row) vector with the column of the weight matrix that has the weights for the output unit in question. The (row) vector with all of the net inputs is simply the product of the input vector and the weight matrix.

We repeat the steps of the application procedure for the input vector **x**, which is the first of the training input vectors **s**.

Step 0. $\mathbf{W} = \begin{bmatrix} 2 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}.$

Step 1. For the input vector:

Step 2. $\mathbf{x} = (1, 0, 0, 0)$.

Step 3. $\mathbf{x}\,\mathbf{W} = (y\_in_1, y\_in_2)$

$$(1, 0, 0, 0) \begin{bmatrix} 2 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix} = (2, 0).$$

Step 4. $f(2) = 1; \quad f(0) = 0;$
$\mathbf{y} = (1, 0).$

The entire process (Steps 2–4) can be represented by

$$\mathbf{x}\mathbf{W} = (y\_in_1, y\_in_2) \rightarrow \mathbf{y}$$

$$(1, 0, 0, 0)\begin{bmatrix} 2 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix} = (2, 0) \qquad \rightarrow (1, 0),$$

or, in slightly more compact notation,

$$(1, 0, 0, 0)\cdot\mathbf{W} = (2, 0) \rightarrow (1, 0).$$

Note that the output activation vector is the same as the training output vector that was stored in the weight matrix for this input vector.

Similarly, applying the same algorithm, with $\mathbf{x}$ equal to each of the other three training input vectors, yields

$$(1, 1, 0, 0)\cdot\mathbf{W} = (3, 0) \rightarrow (1, 0),$$

$$(0, 0, 0, 1)\cdot\mathbf{W} = (0, 2) \rightarrow (0, 1),$$

$$(0, 0, 1, 1)\cdot\mathbf{W} = (0, 3) \rightarrow (0, 1).$$

Note that the net has responded correctly to (has produced the desired vector of output activations for) each of the training patterns.

Example 3.4 Testing a heteroassociative net with input similar to the training input

The test vector $\mathbf{x} = (0, 1, 0, 0)$ differs from the training vector $\mathbf{s} = (1, 1, 0, 0)$ only in the first component. We have

$$(0, 1, 0, 0)\cdot\mathbf{W} = (1, 0) \rightarrow (1, 0).$$

Thus, the net also associates a known output pattern with this input.

Example 3.5 Testing a heteroassociative net with input that is not similar to the training input

The test pattern (0 1, 1, 0) differs from each of the training input patterns in at least two components. We have

$$(0\ 1, 1, 0)\cdot\mathbf{W} = (1, 1) \rightarrow (1, 1).$$

The output is not one of the outputs with which the net was trained; in other words, the net does not recognize the pattern. In this case, we can view $\mathbf{x} = (0, 1, 1, 0)$ as differing from the training vector $\mathbf{s} = (1, 1, 0, 0)$ in the first and third components, so that the two "mistakes" in the input pattern make it impossible for the net to recognize it. This is not surprising, since the vector could equally well be viewed as formed from $\mathbf{s} = (0, 0, 1, 1)$, with "mistakes" in the second and fourth components.

In general, a bipolar representation of our patterns is computationally preferable to a binary representation. Examples 3.6 and 3.7 illustrate modifications

of the previous examples to make use of the improved characteristics of bipolar vectors. In the first modification (Example 3.6), binary input and target vectors are converted to bipolar representations for the formation of the weight matrix. However, the input vectors used during testing and the response of the net are still represented in binary form. In the second modification (Example 3.7), all vectors (training input, target output, testing input, and the response of the net) are expressed in bipolar form.

Example 3.6 A heteroassociative net using hybrid (binary/bipolar) data representation

Even if one wishes to use binary input vectors, it may be advantageous to form the weight matrix from the bipolar form of training vector pairs. Specifically, to store a set of binary vector pairs $\mathbf{s}(p):\mathbf{t}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$

and

$$\mathbf{t}(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p)),$$

using a weight matrix formed from the corresponding bipolar vectors, the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p (2s_i(p) - 1)(2t_j(p) - 1).$$

Using the data from Example 3.1, we have

$$\mathbf{s}(1) = (1, 0, 0, 0), \quad \mathbf{t}(1) = (1, 0);$$
$$\mathbf{s}(2) = (1, 1, 0, 0), \quad \mathbf{t}(2) = (1, 0);$$
$$\mathbf{s}(3) = (0, 0, 0, 1), \quad \mathbf{t}(3) = (0, 1);$$
$$\mathbf{s}(4) = (0, 0, 1, 1), \quad \mathbf{t}(4) = (0, 1).$$

The weight matrix that is obtained

$$\mathbf{W} = \begin{bmatrix} 4 & -4 \\ 2 & -2 \\ -2 & 2 \\ -4 & 4 \end{bmatrix}.$$

Example 3.7 A heteroassociative net using bipolar vectors

To store a set of bipolar vector pairs $\mathbf{s}(p):\mathbf{t}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$

and

$$\mathbf{t}(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p)),$$

the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p s_i(p)t_j(p).$$

Using the data from Examples 3.1 through 3.6, we have

$$\mathbf{s}(1) = (\ 1, -1, -1, -1), \qquad \mathbf{t}(1) = (\ 1, -1);$$
$$\mathbf{s}(2) = (\ 1,\ \ 1, -1, -1), \qquad \mathbf{t}(2) = (\ 1, -1);$$
$$\mathbf{s}(3) = (-1, -1, -1,\ \ 1), \qquad \mathbf{t}(3) = (-1,\ \ 1);$$
$$\mathbf{s}(4) = (-1, -1,\ \ 1,\ \ 1), \qquad \mathbf{t}(4) = (-1,\ \ 1).$$

The same weight matrix is obtained as in Example 3.6, namely,

$$\mathbf{W} = \begin{bmatrix} 4 & -4 \\ 2 & -2 \\ -2 & 2 \\ -4 & 4 \end{bmatrix}.$$

We illustrate the process of finding the weights using outer products for this example.

The weight matrix to store the first pattern pair is given by the outer product of the vectors

$$\mathbf{s} = (1, -1, -1, -1)$$

and

$$\mathbf{t} = (1, -1).$$

The weight matrix is

$$\begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} [1 \quad -1] = \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix}.$$

Similarly, to store the second pair,

$$\mathbf{s} = (1, 1, -1, -1)$$

and

$$\mathbf{t} = (1, -1),$$

the weight matrix is

$$\begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} [1 \ -1] = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix}.$$

To store the third pair,

$$\mathbf{s} = (-1, -1, -1, 1)$$

and

$$\mathbf{t} = (-1,\ \ 1),$$

the weight matrix is

$$\begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} [-1 \quad 1] = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

And to store the fourth pair,

$$\mathbf{s} = (-1, -1, 1, 1)$$

and

$$\mathbf{t} = (-1, 1),$$

the weight matrix is

$$\begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} [-1 \quad 1] = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix}.$$

The weight matrix to store all four pattern pairs is the sum of the weight matrices to store each pattern pair separately, namely,

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & -4 \\ 2 & -2 \\ -2 & 2 \\ -4 & 4 \end{bmatrix}.$$

One of the computational advantages of bipolar representation of our patterns is that it gives us a very simple way of expressing two different levels of noise that may be applied to our training inputs to produce testing inputs for our net. For convenience, we shall refer informally to these levels as "missing data" and "mistakes." For instance, if each of our original patterns is a sequence of *yes* or *no* responses, "missing data" would correspond to a response of *unsure*, whereas a "mistake" would be a response of *yes* when the correct response was *no* and vice versa. With bipolar representations, *yes* would be represented by $+1$, *no* by -1, and *unsure* by 0.

Example 3.8 The effect of data representation: bipolar is better than binary

Example 3.5 illustrated the difficulties that a simple net (with binary input) experiences when given an input vector with "mistakes" in two components. The weight matrix formed from the bipolar representation of training patterns still cannot produce the proper response for an input vector formed from a stored vector with two "mistakes," e.g.,

$$(-1, 1, 1, -1)\cdot\mathbf{W} = (0, 0) \rightarrow (0, 0).$$

However, the net can respond correctly when given an input vector formed from a stored vector with two components "missing." For example, consider the vector

$\mathbf{x} = (0, 1, 0, -1)$, which is formed from the training vector $\mathbf{s} = (1, 1, -1, -1)$, with the first and third components "missing" rather than "wrong." We have

$$(0, 1, 0, -1)\cdot\mathbf{W} = (6, -6) \rightarrow (1, -1),$$

the correct response for the stored vector $\mathbf{s} = (1, 1, -1, -1)$. These "missing" components are really just a particular form of noise that produces an input vector which is not as dissimilar to a training vector as is the input vector produced with the more extreme "noise" denoted by the term "mistake."

Character recognition

Example 3.9 A heteroassociative net for associating letters from different fonts

A heteroassociative neural net was trained using the Hebb rule (outer products) to associate three vector pairs. The **x** vectors have 63 components, the **y** vectors 15. The vectors represent two-dimensional patterns. The pattern

is converted to a vector representation that is suitable for processing as follows: The #s are replaced by 1's and the dots by -1's, reading across each row (starting with the top row). The pattern shown becomes the vector

$$(-1, 1, -1, \quad 1, -1, 1, \quad 1, 1, 1, \quad 1, -1, 1 \quad 1, -1, 1).$$

The extra spaces between the vector components, which separate the different rows of the original pattern for ease of reading, are not necessary for the network.

Figure 3.3 shows the vector pairs in their original two-dimensional form.

