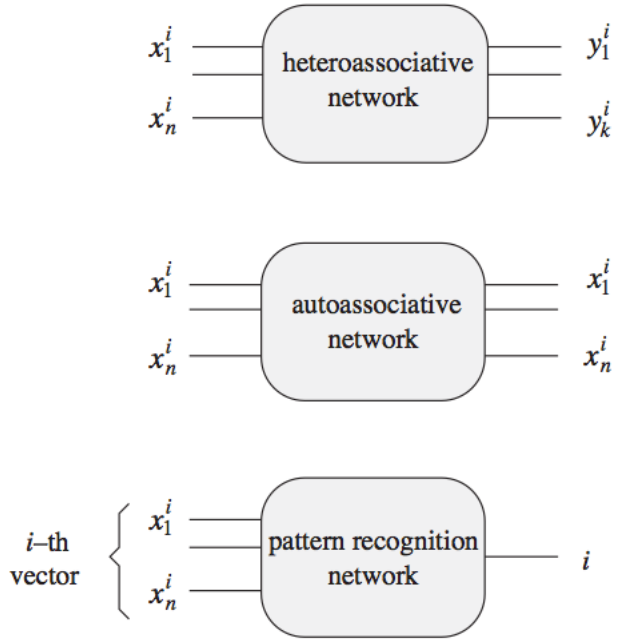
Pattern Association or Associative Networks

Jugal Kalita
University of Colorado at Colorado Springs

- To an extent, learning is forming associations.
- Human memory associates
  - similar items,
  - contrary/opposite items,
  - items close in proximity,
  - items close in succession (e.g., in a song)c etc.

Types of Associative Networks

- Heteroassociative Networks
- Autoassociative Networks
- Pattern Recognition Networks
- Recurrent Networks



Types of Associative Networks

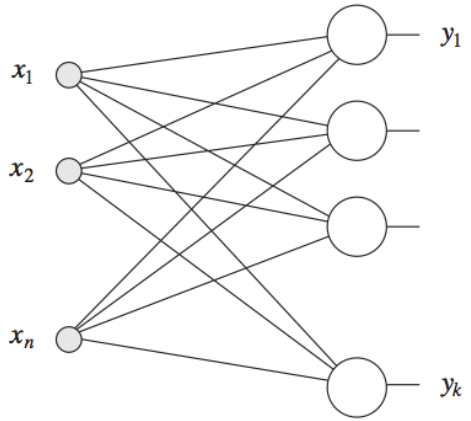


Fig. 12.2. Heteroassociative network without feedback

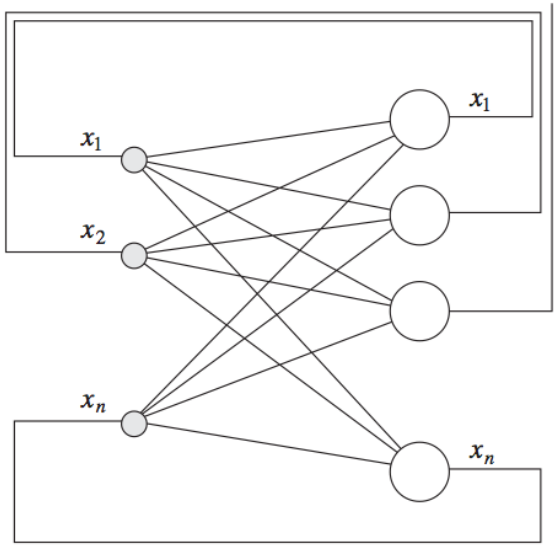


Fig. 12.3. Autoassociative network with feedback

Terminologies

- An *associative network* is a single-layer net in which the weights are determined in such a way that the net can store a set of pattern associations.
- Each *association* is an input-output vector pair **s**:**t**
- *Auto-associative Network*: If vector **t** is the same as **s**, the net is auto-associative.

Terminologies

- If the **t**'s are different from the **s**'s, the net is *hetero-associative*.
- Whether auto- or hetero-associative, the net can associate not only the exact pattern pairs used in training, but is also able to obtain associations if the input is *similar* to one on which it has been trained.

Examples of Hetero-association

- If the **t**'s are different from the **s**'s, the net is *hetero-associative*.
- Example 1: Mapping from 4-inputs to 2-outputs. Whenever the net is shown a 4-bit input pattern, it produces a 2-bit output pattern
- Input Output

  1 0 0 0 1 0

  1 1 0 0 1 0

  0 0 0 1 0 1

  0 0 1 1 0 1

Examples of Heteroassociation

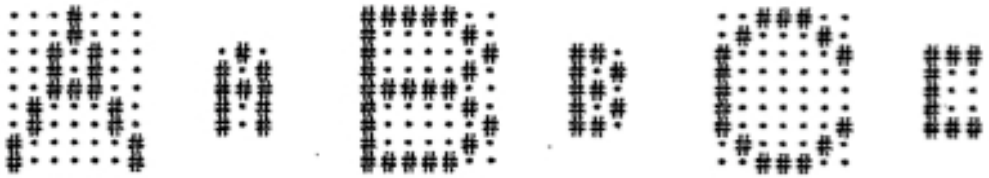
- Example 2: Input similar to a training input
- The net has been trained on the mapping:

Input	Output
1 0 0 0	1 0
1 1 0 0	1 0
0 0 0 1	0 1
0 0 1 1	0 1

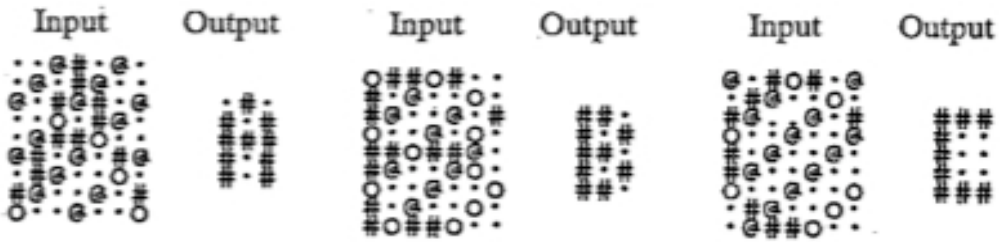
Now, if the input is x = (0 1 0 0), which is different from the second training input in one location, the net still produces (1 0) as the output.

Examples of Heteroassociation

- Example 3: Character Recognition
- We can train the net to learn the mapping from the larger patterns to the smaller patterns, pairwise:



Now, if the input is noisy in some places, the net still produces the same map. Here up to 1/3rd of the input bits are erroneous.



Example of Auto-association

- Example 1: We can train a 4 input, 4 output network to store just one vector **s** = (1 1 1 -1)
- It will remember this vector **s**, i.e., if we give **s** as input it will produce **s** as output.
- In addition, if there are some errors in the input, it will be able to map it to **s** as well.
- For example, it does the following maps:

(-1 1 1 -1) → (1 1 1 -1)

(1 -1 1 -1) → (1 1 1 -1)

(1 1 -1 -1) → (1 1 1 -1)

Architecture of Associative Networks and Learning Rules

- The networks we discuss in this chapter are *single-layer* networks.
- The architectures can be *feed-forward* or *recurrent*.
- In a feed-forward net, information flows from the input units to output nets.
- In a recurrent net, there are connections among the nodes that form closed loops.
- We discuss recurrent nets later.
- The learning rule used can either be the *Hebb Rule* or the *Delta Rule* or an extended version of the Delta Rule.
- The data and weight representation can be binary or bipolar.

