# Artificial Neural Networks
# Lecture Notes

**Stephen Lucci, PhD**

---

## Part 11

About this file:

- This is the printer-friendly version of the file "*lecture11.htm*". In case the page is not properly displayed, use IE 5 or higher.
- Since this is an offline page, each file "lecture*nn*.htm" (for instance "lecture11.htm") is accompanied with a "img*nn*" folder (for instance "img11") containing the images which make part of the notes. So, to see the images, Each html file must be kept in the same directory (folder) as its corresponding "img*nn*" folder.
- If you have trouble reading the contents of this file, or in case of transcription errors, email gi0062@bcmail.brooklyn.cuny.edu
- Acknowledgments:
Background image is from http://www.anatomy.usyd.edu.au/online/neuroanatomy/tutorial1/tutorial1.html (edited) at the University of Sydney Neuroanatomy web page. Mathematics symbols images are from metamath.org's GIF images for Math Symbols web page. Some images are scans from R. Rojas, *Neural Networks* (Springer-Verlag, 1996), as well as from other books to be credited in a future revision of this file. Some image credits may be given where noted, the remainder are native to this file.

## Contents



---

## *Associative Memory Networks*

- Remembering something: Associating an idea or thought with a sensory cue.
- Human memory connects items (ideas, sensations, &c.) that are similar, that are contrary, that occur in close proximity, or that occur in close succsession
  - *Aristotle*
- An input stimulus which is similar to the stimulus for the association will invoke the associated response pattern.
  - A woman's perfume on an elevator...
  - A song on the radio...
  - An old photograph...
- An *Associative Memory Net* may serve as a highly simplified model of human memory.
- These associative memory units should not be confused with *Content Addressable Memory Units*.

## A Taxonomy of Associative Memories

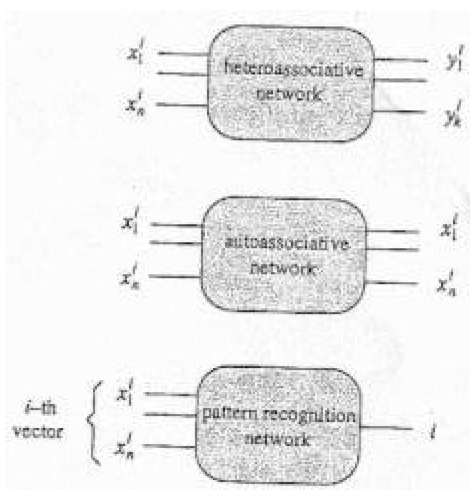



Fig. 12.1. Types of associative networks

**The superscripts of x and y are all i**

- *Heteroassociative network*
  Maps n input vectors $\bar{x}^1$, $\bar{x}^2$, ..., $\bar{x}^n$, in n-dimensional space
  to m output vectors $\bar{y}^1$, $\bar{y}^2$, ..., $\bar{y}^m$, in m-dimensional space,

  $\bar{x}^i \mapsto \bar{y}^i$

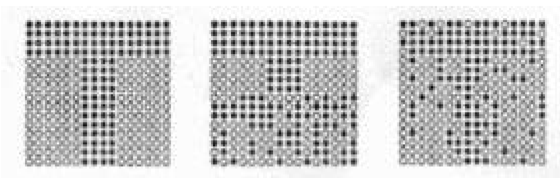
if $\|\tilde{\bar{x}} - \bar{x}^i\|^2 < \varepsilon$ then $\tilde{\bar{x}} \rightarrow \bar{y}^i$

- *Autoassociative Network*
  A type of heteroassociative network.
  Each vector is associated with itself; i.e.,

  $\bar{x}^i = \bar{y}^i$ , i = 1, ..., n.
  Features *correction of noisy input vectors*.

- *Pattern Recognition Network*
  A type of heteroassociative network.
  Each vector $\bar{x}^i$ is associated with the scalar i.
  [*illegible - remainder cut-off in photocopy*]

## An Example of Associative Recall



To the left is a binarized version of the letter "T".

The middle picture is the same "T" but with the bottom half replaced by noise. Pixels have been assigned a value 1 with probability 0.5
Upper half: The cue
Bottom half: has to be recalled from memory.