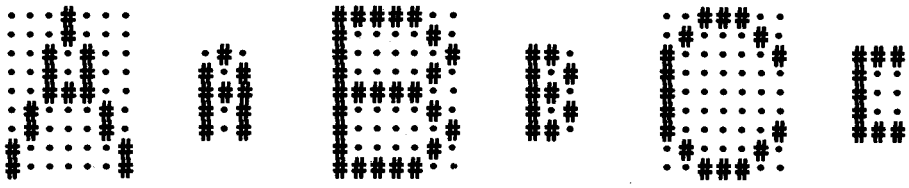

Figure 3.3 Training patterns for character recognition using heteroassociative net.

After training, the net was used with input patterns that were noisy versions of the training input patterns. The results are shown in Figures 3.4 and 3.5. The noise took the form of turning pixels "on" that should have been "off" and vice versa. These are denoted as follows:

@ Pixel is now "on," but this is a mistake (noise).

O Pixel is now "off," but this is a mistake (noise).

Figure 3.5 shows that the neural net can recognize the small letters that are

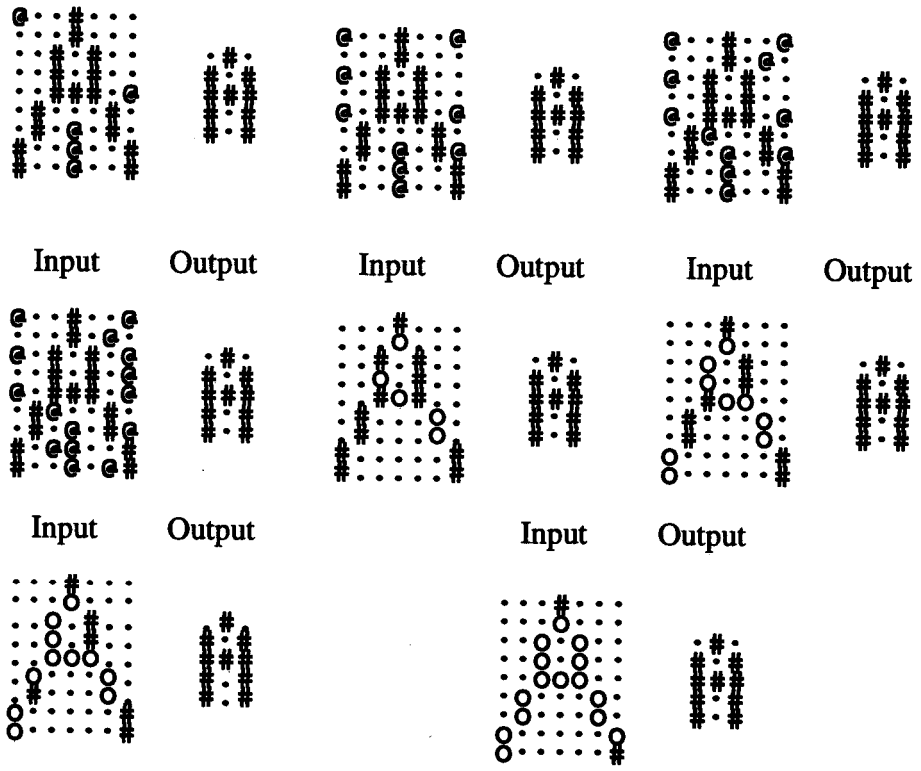

Figure 3.4 Response of heteroassociative net to several noisy versions of pattern A.

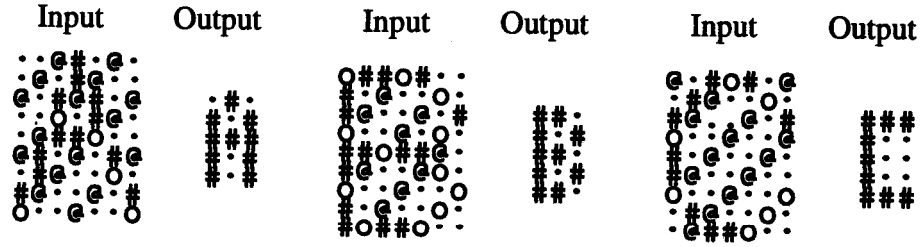

Figure 3.5 Response of heteroassociative net to patterns A, B, and C with mistakes in 1/3 of the components.

stored in it, even when given input patterns representing the large training patterns with 30% noise.

3.3 AUTOASSOCIATIVE NET

The feedforward autoassociative net considered in this section is a special case of the heteroassociative net described in Section 3.2. For an autoassociative net, the training input and target output vectors are identical. The process of training is often called *storing* the vectors, which may be binary or bipolar. A stored vector can be retrieved from distorted or partial (noisy) input if the input is sufficiently similar to it. The performance of the net is judged by its ability to reproduce a stored pattern from noisy input; performance is, in general, better for bipolar vectors than for binary vectors. In Section 3.4, several different versions of iterative autoassociative nets are discussed.

It is often the case that, for autoassociative nets, the weights on the diagonal (those which would connect an input pattern component to the corresponding component in the output pattern) are set to zero. This will be illustrated in Example 3.14. Setting these weights to zero may improve the net's ability to generalize (especially when more than one vector is stored in it) [Szu, 1989] or may increase the biological plausibility of the net [Anderson, 1972]. Setting them to zero is necessary for extension to the iterative case [Hopfield, 1982] or if the delta rule is used (to prevent the training from producing the identity matrix for the weights) [McClelland & Rumelhart, 1988].

3.3.1 Architecture

Figure 3.6 shows the architecture of an autoassociative neural net.

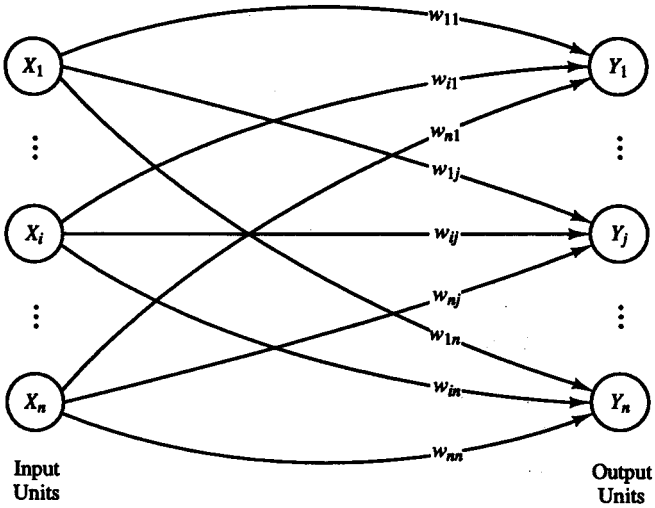

Figure 3.6 Autoassociative neural net.

3.3.2 Algorithm

For mutually orthogonal vectors, the Hebb rule can be used for setting the weights in an autoassociative net because the input and output vectors are perfectly correlated, component by component (i.e., they are the same). The algorithm is as given in Section 3.1.1; note that there are the same number of output units as input units.

Step 0. Initialize all weights, $i = 1, \ldots, n; j = 1, \ldots, n$:

$$w_{ij} = 0;$$

Step 1. For each vector to be stored, do Steps 2–4:

Step 2. Set activation for each input unit, $i = 1, \ldots, n$:

$$x_i = s_i.$$

Step 3. Set activation for each output unit, $j = 1, \ldots, n$:

$$y_j = s_j;$$

Step 4. Adjust the weights, $i = 1, \ldots, n; j = 1, \ldots, n$:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j.$$

As discussed earlier, in practice the weights are usually set from the formula

$$\mathbf{W} = \sum_{p=1}^{P} \mathbf{s}^{\mathrm{T}}(p)\mathbf{s}(p),$$

rather than from the algorithmic form of Hebb learning.

3.3.3 Application

An autoassociative neural net can be used to determine whether an input vector is "known" (i.e., stored in the net) or "unknown." The net recognizes a "known" vector by producing a pattern of activation on the output units of the net that is the same as one of the vectors stored in it. The application procedure (with bipolar inputs and activations) is as follows:

Step 0. Set the weights (using Hebb rule, outer product).

Step 1. For each testing input vector, do Steps 2–4.

Step 2. Set activations of the input units equal to the input vector.

Step 3. Compute net input to each output unit, $j = 1, \ldots, n$:

$$y\_in_j = \sum_i x_i w_{ij}.$$

Step 4. Apply activation function ($j = 1, \ldots, n$):

$$y_j = f(y\_in_j) = \begin{cases} 1 & \text{if } y\_in_j > 0; \\ -1 & \text{if } y\_in_j \leq 0. \end{cases}$$

(or use f from p. 109. Step 4)

Simple examples

Example 3.10 An autoassociative net to store one vector: recognizing the stored vector

We illustrate the process of storing one pattern in an autoassociative net and then recalling, or recognizing, that stored pattern.

Step 0. The vector $\mathbf{s} = (1, 1, 1, -1)$ is stored with the weight matrix:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}.$$

Step 1. For the testing input vector:
Step 2. $\mathbf{x} = (1, 1, 1, -1)$.
Step 3. $\mathbf{y\_in} = (4, 4, 4, -4)$.
Step 4. $\mathbf{y} = \mathrm{f}(4, 4, 4, -4) = (1, 1, 1, -1)$.

Since the response vector y is the same as the stored vector, we can say the input vector is recognized as a "known" vector.

The preceding process of using the net can be written more succinctly as

$$(1, 1, 1, -1)\cdot\mathbf{W} = (4, 4, 4, -4) \rightarrow (1, 1, 1, -1).$$

Now, if recognizing the vector that was stored were all that this weight matrix enabled the net to do, it would be no better than using the identity matrix for the weights. However, an autoassociative neural net can recognize as "known" vectors that are similar to the stored vector, but that differ slightly from it. As before, the differences take one of two forms: "mistakes" in the data or "missing" data. The only "mistakes" we consider are changes from $+1$ to -1 or vice versa. We use the term "missing" data to refer to a component that has the value 0, rather than either $+1$ or -1.