Using Hebb Algorithm for Pattern Association

//It's the same algorithm as before

//initialize all weights

for i=1 to n {for j=1 to n { set $w_{ij} = 0$ 1 <= I}}

for each training pair **s**:**t** {

$x_i = s_i$

$y_j = t_j$

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

}

Example: Hetero/Hebb, using training algorithm

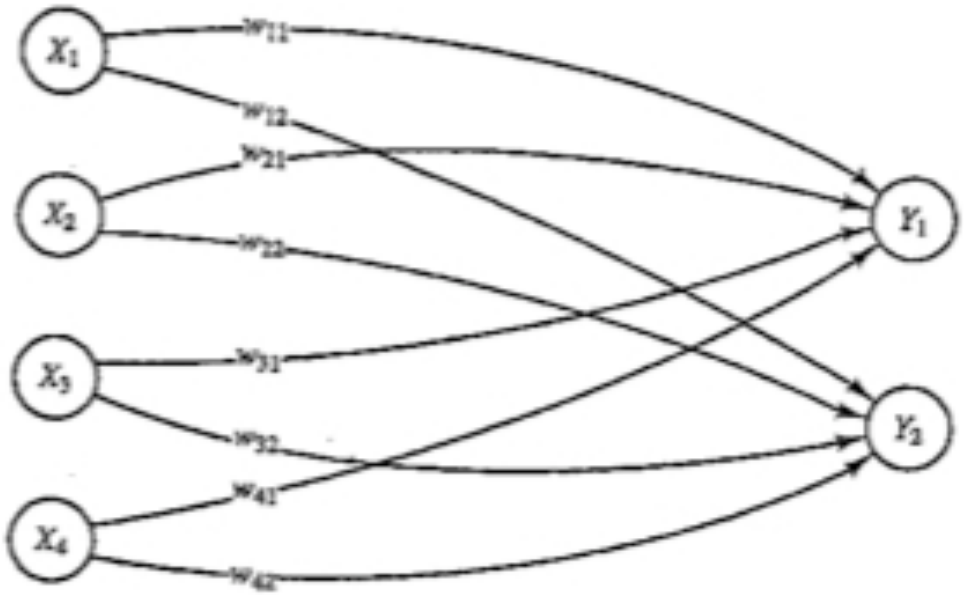
- Train a hetero-associative neural network using the Hebb Rule to learn the following mapping:
- Input Output

s1	s2	s3	s4	t1	t2
1	0	0	0	1	0
1	1	0	0	1	0
0	0	0	1	0	1
0	0	1	1	0	1

- The input vectors are not orthogonal. E.g., if we take the first two, 1x1 + 0 x 1 + 0 x 0 + 0 x 0 not 0. This sum must be 0 for the two vectors to be orthogonal.
- But because the target values are related to the input vectors in a simple manner, cross talk between first and second input vectors does not pose difficulties.

Example: Hetero/Hebb, using training algorithm

- Here is the network used: It has 4 input nodes and 2 output nodes
- Training rule used: Set $x_i = s_i$
- w_{ij} (new) = w_{ij} (old) + s_i t_j, i.e., $\Delta w_{ij} = s_i$ t_j



Example: Hetero/Hebb, using training algorithm

- **Training**: Only the weights that change at each step of the algorithm are shown.
- Initialize all weights to 0
- First **s**:**t** pair (1 0 0 0): (1 0)

  x1 = 1; x2 = x3 = x4 = 0 y1 = 1; y2 =0

  w11 (new) = w11 (old) + x1 y1 = 0 + 1 = 1

  (All other weights remain 0)

Second **s**:**t** pair (1 1 0 0): (1 0)

x1 = 1; x2 = 1; x3 = x4 = 0 y1 = 1; y2 = 0

w11 (new) = w11 (old) + x1 y1 = 1 + 1 = 2

w21 (new) = w21 (old) + x2 y1 = 0 + 1 = 1

(All other weights remain 0)

Example: Hetero/Hebb, using training algorithm

- Third **s**:**t** pair (0 0 0 1): (0 1)

  …

  Fourth **s**:**t** pair (0 0 1 1): (0 1)

  …

  The weight matrix becomes

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

Example: Hetero/Hebb, using training algorithm

The weight matrix is

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

Testing: We need to test to see if the net produces the correct output for each of the training inputs. The activation function used is $f(x) = 1$ if $x > 0$; 0 if $x \leq 0$

First Input: **x** = (1 0 0 0)

$y\_in1 = x1\ w11 + x2\ w21 + x3\ w31 + x4\ w41$

$= 1 * 2 + 0 * 1 + 0 * 0 + 0 * 0 = 2$

$y\_in2 = 0$

$y1 = f(y\_in1) = f(2) = 1$

$y2 = f(y\_in2) = f(0) = 0$

This is correct for the first input.

Similarly, we can show that the correct output is produced for the other inputs.

Example: Hetero/Hebb, using training algorithm

The weight matrix is

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

Testing with an input similar to a training input. Let **x** = (0 1 0 0) differ from the training vector **s2** = (1 1 0 0) in the first component.

If we compute the output, it comes out to be (1 0), the same as for **s2**.

Testing with an input not similar to the training inputs. Let **x** = (0 1 1 0). It is different from each training input in at least two components.

If we compute the output, it comes out to be (1 1). It is not one of the outputs for which the net has been trained. In other words, the net doesn't recognize the input.

Alternative to Training: Hebb Net Computations using matrix products

The weights found by Hebb rule, with all weights initially 0, can be obtained also by computing the outer product of input vector and output vector pairs.

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 $\mathbf{S} = \mathrm{s}^T$ and the $1 \times m$ matrix $\mathrm{T} = \mathrm{t}$:

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

This is just the weight matrix to store the association s:t found using the Hebb rule.

Alternative to Training: Hebb Net by matrix products

To store a set of associations $s(p) : t(p), p = 1, \ldots, P$, 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 $W = \{w_{ij}\}$ 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].

Hebb Rule Computations Using Outer Products

- We can solve the same problem, without going through all the Hebb Rule training, and obtain the weights for each of the training examples one by one by simply computing outer products individually.
- If the training examples are orthogonal, we can add the individual weights for each of the examples to obtain the weights for an associative network that stores all the training examples.