The pattern on the right is obtained from the original "T" by adding 20% noise. Each pixel is inverted with probability 0.2.
The whole memory is available, but in an imperfectly recalled form ("hazy" or inaccurate memory of some scene.)

*(Compare/contrast the following with database searches)*
In each case, when part of the pattern of data is presented in the form of a sensory cue, the rest of the pattern (memory) is associated with it.

Alternatively, we may be offered an *imperfect* version of the...
[*illegible - remainder cut-off in photocopy*]

## Hebbian Learning

*Donald Hebb* - psychologist, 1949.

Two neurons which are simultaneously active should develop a degree of interaction higher than those neurons whose activities are uncorrelated.

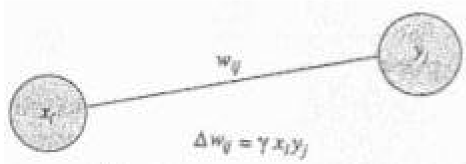



Fig. 12.4. The Hebb rule

Input $x_i$
Output $y_j$

weight update $\Delta w_{ij} = \gamma x_i y_j$

## Hebb Rule for Pattern Association

It can be used with patterns that are represented as either binary or bipolar vectors.

- Training Vector Pairs $\bar{s} : \bar{t}$
- Testing Input Vector $\bar{x}$ (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 ($i = 1, \ldots, n$):

$$x_i = s_i$$

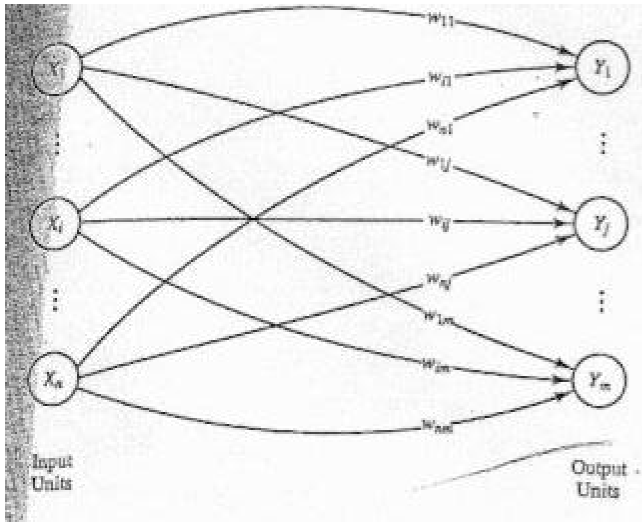
*Step 3.* Set activations for output units to current target output ($j = 1, \ldots, m$):

$$y_j = t_j.$$

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

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

In this simple form of Hebbian Learning, one generally employs *outer product* calculations instead.




**Architecture of a Heteroassociative Neural Net**

**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

$$x_i.$$

*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:

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

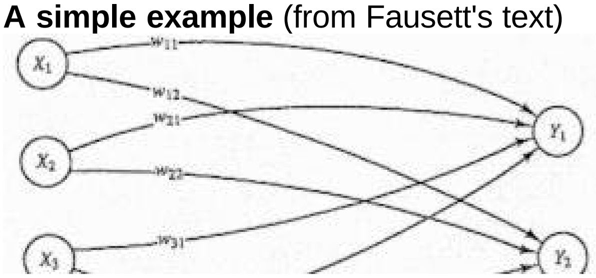
(for bipolar targets).

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

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

**A simple example** (from Fausett's text)




Heteroassociative network.
Input vectors - 4 components
Output vectors - 2 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) |

The input vectors are not mutually orthogonal (i.e., the dot product ≠ 0,) - in which case the response will include a portion of each of their target values - *cross-talk*.

Note: target values are chosen to be related to the input vectors in a simple manner. The *cross-talk* between the first and second input vectors does not pose any difficulties (since these target values...
[Illegible - cut off in photocopy]

The training is accomplished by the Hebb rule:

- $w_{ij}(\text{new}) = w_{ij}(\text{old}) + s_i t_j$
  ie, $\Delta w_{ij} = s_i t_j$ , $\alpha = 1$.