Example 3.11 Testing an autoassociative net: one mistake in the input vector

Using the succinct notation just introduced, consider the performance of the net for each of the input vectors x that follow. Each vector x is formed from the original stored vector s with a mistake in one component.

$$(-1, \ 1, \ 1, -1)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(\ 1, -1, \ 1, -1)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(\ 1, \ 1, -1, -1)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(\ 1, \ 1, \ 1, \ 1)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1).$$

Note that in each case the input vector is recognized as "known" after a single update of the activation vector in Step 4 of the algorithm. The reader can verify that the net also recognizes the vectors formed when one component is "missing." Those vectors are (0, 1, 1, −1), (1, 0, 1, −1), (1, 1, 0, −1), and (1, 1, 1, 0).

In general, a net is more tolerant of "missing" data than it is of "mistakes" in the data, as the examples that follow demonstrate. This is not surprising, since the vectors with "missing" data are closer (both intuitively and in a mathematical sense) to the training patterns than are the vectors with "mistakes."

Example 3.12 Testing an autoassociative net: two "missing" entries in the input vector

The vectors formed from (1, 1, 1, −1) with two "missing" data are (0, 0, 1, −1), (0, 1, 0, −1), (0, 1, 1, 0), (1, 0, 0, −1), (1, 0, 1, 0), and (1, 1, 0, 0). As before, consider the performance of the net for each of these input vectors:

$$(0, 0, 1, -1)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(0, 1, 0, -1)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(0, 1, 1, \ 0)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(1, 0, 0, -1)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(1, 0, 1, \ 0)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1)$$

$$(1, 1, 0, \ 0)\cdot\mathbf{W} = (2, 2, 2, -2) \rightarrow (1, 1, 1, -1).$$

The response of the net indicates that it recognizes each of these input vectors as the training vector (1, 1, 1, −1), which is what one would expect, or at least hope for.

Example 3.13 Testing an autoassociative net: two mistakes in the input vector

The vector (−1, −1, 1, −1) can be viewed as being formed from the stored vector (1, 1, 1, −1) with two mistakes (in the first and second components). We have:

$$(-1, -1, 1, -1)\cdot\mathbf{W} = (0, 0, 0, 0).$$

The net does not recognize this input vector.

Example 3.14 An autoassociative net with no self-connections: zeroing-out the diagonal

It is fairly common for an autoassociative network to have its diagonal terms set to zero, e.g.,

$$\mathbf{W}_0 = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}.$$

Consider again the input vector (−1, −1, 1, −1) formed from the stored vector (1, 1, 1, −1) with two mistakes (in the first and second components). We have:

$$(-1, -1, 1, -1)\cdot\mathbf{W}_0 = (-1, 1, -1, 1).$$

The net still does not recognize this input vector.

It is interesting to note that if the weight matrix $\mathbf{W}_0$ (with 0's on the diagonal) is used in the case of "missing" components in the input data (see Example 3.12), the output unit or units with the net input of largest magnitude coincide with the input unit or units whose input component or components were zero. We have:

$$(0, 0, 1, -1)\cdot\mathbf{W}_0 = (2, 2, 1, -1) \rightarrow (1, 1, 1, -1)$$

$$(0, 1, 0, -1)\cdot\mathbf{W}_0 = (2, 1, 2, -1) \rightarrow (1, 1, 1, -1)$$

$$(0, 1, 1, \ 0)\cdot\mathbf{W}_0 = (2, 1, 1, -2) \rightarrow (1, 1, 1, -1)$$

$$(1, 0, 0, -1)\cdot\mathbf{W}_0 = (1, 2, 2, -1) \rightarrow (1, 1, 1, -1)$$

$$(1, 0, 1, \ 0)\cdot\mathbf{W}_0 = (1, 2, 1, -2) \rightarrow (1, 1, 1, -1)$$

$$(1, 1, 0, \ 0)\cdot\mathbf{W}_0 = (1, 1, 2, -2) \rightarrow (1, 1, 1, -1).$$

The net recognizes each of these input vectors.

3.3.4 Storage Capacity

An important consideration for associative memory neural networks is the number of patterns or pattern pairs that can be stored before the net begins to forget. In this section we consider some simple examples and theorems for noniterative autoassociative nets.

Examples

Example 3.15 Storing two vectors in an autoassociative net

More than one vector can be stored in an autoassociative net by adding the weight matrices for each vector together. For example, if $\mathbf{W}_1$ is the weight matrix used to store the vector $(1, 1, -1, -1)$ and $\mathbf{W}_2$ is the weight matrix used to store the vector $(-1, 1, 1, -1)$, then the weight matrix used to store both $(1, 1, -1, -1)$ and $(-1, 1, 1, -1)$ is the sum of $\mathbf{W}_1$ and $\mathbf{W}_2$. Because it is desired that the net respond with one of the stored vectors when it is presented with an input vector that is similar (but not identical) to a stored vector, it is customary to set the diagonal terms in the weight matrices to zero. If this is not done, the diagonal terms (which would each be equal to the number of vectors stored in the net) would dominate, and the net would tend to reproduce the input vector rather than a stored vector. The addition of $\mathbf{W}_1$ and $\mathbf{W}_2$ proceeds as follows:

$$\underset{\mathbf{W}_1}{\begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{bmatrix}} + \underset{\mathbf{W}_2}{\begin{bmatrix} 0 & -1 & -1 & 1 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & -1 \\ 1 & -1 & -1 & 0 \end{bmatrix}} = \underset{\mathbf{W}_1 + \mathbf{W}_2}{\begin{bmatrix} 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & -2 \\ -2 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \end{bmatrix}}.$$

The reader should verify that the net with weight matrix $\mathbf{W}_1 + \mathbf{W}_2$ can recognize both of the vectors $(1, 1, -1, -1)$ and $(-1, 1, 1, -1)$. The number of vectors that can be stored in a net is called the *capacity* of the net.

Example 3.16 Attempting to store two nonorthogonal vectors in an autoassociative net

Not every pair of bipolar vectors can be stored in an autoassociative net with four nodes; attempting to store the vectors (1, −1, −1, 1) and (1, 1, −1, 1) by adding their weight matrices gives a net that cannot distinguish between the two vectors it was trained to recognize:

$$\begin{bmatrix} 0 & -1 & -1 & 1 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & -1 \\ 1 & -1 & -1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & -1 & 1 \\ 1 & 0 & -1 & 1 \\ -1 & -1 & 0 & -1 \\ 1 & 1 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -2 & 2 \\ 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & -2 \\ 2 & 0 & -2 & 0 \end{bmatrix}.$$

The difference between Example 3.15 and this example is that there the vectors are orthogonal, while here they are not. Recall that two vectors **x** and **y** are orthogonal if

$$\mathbf{x}\,\mathbf{y}^T = \sum_i x_i y_i = 0.$$

Informally, this example illustrates the difficulty that results from trying to store vectors that are too similar.

An autoassociative net with four nodes can store three orthogonal vectors (i.e., each vector is orthogonal to each of the other two vectors). However, the weight matrix for four mutually orthogonal vectors is always singular (so four vectors cannot be stored in an autoassociative net with four nodes, even if the vectors are orthogonal). These properties are illustrated in Examples 3.17 and 3.18.

Example 3.17 Storing three mutually orthogonal vectors in an autoassociative net

Let $\mathbf{W}_1 + \mathbf{W}_2$ be the weight matrix to store the orthogonal vectors (1, 1, −1, −1) and (−1, 1, 1, −1) and $\mathbf{W}_3$ be the weight matrix that stores (−1, 1, −1, 1). Then the weight matrix to store all three orthogonal vectors is $\mathbf{W}_1 + \mathbf{W}_2 + \mathbf{W}_3$. We have

$$\underset{\mathbf{W}_1 + \mathbf{W}_2}{\begin{bmatrix} 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & -2 \\ -2 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \end{bmatrix}} + \underset{\mathbf{W}_3}{\begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}} = \underset{\mathbf{W}_1 + \mathbf{W}_2 + \mathbf{W}_3}{\begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}},$$

which correctly classifies each of the three vectors on which it was trained.

Example 3.18 Attempting to store four vectors in an autoassociative net

Attempting to store a fourth vector, (1, 1, 1, 1), with weight matrix $\mathbf{W}_4$, orthogonal to each of the foregoing three, demonstrates the difficulties encountered in over training a net, namely, previous learning is erased. Adding the weight matrix for the new vector to the matrix for the first three vectors gives

$$\underbrace{\begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}}_{\mathbf{W}_1 + \mathbf{W}_2 + \mathbf{W}_3} + \underbrace{\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}}_{\mathbf{W}_4} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{W}^*},$$

which cannot recognize any vector.

Theorems

The capacity of an autoassociative net depends on the number of components the stored vectors have and the relationships among the stored vectors; more vectors can be stored if they are mutually orthogonal.

Expanding on ideas suggested by Szu (1989), we prove that $n - 1$ mutually orthogonal bipolar vectors, each with n components, can always be stored using the sum of the outer product weight matrices (with diagonal terms set to zero), but that attempting to store n mutually orthogonal vectors will result in a weight matrix that cannot reproduce any of the sotred vectors. Recall again that two vectors $\mathbf{x}$ and $\mathbf{y}$ are orthogonal if $\sum_i x_i y_i = 0$.