Hebb Rule Computations Using Outer Products

- Train a hetero-associative neural network using the Hebb Rule to learn the following mapping (Repeated from before)

Input				Output	
s1	s2	s3	s4	t1	t2
1	0	0	0	1	0
1	1	0	0	1	0
0	0	0	1	0	1
0	0	1	1	0	1

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

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

and

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

Hebb Rule Computations Using Outer Products

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,

Hebb Rule Computations Using Outer Products

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

Testing a Hebb-trained Hetero-Associative Net

- Suppose we have trained a hetero-associative network using Hebb rule as in the previous example, with mapping

  1000 →10, 1100 → 10, 0001 → 01, 0011 → 01
- Let's *now test using the training inputs*: (See Fausett, pp. 113-115)
- Testing (i.e., performing the computations with the learned weights) with 1000 produces 10. Similarly, 1100 produces 10, 0001 produces 01 and 0011 produces 01. In other words, *recall is perfect.*
- Testing with *slightly different input*:

  0100 produces 10 (0100 is different from the first or second training vectors in one position)

  0110 produces 11 (0110 is different from the input patterns in at least two positions). Thus, the output is not one of the trained outputs, it's something different.

Perfect Learning and Cross Talk in Hebb Net

- Whether Hebb rule is suitable for an associative net depends on how the input training vectors are related to each other.
- If the input vectors are *uncorrelated or orthogonal*, Hebb rule produces correct weights, and the response of the vector, when tested with one of the training vectors, will be perfect recall of the input vector's associated weight.
- In other words, orthogonal vectors can be learned perfectly.
- If the input vectors are not orthogonal, the response may be messed up because the storage is not perfect.
- This is called *Cross Talk*.

Delta Rule for Pattern Association

- Hebb rule is simple, and results in cross talk.
- So, we can use the Delta Rule for training as well.
- As we know, the Delta Rule was introduced in the 1960s for ADALINE by Widrow and Hoff
- When the input vectors are linearly independent, the Delta Rule produces exact solutions.
- Whether the input vectors are linearly independent or not, the Delta Rule produces a least squares solution, i.e., it optimizes for the lowest sum of least squared errors.

Delta Rule

- The original delta rule reduces the difference between the computed value of an output unit and the net input to it (*the cumulative squared error between the computed value and the net input*).
- **Delta Rule for input node i and output node j**

  $$w_{ij}(new) = w_{ij}(old) + \alpha(t_j - y_j)*x_i*1$$

- This original Delta Rule assumes that the activation function for the output units is the identity function.
- The original Delta Rule minimizes the square of the difference between net input to the output units and the target values.

Extended Delta Rule

- The original delta rule can be extended to allow for any arbitrary, differentiable activation function at the output units.
- The Extended Delta rule is a minor modification of the original Delta Rule.
- The goal in the Extended Delta rule is to reduce the difference (*the cumulative squared error) between the computed output and the target value*, rather than between the net input and the target value.
- Both our texts (Fausett or Rojas) derive the Extended Delta Rule
- **Extended Delta Rule**

$$w_{ij}(new) = w_{ij}(old) + \alpha(t_j - y_j)*x_i*f'(y_{in.j})$$

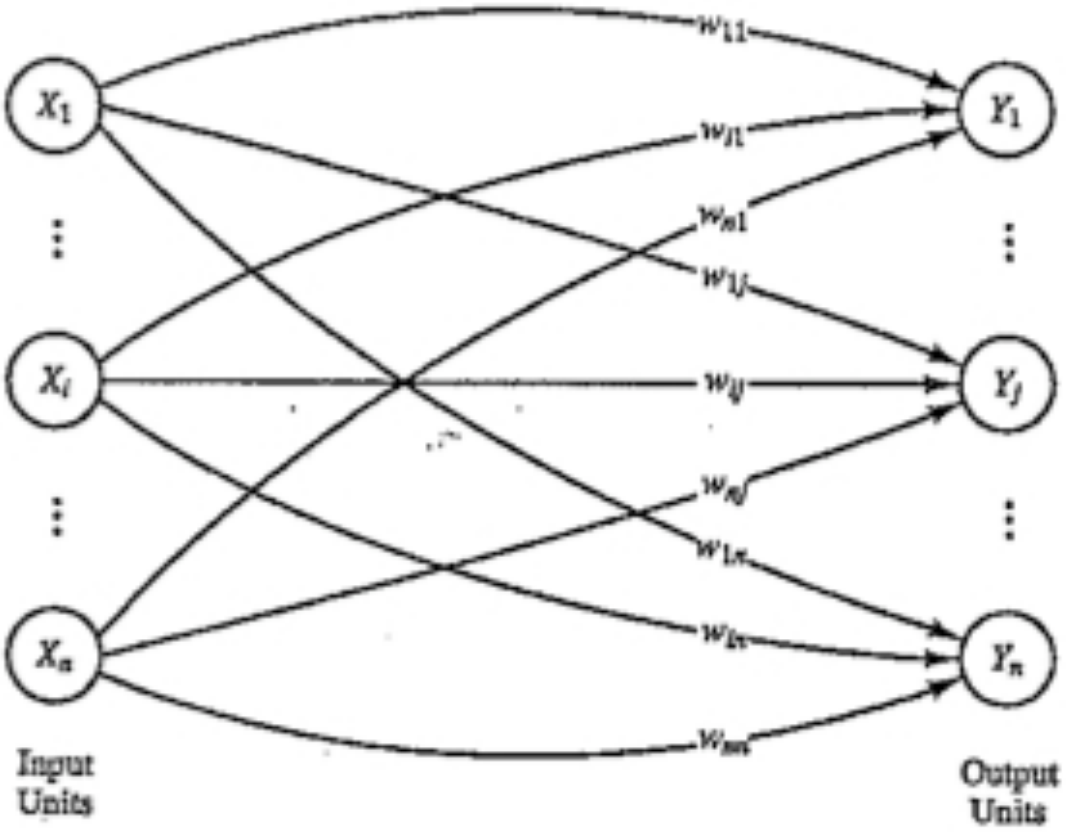
- You can try to work out the same examples with Extended Delta Rule as well.

Auto-Associative Nets

- Now, we move on to feed-forward auto-associative nets
- An auto-associative net remembers one or more patterns.
- For an auto-associative net, the training input and target output vectors are identical.
- The process of training is called *storing* the vectors.
- The representation can be bipolar or binary.
- Not only does it remember patterns exactly, but in addition, shown some degraded versions of a pattern (e.g., with missing components or with errors in components), it returns the original uncorrupted version of the pattern.
- Either Hebb Rule, Delta Rule or Extended Delta Rule can be used for training.
- It is often the case that for auto-associative nets, the diagonal weights (those which connect an input component to the corresponding output component) are set to 0. There are papers that say this helps learning.
- Setting diagonal elements to 0 is necessary if we use iterations (iterative nets) or the delta rule is used.

Autoassociative Net

- Architecture



Training Algorithm

- If the training vectors are orthogonal, we can use the Hebb rule algorithm given earlier.
- Otherwise, use Delta Rule or Extended Delta Rule for better learning results.

- Application: Find out whether an input vector is familiar or unfamiliar.

Autoassociate Example

- One can store the vector s = (1 1 1 -1) with the weight matrix

$$\begin{matrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{matrix}$$

One can test with the input vector and perform the computations to see that it is recalled.