**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 3.3 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

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

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.*
$$\begin{aligned} y\_in_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} \\ &= 1(2) + 0(1) + 0(0) + 0(0) \\ &= 2; \end{aligned}$$
$$\begin{aligned} y\_in_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} \\ &= 1(0) + 0(0) + 0(1) + 0(2) \\ &= 0. \end{aligned}$$

*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.*
$$\begin{aligned} y\_in_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} \\ &= 1(2) + 1(1) + 0(0) + 0(0) \\ &= 3; \end{aligned}$$
$$\begin{aligned} y\_in_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} \\ &= 1(0) + 1(0) + 0(1) + 0(2) \\ &= 0. \end{aligned}$$

*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.)

testing - cont'd:

*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.)

We can employ vector-matrix rotation to illustrate the testing process.

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

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

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

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

or, in slightly more compact notation,

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

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 **x** equal to each of the other three training input vectors, yields

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

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

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

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

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

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

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

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.

- A <u>bipolar representation</u> would be preferable. More robust in the presence of noise.

- The weight matrix obtained from the previous examples would be:

$$w = \begin{bmatrix} 4 & -4 \\ 2 & -2 \\ -2 & 2 \\ -4 & 4 \end{bmatrix}$$

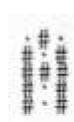
with two "mistakes".
Trouble remains:
ie, (-1, 1, 1, -1) · w = (0,0) → (0,0).

- <u>However</u>, the net can respond correctly when given an input vector with <u>two components missing</u>.
e.g., X = (0, 1, 0, -1) formed from S = (1, 1,-1, -1) with the first and third components missing rather than wrong.
(0, 1, 0, -1) · w = (6,-6) = (1,1) which is the ...
[illegible - cut off in photocopy]

## Character Recognition Example

**(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 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 1,-1,1 1,1,1 1,-1,1 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.
The figure below shows the vector pairs in their original two-dimensional form.

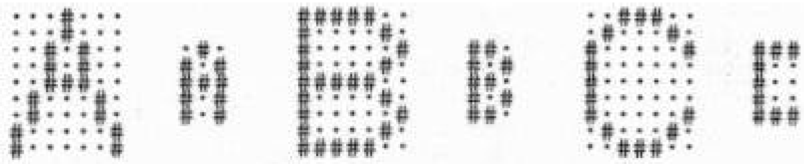


Figure 3.3 Training patterns for character recognition using heteroassociative net.

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 (below). 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 (below) shows that the neural net can recognize the small letters that are stored in it, even when given input patterns representing the large training patterns with 30% noise.

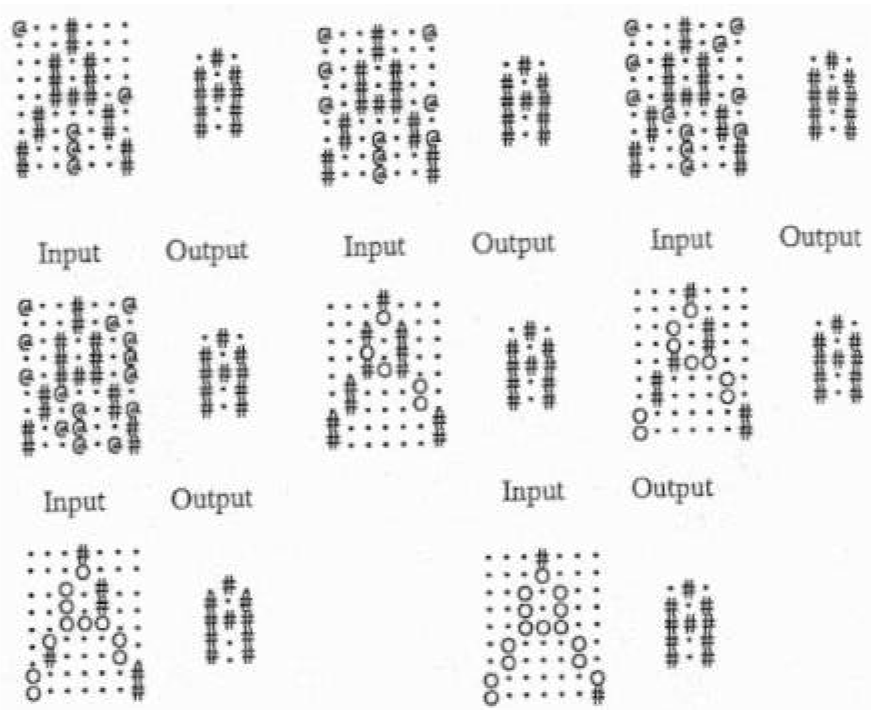



Figure 3.4 Response of heteroassociative net to several noisy versions of pattern A.