Notation. The kth vector to be stored is denoted by the row vector

$$\mathbf{a}(k) = (a_1(k), a_2(k), \ldots, a_n(k)).$$

The weight matrix to store $\mathbf{a}(k)$ is given by

$$\mathbf{W}(k) = \begin{bmatrix} 0 & a_1(k)a_2(k) & \cdots & a_1(k)a_n(k) \\ a_2(k)a_1(k) & 0 & \cdots & a_2(k)a_n(k) \\ \vdots & \vdots & \cdots & \vdots \\ a_n(k)a_1(k) & a_n(k)a_2(k) & \cdots & 0 \end{bmatrix}.$$

The weight matrix to store $\mathbf{a}(1), \mathbf{a}(2), \ldots, \mathbf{a}(m)$ has the general element

$$w_{ij} = \begin{cases} 0 & \text{if } i = j; \\ \sum_{p=1}^{m} a_i(p)a_j(p) & \text{otherwise.} \end{cases}$$

The vector $\mathbf{a}(k)$ can be recalled when it is input to a net with weight matrix $\mathbf{W}$ if $\mathbf{a}(k)$ is an eigenvector of matrix $\mathbf{W}$. To test whether $\mathbf{a}(k)$ is an eigenvector, and to determine the corresponding eigenvalue, consider the formula

$$(a_1(k), a_2(k), \ldots, a_n(k))\,\mathbf{W} = \left(\sum_{i=1}^{n} a_i(k)w_{i1}, \sum_{i=1}^{n} a_i(k)w_{i2}, \ldots, \sum_{i=1}^{n} a_i(k)w_{in}\right).$$

The jth component of $\mathbf{a}(k)\,\mathbf{W}$ is

$$\sum_{i=1}^{n} a_i(k)w_{ij} = \sum_{i\neq j} a_i(k) \sum_{p=1}^{m} a_i(p)a_j(p) = \sum_{p=1}^{m} a_j(p) \sum_{i\neq j} a_i(k)a_i(p).$$

Because the stored vectors are orthogonal,

$$\sum_{i=1}^{n} a_i(k)a_i(p) = 0 \qquad \text{for } k \neq p$$

and

$$\sum_{i\neq j} a_i(k)a_i(p) = \begin{cases} -a_j(k)a_j(p) & \text{for } k \neq p; \\ n-1 & \text{for } k = p. \end{cases}$$

Since the vectors are bipolar, it is always the case that $[a_i(k)]^2 = 1$.

Combining these results, we get, for the jth component of $\mathbf{a}(k)\,\mathbf{W}$,

$$\begin{aligned} \sum_{p=1}^{m} a_j(p) \sum_{i\neq j} a_i(k)a_i(p) &= \sum_{p\neq k} a_j(p) \sum_{i\neq j} a_i(k)a_i(p) + a_j(k) \sum_{i\neq j} a_i(k)a_i(p) \\ &= \sum_{p\neq k} a_j(p)\,[-a_j(k)a_j(p)] + a_j(k)(n-1) \\ &= \sum_{p\neq k} -a_j(k) + a_j(k)(n-1) \\ &= -(m-1)a_j(k) + a_j(k)(n-1) \\ &= (n-m)a_j(k). \end{aligned}$$

Thus, $\mathbf{a}(k)\mathbf{W} = (n-m)\mathbf{a}(k)$, which shows that $\mathbf{a}(k)$ is an eigenvector of the weight matrix $\mathbf{W}$, with eigenvalue $(n-m)$, where n is the number of components of the vectors being stored and m is the number of stored (orthogonal) vectors. This establishes the following result.

Theorem 3.1. For $m < n$, the weight matrix is nonsingular. The eigenvalue $(n-m)$ has geometric multiplicity m, with eigenvectors $\mathbf{a}(1)$, $\mathbf{a}(2)$, . . . , $\mathbf{a}(m)$. For $m = n$, zero is an eigenvalue of multiplicity n, and there are no nontrivial eigenvectors.

The following result can also be shown.

Theorem 3.2. A set of 2^k mutually orthogonal bipolar vectors can be constructed for $n = 2^k\,m$ (for m odd), and no larger set can be formed.

The proof is based on the following observations:

1. Let [v, v] denote the concatenation of the vector v with itself (producing a vector with $2n$ components if v is an n-tuple).

2. If **a** and **b** are any two mutually orthogonal bipolar vectors (n-tuples), then [**a**, **a**], [**a**, $-$**a**], [**b**, **b**], and [**b**, $-$**b**] are mutually orthogonal $2n$-tuples.
3. Any number n can be expressed as 2^k m, where m is odd and $k \geq 0$.
4. It is clear that it is not possible to construct a pair of orthogonal bipolar n-tuples for n odd ($k = 0$), since the dot product of two bipolar n-tuples has the same parity as n.

The construction of the desired set of mutually orthogonal vectors proceeds as follows:

1. Form vector $\mathbf{v}_m(1) = (1, 1, 1, \ldots, 1)$, an m-tuple.
2. Form

$$\mathbf{v}_{2m}(1) = [\mathbf{v}_m(1), \quad \mathbf{v}_m(1)]$$

and

$$\mathbf{v}_{2m}(2) = [\mathbf{v}_m(1), \; -\mathbf{v}_m(1)];$$

$\mathbf{v}_{2m}(1)$ and $\mathbf{v}_{2m}(2)$ are othogonal $2m$-tuples.

3. Form the four orthogonal $4m$-tuples

$$\mathbf{v}_{4m}(1) = [\mathbf{v}_{2m}(1), \quad \mathbf{v}_{2m}(1)],$$

$$\mathbf{v}_{4m}(2) = [\mathbf{v}_{2m}(1), \; -\mathbf{v}_{2m}(1)],$$

$$\mathbf{v}_{4m}(3) = [\mathbf{v}_{2m}(2), \quad \mathbf{v}_{2m}(2)],$$

and

$$\mathbf{v}_{4m}(4) = [\mathbf{v}_{2m}(2), \; -\mathbf{v}_{2m}(2)].$$

4. Continue until $\mathbf{v}_n(1), \ldots, \mathbf{v}_n(2^k)$ have been formed; this is the required set of 2^k orthogonal vectors with $n = 2^k\, m$ components.

The method of proving that the set of orthogonal vectors constructed by the preceding procedure is maximal is illustrated here for $n = 6$. Consider the orthogonal vectors $\mathbf{v}(1) = (1, 1, 1, 1, 1, 1)$ and $\mathbf{v}(2) = (1, 1, 1, -1, -1, -1)$ constructed with the technique. Assume that there is a third vector, (a, b, c, d, e, f), which is orthogonal to both $\mathbf{v}(1)$ and $\mathbf{v}(2)$. This requires that $a + b + c + d + e + f = 0$ and $a + b + c - d - e - f = 0$. Combining these equations gives $a + b + c = 0$, which is impossible for $a, b, c \in \{1, -1\}$.

3.4 ITERATIVE AUTOASSOCIATIVE NET

We see from the next example that in some cases the net does not respond immediately to an input signal with a stored target pattern, but the response may be enough like a stored pattern (at least in the sense of having more nodes com-

mitted to values of +1 or −1 and fewer nodes with the "unsure" response of 0) to suggest using this first response as input to the net again.

Example 3.19 Testing a recurrent autoassociative net: stored vector with second, third and fourth components set to zero

The weight matrix to store the vector (1, 1, 1, −1) is

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}.$$

The vector (1, 0, 0, 0) is an example of a vector formed from the stored vector with three "missing" components (three zero entries). The performance of the net for this vector is given next.

Input vector (1, 0, 0, 0):

$$(1, 0, 0, 0)\cdot\mathbf{W} = (0, 1, 1, -1) \rightarrow \text{iterate}$$

$$(0, 1, 1, -1)\cdot\mathbf{W} = (3, 2, 2, -2) \rightarrow (1, 1, 1, -1).$$

Thus, for the input vector (1, 0, 0, 0), the net produces the "known" vector (1, 1, 1, −1) as its response in two iterations.

We can also take this iterative feedback scheme a step further and simply let the input and output units be the same, to obtain a recurrent autoassociative neural net. In Sections 3.4.1–3.4.3, we consider three that differ primarily in their activation function. Then, in Section 3.4.4, we examine a net developed by Nobel prize–winning physicist John Hopfield (1982, 1988). Hopfield's work (and his prestige) enhanced greatly the respectability of neural nets as a field of study in the 1980s. The differences between his net and the others in this section, although fairly slight, have a significant impact on the performance of the net. For iterative nets, one key question is whether the activations will converge. The weights are fixed (by the Hebb rule for example), but the activations of the units change.

3.4.1 Recurrent Linear Autoassociator

One of the simplest iterative autoassociator neural networks is known as the *linear autoassociator* [McClelland & Rumelhart, 1988; Anderson et al., 1977]. This net has n neurons, each connected to all of the other neurons. The weight matrix is symmetric, with the connection strength w_{ij} proportional to the sum over all training patterns of the product of the activations of the two units x_i and x_j. In other words, the weights can be found by the Hebb rule. McClelland and Rumelhart do not restrict the weight matrix to have zeros on the diagonal. Anderson et al. show that setting the diagonal elements in the weight matrix to zero, which they believe represents a biologically more plausible model, does not change the performance of the net significantly.

The performance of the net can be analyzed [Anderson et al., 1977] using ideas from linear algebra. An $n \times n$ nonsingular symmetric matrix (such as the weight matrix) has n mutually orthogonal eigenvectors. A recurrent linear autoassociator neural net is trained using a set of K orthogonal unit vectors $\mathbf{f}_1, \ldots, \mathbf{f}_K$, where the number of times each vector is presented, say, $\beta_1, \ldots, \beta_K$, is not necessarily the same. A formula for the components of the weight matrix could be derived as a simple generalization of the formula given before for the Hebb rule, allowing for the fact that some of the stored vectors were repeated. It is easy to see that each of these stored vectors is an eigenvector of the weight matrix. Furthermore, the number of times the vector was presented is the corresponding eigenvalue.