One mistake: One can test with one mistake in the input vector s, say vectors such as (-1 1 1 -1), (1 -1 1 -1), (1 1 -1 -1) and (1 1 1 1) and see that (1 1 1 -1) is recalled.

Two mistakes: One can test with two mistakes, e.g., the vector (-1 -1 1 -1) and the net will return (0 0 0 0) saying the net doesn't recognize the vector.

Storage Capacity

- More than one vector can be stored in an auto-associative net.
- Up to n-1 bipolar orthogonal vectors of n dimensions can be stored in an auto-associative net.

Example: Storing 2 vectors in an auto-associative net

- More than one vector can be stored in an auto-associative net by simply adding the weights needed for each vector.
- Assume we have two vectors that we want to store in an auto-associative net.
- Assume we want to store (1 1 -1 -1) and (-1 1 1 -1) in an auto-associative net.
- We obtain the weight matrices for each input and add them up.

Example: Storing 2 vectors in an auto-associative net

- Assume to store (1 1 -1 -1), we need a weight matrix $\mathbf{W}_1$ and to store (-1 1 1 -1), we need a weight matrix $\mathbf{W}_2$.
- $\mathbf{W}_1 =$ 0 1 -1 1 / 1 0 -1 1 / -1 -1 0 1 / -1 -1 1 0 $\quad \mathbf{W}_2 =$ 0 -1 -1 1 / -1 0 1 -1 / -1 1 0 -1 / 1 -1 -1 0
- To store (1 1 -1 -1) and (-1 1 1 -1), we simply add $\mathbf{W}_1$ and $\mathbf{W}_2$ to obtain the new weight matrix W. This is because the two vectors are orthogonal.
- $\mathbf{W} = \mathbf{W}_1 + \mathbf{W}_2 =$ 0 0 -2 0 / 0 0 0 -2 / -2 0 0 0 / 0 -2 0 0

Example: Storing 2 non-orthogonal vectors in an auto-associative 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:

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

Example: Storing 3 mutually orthogonal vectors in an auto-associative net

Let **W1 + W2** be the weight matrix to store the orthogonal vectors (1, 1, −1, −1) and (−1, 1, 1, −1) and **W3** be the weight matrix that stores (−1, 1, −1, 1). Then the weight matrix to store all three orthogonal vectors is **W1 + W2 + W3**. 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: Storing 4 mutually orthogonal vectors in an auto-associative net

Attempting to store a fourth vector, (1, 1, 1, 1), with weight matrix **W4**, 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

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

which cannot recognize any vector.

Storing several vectors in an auto-associative net

- Non-orthogonal vectors cannot be stored in an auto-associative net in a reliable manner so that they will be always identified.
- Two vectors x and y are orthogonal if $x\ y^T = 0$ i.e., $\Sigma\ x_i\ y_i = 0$

- **Theorem**: *n-1 mutually orthogonal bipolar vectors, each with n components can be stored in an auto-associative net using the sum of weight matrices, with diagonal terms set to zero*.
- **Theorem**: *Attempting to store n mutually orthogonal vectors in an auto-associative net using the sum of weight matrices will result in a weight matrix that cannot reproduce any of the stored vectors.*

Iterative Autoassociative Net

- In some cases, *an auto-associative net does not reproduce a stored pattern the first time around, but if the result of the first showing is input to the net again, the stored pattern is reproduced*.
- It is the case usually if there is error or missing components in the input.
- Example: Assume we create an auto-associative net for storing (1 1 1 -1). The weight matrix is

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

If we now show this network the input (1 0 0 0) where there are three missing components (zeros), the first showing results in output of (0 1 1 -1), showing this as input a second time produces (1 1 1 -1) as the output the second time around.

Iterative Auto-associative Net

- *Researchers observed this iterative feedback step and took it a little further by simply let the input and output units be the same, or by connecting the input and corresponding output nodes*.
- In such a case, we get a recurrent auto-associative network.
- We look at three different kinds of iterative auto-associative nets. These are the names the authors of the papers used.
  - Recurrent linear auto-associator
  - Brain-State-in-a-Box net
  - Discrete Hopfield net

Recurrent Linear Auto-associator

- This is a kind of iterative associator proposed by (McClelland and Rumelhart 1988) who use a weight matrix where the connection strength w_{ij} is the same as w_{ji}. I.e., the weight matrix is symmetric.
- This net has *n* neurons, each connected to all of the other neurons.
- The weight matrix **W** can be obtained by Hebb learning.

Recurrent Linear Auto-associator

- (Anderson et al. 1977) require that the weight matrix must always be non-singular for this net to work. A square matrix is singular if it has an inverse. Also, a square matrix is singular if its determinant is non-0.
- A non-singular *n x n* square matrix has *n* mutually orthogonal eigenvectors.
- A recurrent linear auto-associator neural net is trained using a set of *K* orthogonal vectors, where each vector may be shown different number of times.
- (Anderson et al. 1977) show that each of the stored vectors are an eigenvector of the weight matrix **W**.
- A recurrent linear auto-associator is intended to produce as its response the stored vector (eigenvector) to which the input vector is most similar.

Eigenvectors and Eigenvalues

- An **eigenvector** of a square matrix ***A*** is a non-zero vector v that, when the matrix multiplies v, yields a constant multiple of v, the latter multipler being commonly denoted by λ.
- That is,

$$\boldsymbol{A}v = \lambda v$$

- The number λ is called the **eigenvalue** of ***A*** corresponding to v.

Eigenvectors and Eigenvalues

For the transformation matrix

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix},$$

the vector

$$v = \begin{bmatrix} 4 \\ -4 \end{bmatrix}$$

is an eigenvector with eigenvalue 2. Indeed,

$$Av = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ -4 \end{bmatrix} = \begin{bmatrix} 3 \cdot 4 + 1 \cdot (-4) \\ 1 \cdot 4 + 3 \cdot (-4) \end{bmatrix} = \begin{bmatrix} 8 \\ -8 \end{bmatrix} = 2 \cdot \begin{bmatrix} 4 \\ -4 \end{bmatrix}.$$

On the other hand the vector

$$v = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

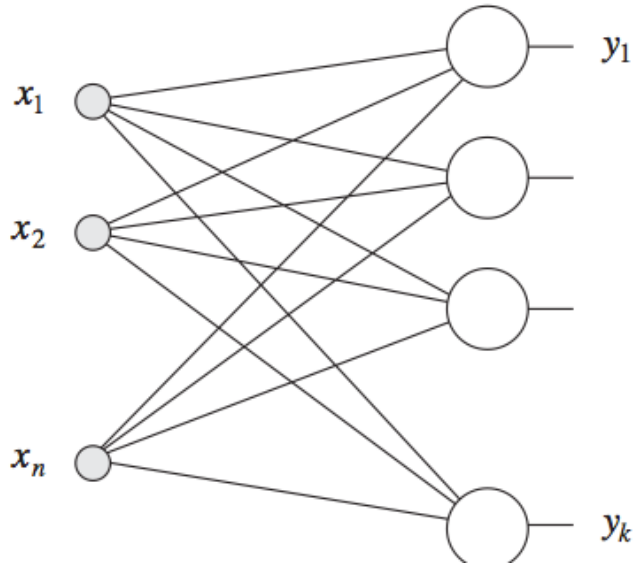
is *not* an eigenvector, since

$$\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \cdot 0 + 1 \cdot 1 \\ 1 \cdot 0 + 3 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix},$$

and this vector is not a multiple of the original vector v.