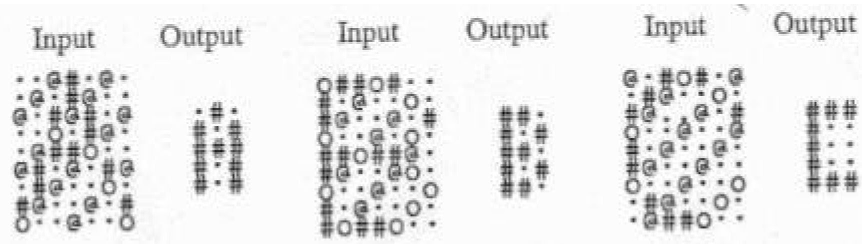
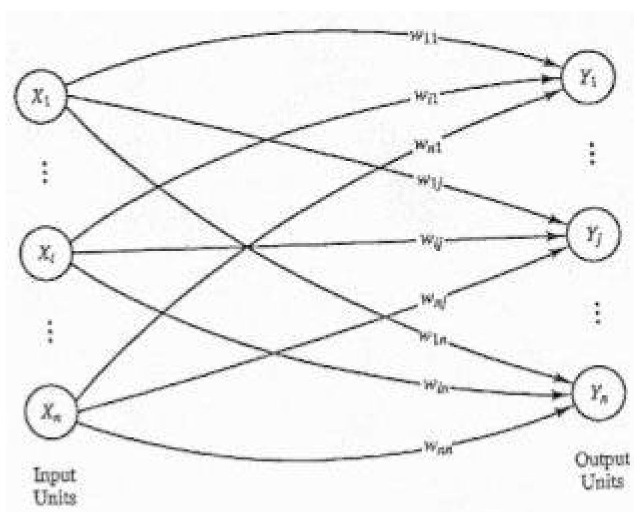



Figure 3.5 Response of heteroassociative net to patterns A, B, and C with mistakes in 1/3 of the components.

---

## Autoassociative Nets

- 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 close to it.
- The performance of the net is judged by its ability to reproduce a stored pattern from noisy input; performance is generally better for bipolar vectors than for binary vectors.




**Architecture of an Autoassociative neural net**

It is common for weights on the diagonal (those which connect an input pattern component to the corresponding component in the output pattern) to be set to zero.

### 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, $i = 1, \ldots, n$; $j = 1, \ldots, m$:

$$w_{ij} = 0;$$

*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, $j = 1, \ldots, m$:

$$y_j = s_j;$$

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

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

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.

## Application and examples of Autoassociative Nets

### 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 ($j = 1, \ldots, n$):

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

**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.* $y\_in = (4, 4, 4, -4)$.

*Step 4.* $\mathbf{y} = f(4, 4, 4, -4) = (1, 1, 1, -1)$.

Since the response vector $\mathbf{y}$ 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 vecto:**

Using the succinct notation just introduced, consider the performance of the net for each of the input vectors $\mathbf{x}$ that follow. Each vector $\mathbf{x}$ is formed from the original stored vector $\mathbf{s}$ 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.

## 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 vectr r. 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 $W_1 + W_2$ be the weight matrix to store the orthogonal vectors $(1, 1, -1, -1)$ and $(-1, 1, 1, -1)$ and $W_3$ be the weight matrix that stores $(-1, 1, -1, 1)$. Then the weight matrix to store all three orthogonal vectors is $W_1 + W_2 + W_3$. We have

$$\underset{W_1 + W_2}{\begin{bmatrix} 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & -2 \\ -2 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \end{bmatrix}} + \underset{W_3}{\begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}} = \underset{W_1 + W_2 + W_3}{\begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}},$$

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 $W_4$, 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

$$\underset{W_1 + W_2 + W_3}{\begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}} + \underset{W_4}{\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}} = \underset{W^*}{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}},$$

which cannot recognize any vector.

## 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.