The response of the net, when presented with input (row) vector $\mathbf{x}$, is $\mathbf{xW}$, where $\mathbf{W}$ is the weight matrix. We know from linear algebra that the largest value of $\|\mathbf{xW}\|$ occurs when $\mathbf{x}$ is the eigenvector corresponding to the largest eigenvalue, the next largest value of $\|\mathbf{xW}\|$ occurs when $\mathbf{x}$ is the eigenvector associated with the next largest eigenvalue, etc. The recurrent linear autoassociator is intended to produce as its response (after perhaps several iterations) the stored vector (eigenvector) to which the input vector is most similar.

Any input pattern can be expressed as a linear combination of eigenvectors. The response of the net when an input vector is presented can be expressed as the corresponding linear combination of the eigenvalues (the net's response to the eigenvectors). The eigenvector to which the input vector is most similar is the eigenvector with the largest component in this linear expansion. As the net is allowed to iterate, contributions to the response of the net from eigenvectors with large eigenvalues (and with large coefficients in the input vector's eigenvector expansion) will grow relative to contributions from other eigenvectors with smaller eigenvalues (or smaller coefficients).

However, even though the net will increase its response corresponding to components of the input pattern on which it was trained most extensively (i.e., the eigenvectors associated with the largest eigenvalues), the overall response of the system may grow without bound. This difficulty leads to the modification of the next section.

3.4.2 Brain-State-in-a-Box Net

The response of the linear associator (Section 3.4.1) can be prevented from growing without bound by modifying the activation function (the identity function for the linear associator) to take on values within a cube (i.e., each component is restricted to be between -1 and 1) [Anderson, et al., 1977]. The units in the brain-state-in-a-box (BSB) net (as in the linear associator) update their activations simultaneously.

The architecture of the BSB net, as for all the nets in this section, consists of n units, each connected to every other unit. However, in this net there is a trained weight on the self-connection (i.e., the diagonal terms in the weight matrix

are not set to zero). There is also a self-connection with weight 1. The algorithm given here is based the original description of the process in Anderson et al. (1977); it is similar to that given in Hecht-Nielsen (1990). Others [McClelland & Rumelhart, 1988] present a version that does not include the learning phase.

Algorithm

Step 0. Initialize weights (small random values).
Initialize learning rates, α and β.

Step 1. For each training input vector, do Steps 2–6.

Step 2. Set initial activations of net equal to the external input vector **x**:

$$y_i = x_i.$$

Step 3. While activations continue to change, do Steps 4 and 5:

Step 4. Compute net inputs:

$$y\_in_i = y_i + \alpha \sum_j y_j w_{ji}.$$

(Each net input is a combination of the unit's previous activation and the weighted signal received from all units.)

Step 5. Each unit determines its activation (output signal):

$$y_i = \begin{cases} 1 & \text{if } y\_in_i > 1 \\ y\_in_i & \text{if } -1 \le y\_in_i \le 1 \\ -1 & \text{if } y\_in_i < -1. \end{cases}$$

(A stable state for the activation vector will be a vertex of the cube.)

Step 6. Update weights:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \beta y_i y_j.$$

3.4.3 Autoassociator With Threshold Function

A threshold function can also be used as the activation function for an iterative autoassociator net. The application procedure for bipolar (+1 or −1) vectors and activations with symmetric weights and no self-connections, i.e.,

$$w_{ij} = w_{ji},$$

$$w_{ii} = 0,$$

is as follows:

Step 0. Initialize weights to store patterns.
(Use Hebbian learning.)

Step 1. For each testing input vector, do Steps 2–5.

Step 2. Set activations **x**.

Step 3. While the stopping condition is false, repeat Steps 4 and 5.

Step 4. Update activations of all units
(the threshold, θ_i, is usually taken to be zero):

$$x_i = \begin{cases} 1 & \text{if } \sum_j x_j w_{ij} > \theta_i \\ x_i & \text{if } \sum_j x_j w_{ij} = \theta_i \\ -1 & \text{if } \sum_j x_j w_{ij} < \theta_i. \end{cases}$$

Step 5. Test stopping condition: the net is allowed to iterate until the correct vector **x** matches a stored vector, or **x** matches a previous vector **x**, or the maximum number of iterations allowed is reached.

The results for the input vectors described in Section 3.3 for the autoassociative net are the same if the net is allowed to iterate. Example 3.20 shows a situation in which the autoassociative net fails to recognize the input vector on the first presentation, but recognizes it when allowed to iterate.

Example 3.20 A recurrent autoassociative net recognizes *all* vectors formed from the stored vector with three "missing components"

The weight matrix to store the vector (1, 1, 1, −1) is

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}.$$

The vectors formed from the stored vector with three "missing" components (three zero entries) are (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), and (0, 0, 0, −1). The performance of the net on each of these is as follows:

First input vector, (1, 0, 0, 0)

Step 4: (1, 0, 0, 0)·**W** = (0, 1, 1, −1).
Step 5: (0, 1, 1, −1) is neither the stored vector nor an activation vector produced previously (since this is the first iteration), so we allow the activations to be updated again.
Step 4: (0, 1, 1, −1)·**W** = (3, 2, 2, −2) → (1, 1, 1, −1).
Step 5: (1, 1, 1, −1) is the stored vector, so we stop.

Thus, for the input vector (1, 0, 0, 0), the net produces the "known" vector (1, 1, 1, -1) as its response after two iterations.

Second input vector, (0, 1, 0, 0)

Step 4: $(0, 1, 0, 0)\cdot\mathbf{W} = (1, 0, 1, -1)$.
Step 5: (1, 0, 1, −1) is not the stored vector or a previous activation vector, so we iterate.
Step 4: $(1, 0, 1, -1)\cdot\mathbf{W} = (2, 3, 2, -2) \rightarrow (1, 1, 1, -1)$.
Step 5: (1, 1, 1, −1) is the stored vector, so we stop.

As with the first testing input, the net recognizes the input vector (0, 1, 0, 0) as the "known" vector (1, 1, 1, −1).

Third input vector, (0, 0, 1, 0)

Step 4: $(0, 0, 1, 0)\cdot\mathbf{W} = (1, 1, 0, -1)$.
Step 5: (1, 1, 0, −1) is neither the stored vector nor a previous activation vector, so we iterate.
Step 4: $(1, 1, 0, -1)\cdot\mathbf{W} = (2, 2, 3, -2) \rightarrow (1, 1, 1, -1)$.
Step 5: (1, 1, 1, −1) is the stored vector, so we stop.

Again, the input vector, (0, 0, 1, 0), produces the "known" vector (1, 1, 1, -1).

Fourth input vector, (0, 0, 0, −1)

Step 4: $(0, 0, 0, -1)\cdot\mathbf{W} = (1, 1, 1, 0)$
Step 5: Iterate.
Step 4: $(1, 1, 1, 0)\cdot\mathbf{W} = (2, 2, 2, -3) \rightarrow (1, 1, 1, , -1)$.
Step 5: (1, 1, 1, −1) is the stored vector, so we stop.

Example 3.21 Testing a recurrent autoassociative net: mistakes in the first and second components of the stored vector

One example of a vector that can be formed from the stored vector (1, 1, 1, −1) with mistakes in two components (the first and second) is (−1, −1, 1, −1). The performance of the net (with the weight matrix given in Example 3.20) is as follows.

For input vector (−1, −1, 1, 1).

Step 4: $(-1, -1, 1, -1)\cdot\mathbf{W} = (1, 1, -1, 1)$.
Step 5: Iterate.
Step 4: $(1, 1, -1, 1)\cdot\mathbf{W} = (-1, -1, 1, -1)$.
Step 5: Since this is the input vector repeated, stop.

(Further iterations would simply alternate the two activation vectors produced already.)

The behavior of the net in this case is called a *fixed-point cycle of length two.* It has been proved [Szu, 1989] that such a cycle occurs whenever the input vector is orthogonal to all of the stored vectors in the net (where the vectors have been stored using the sum of outer products with the diagonal terms set to zero). The vector $(-1, -1, 1, -1)$ is orthogonal to the stored vector $(1, 1, 1, -1)$. In general, for a bipolar vector with $2k$ components, mistakes in k components will produce a vector that is orthogonal to the original vector. We shall consider this example further in Section 3.4.4.

3.4.4 Discrete Hopfield Net

An iterative autoassociative net similar to the nets described in this chapter has been developed by Hopfield (1982, 1984). The net is a fully interconnected neural net, in the sense that each unit is connected to every other unit. The net has symmetric weights with no self-connections, i.e.,

$$w_{ij} = w_{ji}$$

and

$$w_{ii} = 0.$$

The two small differences between this net and the iterative autoassociative net presented in Section 3.4.3 can have a significant effect in terms of whether or not the nets converge for a particular input pattern. The differences are that in the Hopfield net presented here,

1. only one unit updates its activation at a time (based on the signal it receives from each other unit) and
2. each unit continues to receive an external signal in addition to the signal from the other units in the net.

The asynchronous updating of the units allows a function, known as an *energy or Lyapunov function,* to be found for the net. The existence of such a function enables us to prove that the net will converge to a stable set of activations, rather than oscillating, as the net in Example 3.21 did [Hopfield, 1982, 1984]. Lyapunov functions, developed by the Russian mathematician and mechanical engineer Alexander Mikhailovich Lyapunov (1857–1918), are important in the study of the stability of differential equations. See *Differential Equations with Applications and Historical Notes* [Simmons, 1972] for further discussion.