Math behind (recurrent) association networks

- As we know, there are only two layers in an association network we have seen so far, input and output.
- Consider the network structure given below again.



Math behind recurrent association networks

- Let w_{ij} be the weight between input node *i* and output unit *j*.
- Let **W** be the *n x k* weight matrix [w_{ij}].
- The vector **x** = (x_1, x_2, …x_n) produces the input excitation vector **e** = (e_1, …e_n) for each of the output through the computation

$$\mathbf{e} = \mathbf{x}\,\mathbf{W}$$

Math behind recurrent association networks

- In general, *m* different *n*-dimensional row vectors $\mathbf{x}^1$, $\mathbf{x}^2$, … $\mathbf{x}^m$ have to be associated with *m* *k*-dimensional row vectors $\mathbf{y}^1$, $\mathbf{y}^2$…$\mathbf{y}^m$.
- Let **X** be the *m x n* matrix who rows are each one of the input vectors.
- Let **Y** be the *m x k* matrix whose rows are the output vectors.
- The weights of an association network are given by a weight matrix **W** for which

  $$\mathbf{XW} = \mathbf{Y}$$

- In the auto-associative case, we can associate each vector with itself and this equation becomes

  $$\mathbf{XW} = \mathbf{X}$$

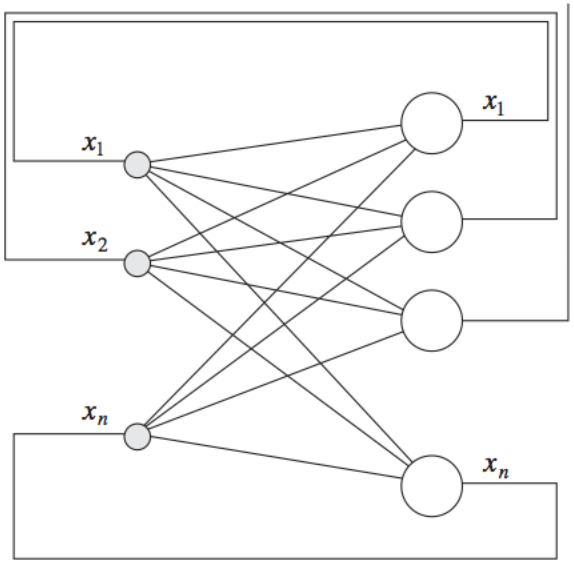
Math behind recurrent association networks

- If $m = n$, then $\mathbf{X}$ is a square matrix.
- If X is invertible, the solution for the general case can be written as

$$\mathbf{W} = \mathbf{X}^{-1}\,\mathbf{Y}$$

which means that finding $\mathbf{W}$ is solving a linear system of equations.

- *Interesting question*: What happens now if the output of the network is used as new input to itself?
- We see such a *recurrent auto-associative network* on the side.

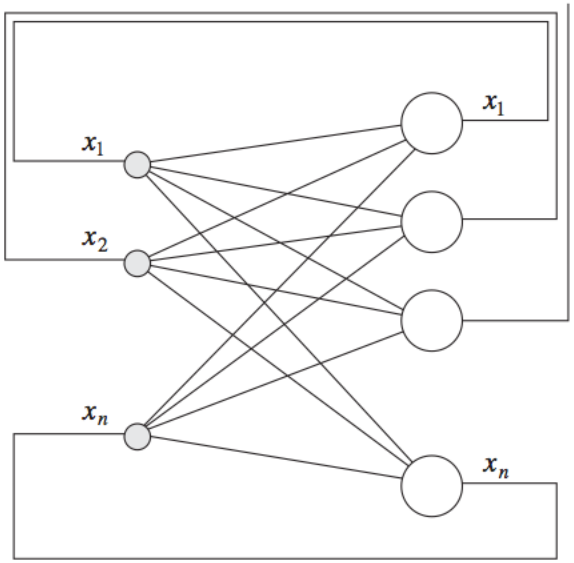


Math behind recurrent association networks

- We assume that all units compute their outputs simultaneously and such networks are called *asynchronous*.
- In each step, the network is fed an input vector **x**(*i*) and produces an output vector **x**(*i+1*).
- The question is for such networks, is there a fixed point such that

$$\xi \mathbf{W} = \xi$$

The vector is an eigenvector of **W** with eigenvalue 1.



Math behind recurrent association networks

- An *n x n* matrix **W** has at most *n* eigenvectors and *n* eigenvalues.
- The eigenvectors $\mathbf{v}^1$, $\mathbf{v}^2$, ..$\mathbf{v}^n$ satisfy the following

  $$\mathbf{v}^i \mathbf{W} = \lambda_i \mathbf{v}^i \quad \text{for } i=1\ldots n$$

- Assume without loss of generality that λ^1 is the eigenvalue of W with the largest magnitude and λ^2 is the second largest and so on.
- Also let $\lambda^1 > 0$.
- In general eigenvectors can be positive or negative, and there are theorems that describe the nature of the largest eigenvalue (Perron-Frobenius theorem) whether it's real or not, etc.

Math behind recurrent association networks

- Now, take any vector $\mathbf{a}^0$ or length *n*.
- Any vector can be written as a linear combination of the *n* eigenvectors of **W**:

$$\mathbf{a}_0 = \alpha_1 \mathbf{x}^1 + \alpha_2 \mathbf{x}^2 + \cdots + \alpha_n \mathbf{x}^n.$$

Assume that all constants α are non-zero. After the first iteration with the weight matrix $\mathbf{W}$ we get

$$\begin{aligned} \mathbf{a}_1 &= \mathbf{a}_0 \mathbf{W} \\ &= (\alpha_1 \mathbf{x}^1 + \alpha_2 \mathbf{x}^2 + \cdots + \alpha_n \mathbf{x}^n)\mathbf{W} \\ &= \alpha_1 \lambda_1 \mathbf{x}^1 + \alpha_2 \lambda_2 \mathbf{x}^2 + \cdots + \alpha_n \lambda_n \mathbf{x}^n. \end{aligned}$$

After t iterations the result is

$$\mathbf{a}_t = \alpha_1 \lambda_1^t \mathbf{x}^1 + \alpha_2 \lambda_2^t \mathbf{x}^2 + \cdots + \alpha_n \lambda_n^t \mathbf{x}^n.$$

- We can conclude that the eigenvalue λ_1, the one with the largest magnitute dominates the expression for a large value of t.

Recurrent Linear Auto-associator

- Any input pattern can be expressed as a linear combination of eigenvectors.
- The response of the net when an input is presented can be expressed as the corresponding linear combination of eigenvectors.
- The eigenvector to which the input is most similar is the eigenvector with the largest component in the linear combination.
- As the net is allowed to iterate, contributions to the response of the net from the eigenvectors with large eigenvalues grow relative to contributions of other eigenvectors with smaller eigenvalues.
- The units in a recurrent linear auto-associator update their activations simultaneously.
- The output of a linear associator can grow without bound.