The original formulation of the discrete Hopfield net showed the usefulness of the net as content-addressable memory. Later extensions [Hopfield & Tank, 1985] for continuous-valued activations can be used either for pattern association or constrained optimization. Since there use in optimization problems illustrates

the "value added" from the additional computation required for the continuous activation function, we shall save our discussion of the continuous Hopfield net until Chapter 7, where we discuss the use of other nets for constrained optimization problems.

Architecture

An expanded form of a common representation of the Hopfield net is shown in Figure 3.7.

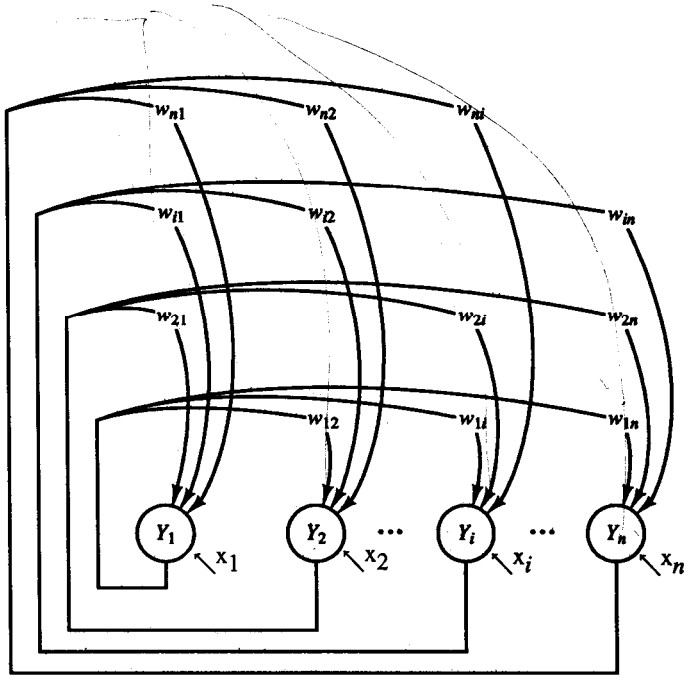

Figure 3.7 Discrete Hopfield net.

Algorithm

There are several versions of the discrete Hopfield net. Hopfield's first description [1982] used binary input vectors.

To store a set of binary patterns $\mathbf{s}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p)),$$

the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p [2s_i(p) - 1][2s_j(p) - 1] \qquad \text{for } i \neq j$$

and

$$w_{ii} = 0.$$

Other descriptions [Hopfield, 1984] allow for bipolar inputs. The weight matrix is found as follows:

To store a set of bipolar patterns $\mathbf{s}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p)),$$

the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p s_i(p)s_j(p) \qquad \text{for } i \neq j$$

and

$$w_{ii} = 0.$$

The application algorithm is stated for binary patterns; the activation function can be modified easily to accommodate bipolar patterns.

Application Algorithm for the Discrete Hopfield Net

Step 0. Initialize weights to store patterns.
(Use Hebb rule.)

While activations of the net are not converged, do Steps 1–7.

Step 1. For each input vector $\mathbf{x}$, do Steps 2–6.

Step 2. Set initial activations of net equal to the external input vector $\mathbf{x}$:

$$y_i = x_i, (i = 1, \ldots n)$$

Step 3. Do Steps 4–6 for each unit Y_i.
(Units should be updated in random order.)

Step 4. Compute net input:

$$y\_in_i = x_i + \sum_j y_j w_{ji}.$$

Step 5. Determine activation (output signal):

$$y_i = \begin{cases} 1 & \text{if } y\_in_i > \theta_i \\ y_i & \text{if } y\_in_i = \theta_i \\ 0 & \text{if } y\_in_i < \theta_i. \end{cases}$$

Step 6. Broadcast the value of y_i to all other units.
(This updates the activation vector.)

Step 7. Test for convergence.

The threshold, θ_i, is usually taken to be zero. The order of update of the units is random, but each unit must be updated at the same average rate. There are a number of variations on the discrete Hopfield net presented in this algorithm. Originally, Hopfield used binary activations, with no external input after the first time step [Hopfield, 1982]. Later, the external input was allowed to continue during processing [Hopfield, 1984]. Although typically, Hopfield used binary ac-

tivations, the model was formulated for any two distinct activation values. Descriptions by other authors use different combinations of the features of the original model; for example, Hecht-Nielsen uses bipolar activations, but no external input [Hecht-Nielsen, 1990].

The analysis of the Lyapunov function (energy function) for the Hopfield net will show that the important features of the net that guarantee convergence are the asynchronous update of the weights and the zero weights on the diagonal. It is not important whether an external signal is maintained during processing or whether the inputs and activations are binary or bipolar.

Before considering the proof that the net will converge, we consider an example of the application of the net.

Application

A binary Hopfield net can be used to determine whether an input vector is a "known" vector (i.e., one that was stored in the net) or an "unknown" vector. The net recognizes a "known" vector by producing a pattern of activation on the units of the net that is the same as the vector stored in the net. If the input vector is an "unknown" vector, the activation vectors produced as the net iterates (repeats Step 3 in the preceding algorithm) will converge to an activation vector that is not one of the stored patterns; such a pattern is called a *spurious stable state*.

Example 3.22 Testing a discrete Hopfield net: mistakes in the first and second components of the stored vector

Consider again Example 3.21, in which the vector (1, 1, 1, 0) (or its bipolar equivalent (1, 1, 1, −1)) was stored in a net. The binary input vector corresponding to the input vector used in that example (with mistakes in the first and second components) is (0, 0, 1, 0). Although the Hopfield net uses binary vectors, the weight matrix is bipolar, the same as was used in Example 3.16. The units update their activations in a random order. For this example the update order is Y_1, Y_4, Y_3, Y_2.

Step 0. Initialize weights to store patterns:

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}.$$

Step 1. The input vector is $\mathbf{x} = (0, 0, 1, 0)$. For this vector,

Step 2. $\mathbf{y} = (0, 0, 1, 0)$.

Step 3. Choose unit Y_1 to update its activation:

Step 4. $y\_in_1 = x_1 + \sum_j y_j w_{j1} = 0 + 1.$

Step 5. $y\_in_1 > 0 \rightarrow y_1 = 1.$

Step 6. $\mathbf{y} = (1, 0, 1, 0)$.

Step 3. Choose unit Y_4 to update its activation:

Step 4. $y\_in_4 = x_4 + \sum_j y_j w_{j4} = 0 + (-2).$

Step 5. $y\_in_4 < 0 \rightarrow y_4 = 0$.
Step 6. y = (1, 0, 1, 0).
Step 3. Choose unit Y_3 to update its activation:
Step 4. $y\_in_3 = x_3 + \sum_j y_j w_{j3} = 1 + 1$.
Step 5. $y\_in_3 > 0 \rightarrow y_3 = 1$.
Step 6. y = (1, 0, 1, 0).
Step 3. Choose unit Y_2 to update its activation:
Step 4. $y\_in_2 = x_2 + \sum_j y_j w_{j2} = 0 + 2$.
Step 5. $y\_in_2 > 0 = y_2 = 1$.
Step 6. y = (1, 1, 1, 0).
Step 7. Test for convergence.

Since some activations have changed during this update cycle, at least one more pass through all of the input vectors should be made. The reader can confirm that further iterations do not change the activation of any unit. The net has converged to the stored vector.

Analysis

Energy Function. Hopfield [1984] proved that the discrete net bearing his name will converge to a stable limit point (pattern of activation of the units) by considering an energy (or Lyapunov) function for the system. An energy function is a function that is bounded below and is a nonincreasing function of the state of the system. For a neural net, the state of the system is the vector of activations of the units. Thus, if an energy function can be found for an iterative neural net, the net will converge to a stable set of activations. An energy function for the discrete Hopfield net is given by

$$E = -.5 \sum_{i \neq j} \sum_j y_i y_j w_{ij} - \sum_i x_i y_i + \sum_i \theta_i y_i.$$

If the activation of the net changes by an amount Δy_i, the energy changes by an amount

$$\Delta E = -\left[\sum_j y_j w_{ij} + x_i - \theta_i\right] \Delta y_i.$$

(This relationship depends on the fact that only one unit can update its activation at a time.)

We now consider the two cases in which a change Δy_i will occur in the activation of neuron Y_i.

If y_i is positive, it will change to zero if

$$x_i + \sum_j y_j w_{ji} < \theta_i.$$

This gives a negative change for y_i. In this case, $\Delta E < 0$.

If y_i is zero, it will change to positive if

$$x_i + \sum_j y_j w_{ji} > \theta_i.$$

This gives a positive change for y_i. In this case, $\Delta E < 0$.

Thus Δy_i is positive only if $[\sum_j y_j w_{ji} + x_i - \theta_i]$ is positive, and Δy_i is negative only if this same quantity is negative. Therefore, the energy cannot increase. Hence, since the energy is bounded, the net must reach a stable equilibrium such that the energy does not change with further iteration.

This proof shows that it is not necessary that the activations be binary. It is also not important whether the external signals are maintained during the iterations. The important aspects of the algorithm are that the energy change depend only on the change in activation of one unit and that the weight matrix be symmetric with zeros on the diagonal.

Storage Capacity. Hopfield found experimentally that the number of binary patterns that can be stored and recalled in a net with reasonable accuracy, is given approximately by

$$P \approx 0.15n,$$

where n is the number of neurons in the net.

Abu-Mostafa and St Jacques (1985) have performed a detailed theoretical analysis of the information capacity of a Hopfield net. For a similar net using bipolar patterns, McEliece, Posner, Rodemich, and Venkatesh (1987) found that

$$P \approx \frac{n}{2 \log_2 n}.$$

3.5 BIDIRECTIONAL ASSOCIATIVE MEMORY (BAM)

We now consider several versions of the heteroassociative recurrent neural network, or bidirectional associative memory (BAM), developed by Kosko (1988, 1992a).

A bidirectional associative memory [Kosko, 1988] stores a set of pattern associations by summing bipolar correlation matrices (an n by m outer product matrix for each pattern to be stored). The architecture of the net consists of two layers of neurons, connected by directional weighted connection paths. The net iterates, sending signals back and forth between the two layers until all neurons reach equilibrium (i.e., until each neuron's activation remains constant for several steps). Bidirectional associative memory neural nets can respond to input to either layer. Because the weights are bidirectional and the algorithm alternates between updating the activations for each layer, we shall refer to the layers as the X-layer and the Y-layer (rather than the input and output layers).

Three varieties of BAM—binary, bipolar, and continuous—are considered

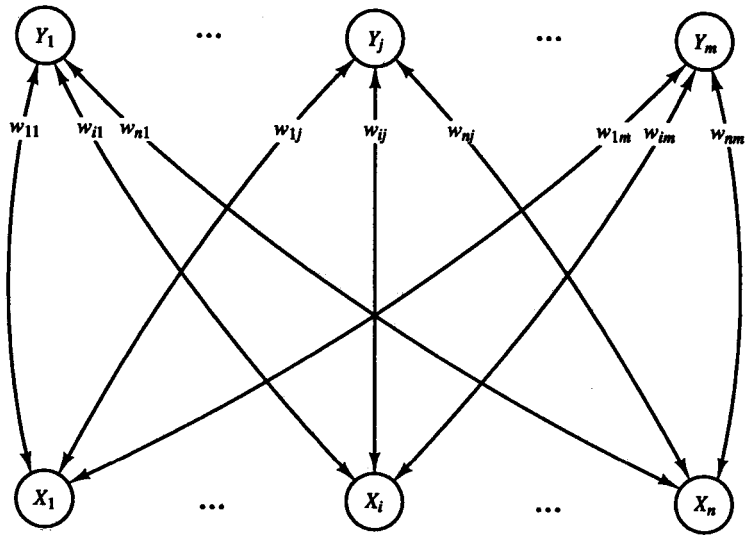

Figure 3.8 Bidirectional associative memory.

here. Several other variations exist. The architecture for each is the same and is illustrated in Figure 3.8.

3.5.1 Architecture

The single-layer nonlinear feedback BAM network (with heteroassociative content-addressable memory) has n units in its X-layer and m units in its Y-layer. The connections between the layers are bidirectional; i.e., if the weight matrix for signals sent from the X-layer to the Y-layer is $\mathbf{W}$, the weight matrix for signals sent from the Y-layer to the X-layer is $\mathbf{W}^T$.

3.5.2 Algorithm

Discrete BAM

The two bivalent (binary or bipolar) forms of BAM are closely related. In each, the weights are found from the sum of the outer products of the bipolar form of the training vector pairs. Also, the activation function is a step function, with the possibility of a nonzero threshold. It has been shown that bipolar vectors improve the performance of the net [Kosko, 1988; Haines & Hecht-Nielsen, 1988].

Setting the Weights. The weight matrix to store a set of input and target vectors $\mathbf{s}(p):\mathbf{t}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$

and

$$\mathbf{t}(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p)),$$

can be determined by the Hebb rule. The formulas for the entries depend on whether the training vectors are binary or bipolar. For binary input vectors, the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p (2s_i(p) - 1)(2t_j(p) - 1).$$

For bipolar input vectors, the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p s_i(p)t_j(p).$$

Activation Functions. The activation function for the discrete BAM is the appropriate step function, depending on whether binary or bipolar vectors are used.

For binary input vectors, the activation function for the Y-layer is

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > 0 \\ y_j & \text{if } y\_in_j = 0 \\ 0 & \text{if } y\_in_j < 0, \end{cases}$$

and the activation function for the X-layer is

$$x_i = \begin{cases} 1 & \text{if } x\_in_i > 0 \\ x_i & \text{if } x\_in_i = 0 \\ 0 & \text{if } x\_in_i < 0. \end{cases}$$

For bipolar input vectors, the activation function for the Y-layer is

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > \theta_j \\ y_j & \text{if } y\_in_j = \theta_j \\ -1 & \text{if } y\_in_j < \theta_j, \end{cases}$$

and the activation function for the X-layer is

$$x_i = \begin{cases} 1 & \text{if } x\_in_i > \theta_i \\ x_i & \text{if } x\_in_i = \theta_i \\ -1 & \text{if } x\_in_i < \theta_i. \end{cases}$$

Note that if the net input is exactly equal to the threshold value, the activation function "decides" to leave the activation of that unit at its previous value. For that reason, the activations of all units are initialized to zero in the algorithm that follows. The algorithm is written for the first signal to be sent from the X-layer to the Y-layer. However, if the input signal for the X-layer is the zero vector, the input signal to the Y-layer will be unchanged by the activation function, and the process will be the same as if the first piece of information had been sent from the Y-layer to the X-layer. Signals are sent only from one layer to the other at any step of the process, not simultaneously in both directions.

Algorithm.

Step 0. Initialize the weights to store a set of P vectors;
initialize all activations to 0.

Step 1. For each testing input, do Steps 2–6.

Step 2a. Present input pattern **x** to the X-layer
(i.e., set activations of X-layer to current input pattern).

Step 2b. Present input pattern **y** to the Y-layer.
(Either of the input patterns may be the zero vector.)

Step 3. While activations are not converged, do Steps 4–6.

Step 4. Update activations of units in Y-layer.
Compute net inputs:

$$y\_in_j = \sum_i w_{ij}x_i.$$

Compute activations:

$$y_j = f(y\_in_j).$$

Send signal to X-layer.

Step 5. Update activations of units in X-layer.
Compute net inputs:

$$x\_in_i = \sum_j w_{ij}y_j.$$

Compute activations:

$$x_i = f(x\_in_i).$$

Send signal to Y-layer.

Step 6. Test for convergence:
If the activation vectors **x** and **y** have reached equilibrium, then stop; otherwise, continue.

Continuous BAM

A continuous bidirectional associative memory [Kosko, 1988] transforms input smoothly and continuously into output in the range [0, 1] using the logistic sigmoid function as the activation function for all units.

For binary input vectors $(\mathbf{s}(p), \mathbf{t}(p))$, $p = 1, 2, \ldots, P$, the weights are determined by the aforementioned formula

$$w_{ij} = \sum_p (2s_i(p) - 1)(2t_j(p) - 1).$$

The activation function is the logistic sigmoid

$$f(y\_in_j) = \frac{1}{1 + \exp(-y\_in_j)},$$

where a bias is included in calculating the net input to any unit

$$y\_in_j = b_j + \sum_i x_i w_{ij},$$

and corresponding formulas apply for the units in the X-layer.

A number of other forms of BAMs have been developed. In some, the activations change based on a differential equation known as Cohen-Grossberg activation dynamics [Cohen & Grossberg, 1983]. Note, however, that Kosko uses the term "activation" to refer to the activity level of a neuron before the output function (such as the logistic sigmoid function) is applied. (See Kosko, 1992a, for further discussion of bidirectional associative memory nets.)

3.5.3 Application

Example 3.23 A BAM net to associate letters with simple bipolar codes

Consider the possibility of using a (discrete) BAM network (with bipolar vectors) to map two simple letters (given by 5 × 3 patterns) to the following bipolar codes:

(-1, 1) (1, 1)

The weight matrices are:

(to store $A \rightarrow -11$) ($C \rightarrow 11$) (W, to store both)

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & -2 \\ 0 & 2 \\ 2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ -2 & 0 \\ 2 & 0 \\ 0 & 2 \end{bmatrix}$$

To illustrate the use of a BAM, we first demonstrate that the net gives the correct Y vector when presented with the x vector for either the pattern A or the pattern C:

INPUT PATTERN *A*

$$(-1\ 1\ -1\ 1\ -1\ 1\ 1\ 1\ 1\ 1\ -1\ 1\ 1\ -1\ 1)\mathbf{W} = (-14, 16) \rightarrow (-1, 1).$$

INPUT PATTERN *C*

$$(-1\ 1\ 1\ 1\ -1\ -1\ 1\ -1\ -1\ 1\ -1\ -1\ -1\ 1\ 1)\mathbf{W} = (14, 16) \rightarrow (1, 1).$$

To see the bidirectional nature of the net, observe that the Y vectors can also be used as input. For signals sent from the Y-layer to the X-layer, the weight matrix is the transpose of the matrix $\mathbf{W}$, i.e.,

$$\mathbf{W}^T = \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}.$$

For the input vector associated with pattern A, namely, $(-1, 1)$, we have

$$(-1, 1)\mathbf{W}^T =$$

$$(-1, 1)\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ -2\ \ 2\ \ -2\ \ 2\ \ 2\ \ 2\ \ 2\ \ 2\ \ -2\ \ 2\ \ 2\ \ -2\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ 1\ \ -1\ \ 1).$$

This is pattern A.

Similarly, if we input the vector associated with pattern C, namely, $(1, 1)$, we obtain

$$(1, 1)\mathbf{W}^T =$$

$$(1, 1)\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ 2\ \ 2\ \ -2\ \ -2\ \ 2\ \ -2\ \ -2\ \ 2\ \ -2\ \ -2\ \ -2\ \ 2\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ 1\ \ 1\ \ -1\ \ -1\ \ 1\ \ -1\ \ -1\ \ 1\ \ -1\ \ -1\ \ -1\ \ 1\ \ 1),$$

which is pattern C.