Brain State in a Box (BSB) Net

- Another Iterative Associator called BSB, proposed by (Anderson et al. 1977)
- The output of a linear auto-associator can grow indefinitely.
- The output can be prevented from growing indefinitely by modifying the activation function to produce values that are restricted to between -1 and 1.
- In a BSB, like all other similar nets, there are *n* units connected to every other unit.
- There is also a self-connection weight of 1 like the recurrent linear associator before. .
- *BSB also uses a threshold for the activation function.*

Brain State in a Box (BSB) Net

Initialize all weights to small random values

Initialize learning rates α and β

for each training input vector do

Set initial activations of net equal to external input $y_i = x_i$

while activations continue to change do

Compute net input: $y_{in} = y_i + \alpha * \Sigma_j (y_i w_{ji})$

//Each net input is a combination of the unit's previous activation

//and the weighted signal receive from all units

Each unit determines its activation (output signal)

//Here a threshold of 1 is used; but a different value can be used.

$y_i = 1$ *if* $y_{ini} > 1$; y_i *if* $-1 <= y_{ini} <= 1$; -1 *if* $y_{ini} < -1$

Update weights w_{ij} (new) = w_{ij} (old) + $\beta * y_i y_j$ //Hebb Rule

endwhile

endfor

From Anderson et al. (1977); Other variations are possible.

Recognizing all vectors with 3 “missing components” with an recurrent auto-associator

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

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

This is an example from Fausett’s boo, pp. 133. The diagonal elements are 0; so the matrix W is singular and non-invertible. This seems to conflict with some of the discussions requiring the matrix to be non-singular and invertible.

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.

Recognizing all vectors with 3 “missing components” with an recurrent auto-associator

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).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).W = (2,3,2,–2) → (1,1,1,–1).
Step 5: (1,1,I,–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).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).W = (2,2,3,–2) → (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).W = (1,1, 1,0)
Step 5: Iterate.
Step 4: (1,1,1,0).W = (2, 2, 2,–3) → (1, 1, 1,, –1).
Step 5: (1,1,1,–1) is the stored vector, so we stop.

Discrete Hopfield Net

- An issue that needs to be considered in designing recurrent neural networks is how the units or computing elements are synchronized.
- For example, in a McCulloch-Pitts network assumes that the activation of each unit takes a unit of time. The network is built taking this delay into account. When the network becomes too contrived, explicit delay elements can be inserted.
- In the associative networks seen so far, we have assumed that synchronization takes place because all computing units evaluate their inputs and compute their outputs instantaneously.
- In such situations, the operation of the associative memory can be described in terms of simple linear algebra using vector-matrix multiplication and evaluating the sign of the output, assuming 0 as unit threshold.

Discrete Hopfield Net

- Many ANN researchers do not like the idea of a global synchronizing clock at which all nodes perform computation and produce output. In other words, all nodes work in lock step.
- It is more plausible to think that a computing input computes when it gets its inputs not at specific synchronized points of time.
- Networks in which computing units are activated at different times and which possibly require variable amounts of time to compute are called *stochastic automata.*
- Hopfield Networks (1982, 1984) are a kind of stochasitc automata.

Discrete Hopfield Net

- In a Hopfield net, the units compute their outputs at random points of time.
- At a point in time, only one random computing unit performs its computation based on its inputs.
- However, over a long period of time, Hopfield net requirs that the computing units fire or more less equally on average.
- Fully connected net: each unit is connected to every other node.
- Hopfield net also collapses each input-output node pair into just one node.
- Weights are symmetric
    - $w_{ij} = w_{ji}$
    - $w_{ii} = 0$

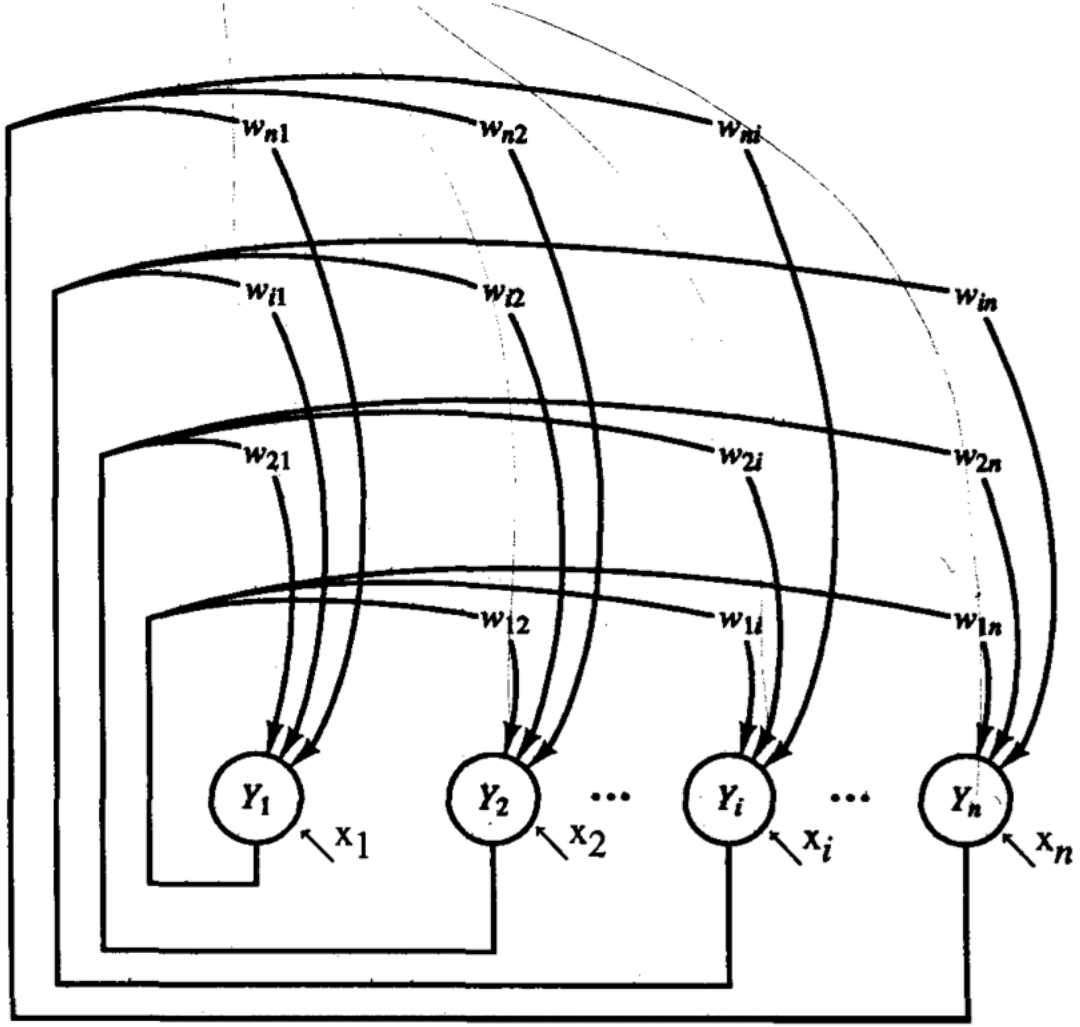
Characteristics of Hopfield Networks

- *Only 1 unit updates its activation at a time based on the signals it receives from other nodes.*
- *Each unit continues to receive an external signal in addition to the signal from other units in the net. (There are variations to this idea, e.g., initially Hopfield said that the input is shown in the first time cycle only)*
- Based on these two assumptions, Hopfield proved that the net will converge.
- The state of the network can be described in terms of an energy (called Lyapunov) function whose value becomes stable as the net converge.
- When the system converges, the weights and activations do not change any more.