The net can also be used with noisy input for the **x** vector, the **y** vector, or both, as is shown in the next example.

Example 3.24 Testing a BAM net with noisy input

In this example, the net is given a **y** vector as input that is a noisy version of one of the training **y** vectors and no information about the corresponding **x** vector (i.e., the **x** vector is identically 0). The input vector is (0, 1); the response of the net is

$$(0, 1)\mathbf{W}^T =$$

$$(0, 1)\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ 0\ \ 2\ \ -2\ \ 0\ \ 2\ \ 0\ \ 0\ \ 2\ \ -2\ \ 0\ \ 0\ \ 0\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ 0\ \ 1\ \ -1\ \ 0\ \ 1\ \ 0\ \ 0\ \ 1\ \ -1\ \ 0\ \ 0\ \ 0\ \ 1).$$

Note that the units receiving 0 net input have their activations left at that value, since the initial x vector is 0. This x vector is then sent back to the Y-layer, using the weight matrix W:

$$(-1 \;\; 1 \;\; 0 \;\; 1 \;\; -1 \;\; 0 \;\; 1 \;\; 0 \;\; 0 \;\; 1 \;\; -1 \;\; 0 \;\; 0 \;\; 0 \;\; 1) \begin{bmatrix} 0 & -2 \\ 0 & 2 \\ 2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ -2 & 0 \\ 2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$\rightarrow (0 \;\; 1).$$

This result is not too surprising, since the net had no information to give it a preference for either A or C. The net has converged (since, obviously, no further changes in the activations will take place) to a spurious stable state, i.e., the solution is not one of the stored pattern pairs.

If, on the other hand, the net was given both the input vector y, as before, and some information about the vector x, for example,

$$\mathbf{y} = (0 \;\; 1), \mathbf{x} = (0 \;\; 0 \;\; -1 \;\; 0 \;\; 0 \;\; 1 \;\; 0 \;\; 1 \;\; 1 \;\; 0 \;\; 0 \;\; 1 \;\; 1 \;\; -1 \;\; 0),$$

the net would be able to reach a stable set of activations corresponding to one of the stored pattern pairs.

Note that the x vector is a noisy version of

$$A = (-1 \;\; 1 \;\; -1 \;\; 1 \;\; -1 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; -1 \;\; 1 \;\; 1 \;\; -1 \;\; 1),$$

where the nonzero components are those that distinguish A from

$$C = (-1 \;\; 1 \;\; 1 \;\; 1 \;\; -1 \;\; -1 \;\; 1 \;\; -1 \;\; -1 \;\; 1 \;\; -1 \;\; -1 \;\; -1 \;\; 1 \;\; 1).$$

Now, since the algorithm specifies that if the net input to a unit is zero, the activation of that unit remains unchanged, we get

$$(0, 1)\mathbf{W}^{\mathrm{T}} =$$

$$(0, 1) \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2 \;\; 2 \;\; 0 \;\; 2 \;\; -2 \;\; 0 \;\; 2 \;\; 0 \;\; 0 \;\; 2 \;\; -2 \;\; 0 \;\; 0 \;\; 0 \;\; 2)$$

$$\rightarrow (-1 \;\; 1 \;\; -1 \;\; 1 \;\; -1 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; -1 \;\; 1 \;\; 1 \;\; -1 \;\; 1),$$

which is pattern A.

Since this example is fairly extreme, i.e., every component that distinguishes A from C was given an input value for A, let us try something with less information given concerning x.

For example, let $\mathbf{y} = (0\ 1)$ and $\mathbf{x} = (0\ 0\ -1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$. Then

$$(0, 1)\mathbf{W}^T =$$

$$(0, 1)\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ 0\ \ 2\ \ -2\ \ 0\ \ 2\ \ 0\ \ 0\ \ 2\ \ -2\ \ 0\ \ 0\ \ 0\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1\ \ -1\ \ 0\ \ 0\ \ 0\ \ 1),$$

which is not quite pattern A.

So we try iterating, sending the $\mathbf{x}$ vector back to the Y-layer using the weight matrix W:

$$(-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1\ \ -1\ \ 0\ \ 0\ \ 0\ \ 1)\begin{bmatrix} 0 & -2 \\ 0 & 2 \\ 2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ -2 & 0 \\ 2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$\rightarrow (-6, 10) \rightarrow (-1, 1).$$

If this pattern is fed back to the X-layer one more time, the pattern A will be produced.

Hamming distance

The number of different bits in two binary or bipolar vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ is called the *Hamming distance* between the vectors and is denoted by $H[\mathbf{x}_1, \mathbf{x}_2]$. The average Hamming distance between the vectors is $\frac{1}{n}H[\mathbf{x}_1, \mathbf{x}_2]$, where n is the number of components in each vector. The $\mathbf{x}$ vectors in Examples 3.23 and 3.24, namely,

differ in the 3rd, 6th, 8th, 9th, 12th, 13th, and 14th positions. This gives an average Hamming distance between these vectors of 7/15. The average Hamming distance between the corresponding $\mathbf{y}$ vectors is 1/2.

Kosko (1988) has observed that "correlation encoding" (as is used in the BAM neural net) is improved to the extent that the average Hamming distance between pairs of input patterns is comparable to the average Hamming distance between the corresponding pairs of output patterns. If that is the case, input patterns that are separated by a small Hamming distance are mapped to output vectors that are also so separated, while input vectors that are separated by a large Hamming distance go to correspondingly distant (dissimilar) output patterns. This is analogous to the behavior of a continuous function.

Erasing a stored association

The complement of a bipolar vector $\mathbf{x}$ is denoted $\mathbf{x}^c$; it is the vector formed by changing all of the 1's in vector $\mathbf{x}$ to -1's and vice versa. Encoding (storing the pattern pair) $\mathbf{s}^c:\mathbf{t}^c$ stores the same information as encoding $\mathbf{s}:\mathbf{t}$; encoding $\mathbf{s}^c:\mathbf{t}$ or $\mathbf{s}:\mathbf{t}^c$ will erase the encoding of $\mathbf{s}:\mathbf{t}$ [Kosko, 1988].

3.5.4 Analysis

Several strategies may be used for updating the activations. The algorithm described in Section 3.5.2 uses a synchronous updating procedure, namely, that all units in a layer update their activations simultaneously. Updating may also be simple asynchronous (only one unit updates its activation at each stage of the iteration) or subset asynchronous (a group of units updates all of its members' activations at each stage).

Energy function

The convergence of a BAM net can be proved using an energy or Lyapunov function, in a manner similar to that described for the Hopfield net. A Lyapunov function must be decreasing and bounded. For a BAM net, an appropriate function is the average of the signal energy for a forward and backward pass:

$$L = -0.5\,(\mathbf{x}W\mathbf{y}^T + \mathbf{y}W^T\mathbf{x}^T).$$

However, since $\mathbf{x}W\mathbf{y}^T$ and $\mathbf{y}W^T\mathbf{x}^T$ are scalars, and the transpose of a scalar is a scalar, the preceding expression can be simplified to

$$\begin{aligned} L &= -\mathbf{x}W\mathbf{y}^T \\ &= -\sum_{j=1}^{m}\sum_{i=1}^{n} x_i w_{ij} y_j. \end{aligned}$$

For binary or bipolar step functions, the Lyapunov function is clearly bounded below by

$$-\sum_{j=1}^{m}\sum_{i=1}^{n} |\, w_{ij} \,|.$$

Kosko [1992a] presents a proof that the Lyapunov function decreases as the net iterates, for either synchronous or subset asynchronous updates.

Storage capacity

Although the upper bound on the memory capacity of the BAM is min(n, m), where n is the number of X-layer units and m is the number of Y-layer units, Haines and Hecht-Nielsen [1988] have shown that this can be extended to min (2^n, 2^m) if an appropriate nonzero threshold value is chosen for each unit. Their choice was based on a combination of heuristics and an exhaustive search.

BAM and Hopfield nets

The discrete Hopfield net (Section 3.4.4) and the BAM net are closely related. The Hopfield net can be viewed as an autoassociative BAM with the X-layer and Y-layer treated as a single layer (because the training vectors for the two layers are identical) and the diagonal of the symmetric weight matrix set to zero.

On the other hand, the BAM can be viewed as a special case of a Hopfield net which contains all of the X- and Y-layer neurons, but with no interconnections between two X-layer neurons or between two Y-layer neurons. This requires all X-layer neurons to update their activations before any of the Y-layer neurons update theirs; then all Y field neurons update before the next round of X-layer updates. The updates of the neurons within the X-layer or within the Y-layer can be done at the same time because a change in the activation of an X-layer neuron does not affect the net input to any other X-layer unit and similarly for the Y-layer units.

3.6 SUGGESTIONS FOR FURTHER STUDY

3.6.1 Readings

The original presentation of the Hebb rule is given in *The Organization of Behavior* [Hebb, 1949)]. The Introduction and Chapter 4 are reprinted in Anderson and Rosenfeld [1988], pp. 45–56. The articles by Anderson and Kohonen that are included in the Anderson and Rosenfeld collections, as well as Kohonen's book, *Self-organization and Associative Memory,* (1989a) provide good discussions of the associative memory nets presented in this chapter.

For further discussion of the Hopfield net, the original articles included in the Anderson and Rosenfeld collection give additional background and development. The article by Tank and Hopfield (1987) in *Scientific American* is also recommended.

The discussion of BAM nets in *Neural Networks and Fuzzy Systems* [Kosko, 1992a] provides a unified treatment of these nets and their relation to Hopfield