Applications

- Hopfield studied these nets as content addressable memory.
- They can also be used to solve constrained optimization problems.
- We will discuss how Hopfield net can be used to solve problems like the Traveling Salesman Problem later in the semester.

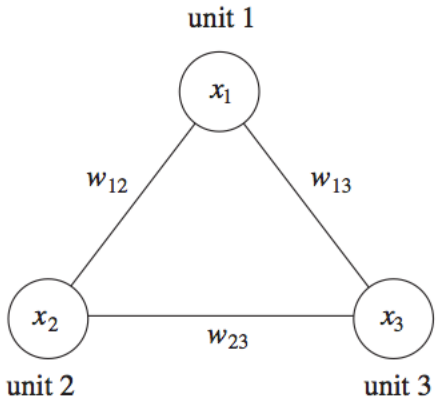
Architecture



x_i is an input
w_{ij} is the weight of the edge from node i to node j, $l \neq j$

Example of Hopfield Nets

- This is a Hopfield Net with 3 units.
- Each unit can be in state 1 or -1.
- Connections in a Hopfield net with *n* units can be represented using an *n x n* matrix **W** = $\{w_{ij}\}$ with 0 diagonal elements.



Energy of Hopfield Nets

Definition 17. *Let* $\mathbf{W}$ *denote the weight matrix of a Hopfield network of* n *units and let* θ *be the* n*-dimensional row vector of units' thresholds. The energy* $E(\mathbf{x})$ *of a state* $\mathbf{x}$ *of the network is given by*

$$E(\mathbf{x}) = -\frac{1}{2}\mathbf{x}\mathbf{W}\mathbf{x}^{\mathrm{T}} + \theta\mathbf{x}^{\mathrm{T}}.$$

The energy function can also be written in the form

$$E(\mathbf{x}) = -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ij}x_i x_j + \sum_{i=1}^{n}\theta_i x_i.$$

The factor $1/2$ is used because the identical terms $w_{ij}x_i x_j$ and $w_{ji}x_j x_i$ are present in the double sum.

The idea of energy of a network is introduced to discuss convergence of a net. It has been shown that when the energy doesn't change from one iteration to another, the network has stabilized and learned to auto-associate. Instead of looking at all the weights and determining if they are changing, it gives us one number to look at.

Energy of Hopfield Nets

- The energy of a Hopfield network is a quadratic function.
- A Hopfield network always finds a ***local*** minimum of the energy function (It doesn't have to be global).
- Hopfield proved that the network will converge if
  - The weight matrix is symmetric, and
  - The diagonal weights are 0, and
  - Weights are updated one at a time, in a random manner
- These conditions are necessary and sufficient.

Algorithm

- There are several versions of the discrete Hopfield net.
- Hopfield's first description (1982) used *binary* inputs.
- *In a Hopfield net, the weights don't change. They are precomputed based on theoretical analysis.*
- To store a set of binary patterns **s**(p), p=1, …., P

  s(p) = (s_1(p), s_2(p),….s_i(p)….s_n(p))

  the weight matrix **W** = {w_{ij}} is given by

  $w_{ij} = \Sigma_p$ [2s_i(p) – 1] [2s_j(p) – 1] for i ≠ j
- To store a set of bipolar patterns **s**(p), p=1, …., P

  s(p) = (s_1(p), s_2(p),….s_i(p)….s_n(p))

  the weight matrix **W** = {w_{ij}} is given by

  $w_{ij} = \Sigma_p$ s_i(p) s_j(p) for i ≠ j
- .

Algorithm for Hopfield Net

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 x, do Steps **2–6.**

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

There are many variations.
Originally, Hopfield didn't allow external input after the first time step.
Later, external input was allowed to continue during processing.
Binary or biploar activations can be used.
Convergence test checks if energy is stable.

The weights don't change. The threshold θ_i can be zero. The order of unit updates is random, but each unit must be updated at the same average rate.

Application of Hopfield Net

- A binary Hopfield net can be used to determine if an input vector is a “known” vector, i.e., one that was stored in the net.
- The net recognizes the “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 a “unknown” vector, the activation vectors produced as the net iterates will converge to a an activation vector that is not one of the stored patterns; such a pattern is called a *spurious activation state*.

Example of Using Hopfield Net

- Testing a discrete Hopfield net: Mistakes in the first and second components of a stored vector.
- Consider storing the vector (1 1 1 0) in a Hopfield net.
- Now assume there is an input with which we test it. The input has two mistakes in the first and second components (0 0 1 0).
- In this case, we have *binary* input and *bipolar* weights.
- The units are updated in the order Y_1, Y_4, Y_3, Y_2.

Example of Using Hopfield Net

- The weight matrix is calculated using the formula given.
- Input = (0 0 1 0)
- Y_1 updates its activation: the activation becomes (1 0 1 0)
- Y_4 updates its activation: the activation remains (1 0 1 0)
- Y_3 updates its activation: the activation remains (1 0 1 0)
- Y_2 updates its activation: the activation becomes (1 1 1 0), the one that was stored.
- Since the activation has changed during this update cycle, we need one more pass. In the next pass, activation doesn't change and remains constant at (1110)

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. $y = (0, 0, 1, 0)$.

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. 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.

Applications of Hopfield net

The purpose of a Hopfield net is to store 1 or more patterns and to recall the full patterns based on partial input. For example, consider the problem of optical character recognition. The task is to scan an input text and extract the characters out and put them in a text file in ASCII form. Okay, so what happens if you spilled coffee on the text that you want to scan? Now some of the characters are not quite as well defined, though they're mostly closer to the original characters than any other character:

Original 'T'

half of image corrupted by noise

So here's the way a Hopfield network would work. You map it out so that each pixel is one node in the network. You train it (or just assign the weights) to recognize each of the 26 characters of the alphabet, in both upper and lower case (that's 52 patterns). Now if your scan gives you a pattern like something on the right of the above illustration, you input it to the Hopfield network, and it chugs away for a few iterations, and eventually reproduces the pattern on the left, a perfect "T".

http://www.cs.ucla.edu/~rosen/161/notes/hopfield.html

Applications of Hopfield net

Economic load dispatch for piecewise quadratic cost function using **Hopfield** neural **network**
JH Park, YS Kim, IK Eom, KY Lee - Power Systems, IEEE …, 1993 - ieeexplore.ieee.org
Abstract-This paper presents a new method to solve the problem of economic power dispatch with piecewise quadratic cost function using the **Hopfield** neural **network**. Traditionally one convex cost function for each generator is assumed. However, it is more **...**
Cited by 307 Related articles All 6 versions Web of Science: 170 Import into BibTeX Save More

Hopfield network for stereo vision correspondence
NM Nasrabadi, CY Choo - Neural Networks, IEEE Transactions …, 1992 - ieeexplore.ieee.org
Abstract-An optimization approach is used to solve the corre-spondence problem for a set of features extracted from a pair of stereo images. A cost function is defined to represent the constraints on the solution, which is then mapped onto a twodimensional **Hopfield** neural **...**
Cited by 177 Related articles All 7 versions Web of Science: 74 Import into BibTeX Save More

Image restoration using a modified **Hopfield network**
JK Paik, AK Katsaggelos - Image Processing, IEEE …, 1992 - ieeexplore.ieee.org
Abstract-In this paper a modified **Hopfield** neural **network** model for regularized image restoration is presented. The proposed **network** allows negative autoconnections for each neuron. A set of algorithms using the proposed neural **network** model is presented, with **...**
Cited by 215 Related articles All 10 versions Web of Science: 107 Import into BibTeX Save More

Computerized tumor boundary detection using a **Hopfield** neural **network**
Y Zhu, H Yan - Medical Imaging, IEEE Transactions on, 1997 - ieeexplore.ieee.org
Abstract—In this paper, we present a new approach for detection of brain tumor boundaries in medical images using a **Hopfield** neural **network**. The boundary detection problem is formulated as an optimization process that seeks the boundary points to minimize an **...**
Cited by 158 Related articles All 5 versions Web of Science: 68 Import into BibTeX Save More

The application of competitive **Hopfield** neural **network** to medical image segmentation
KS Cheng, JS Lin, CW Mao - Medical Imaging, IEEE …, 1996 - ieeexplore.ieee.org
Abstract-In this paper, a parallel and unsupervised approach using the competitive **Hopfield** neural **network** (CHNN) is proposed for medical image segmentation. It is a kind of **Hopfield network** which incorporates the winner-takes-all (WTA) learning mechanism. The image **...**
Cited by 168 Related articles All 7 versions Web of Science: 76 Import into BibTeX Save More

Super-resolution target identification from remotely sensed images using a **Hopfield** neural **network**
AJ Tatem, HG Lewis, PM Atkinson… - … and Remote Sensing, …, 2001 - ieeexplore.ieee.org
Abstract—Fuzzy classification techniques have been developed recently to estimate the class composition of image pixels, but their output provides no indication of how these classes are distributed spatially within the instantaneous field of view represented by the **...**
Cited by 189 Related articles All 12 versions Web of Science: 114 Import into BibTeX Save More

Polygonal approximation using a competitive **Hopfield** neural **network**
PC Chung, CT Tsai, E Chen, YN Sun - Pattern Recognition, 1994 - Elsevier
Polygonal approximation plays an important role in pattern recognition and computer vision. In this paper, a parallel method using a Competitive **Hopfield** Neural **Network** (CHNN) is proposed for polygonal approximation. Based on the CHNN, the polygonal approximation **...**
Cited by 110 Related articles All 3 versions Web of Science: 61 Import into BibTeX Save More

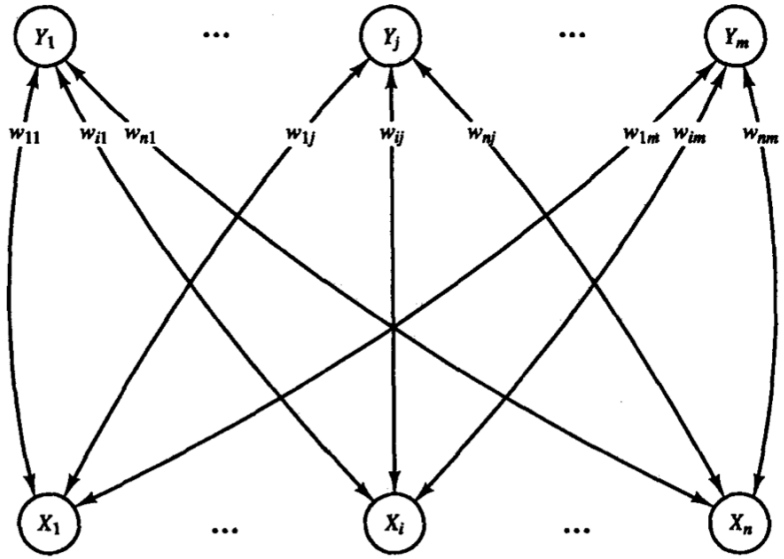
Super-resolution land cover pattern prediction using a **Hopfield** neural **network**
AJ Tatem, HG Lewis, PM Atkinson, MS Nixon - Remote Sensing of …, 2002 - Elsevier
Landscape pattern represents a key variable in management and understanding of the environment, as well as driving many environmental models. Remote sensing can be used to provide information on the spatial pattern of land cover features, but analysis and **...**
Cited by 164 Related articles All 11 versions Web of Science: 103 Import into BibTeX Save More

Heteroassociative Recurrent Neural Net: Bidirectional Associative Memory

- Hopfield net is an autoassociative recurrent neural net.
- Recurrent neural nets can be used as heteroassociators also.
- Bidirectional Asssociative Memory (BAM) is an example (Kosko 1988, 1992).
- The network consists of two layers of neurons, connected by bidirectional weighted paths.
- The net iterates sending signals back and forth between the two layers until each neuron's activation remains constant for several steps.
- Bidirectional associative memory nets can respond to input to either layer.
- A BAM net alternates between updating activations for each layer.
- Fausett calls the two layers X-layer and Y-layer instead of input and output layers.

Bidirectional Associative Memory

- A BAM has *n* units in the X-layer and m units in the Y-layer.
- The connections between the layers are bidirectional, i.e., if the weight matrix for signals sent from X-layer to Y-layer is **W**, the weight matrix for signals sent from Y-layer to S-layer is $\mathbf{W}^T$.
- Fausett discusses three types of BAMS: binary, bipolar and continuous. Binary and bipolar are similar, although bipolar works better for many problems



Bidirectional Associative Memory

- Like most nets, we have a choice of activation function, use of 0 or other thresholds.
- BAMs as discussed in Fausett use step function, with the possibility of non-0 threshold.
- Hebb Rule or Delta Rule can be used to train a BAM.
- For a BAM, the weights are computed using a pre-defined formula and do not change.
- Compute weights for each pair of input $s_i(p)$ and $t_j(p)$, and add all the individual weight matrices up. For example,

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

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

Activation Function

We can use any appropriate activation function and learning rule, though Hebb Rule and Delta Rule are commonly used, with step functions or smoother functions.

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

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

Algorithm

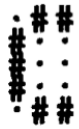
1. initialize weights
2. for each test vector do
3. present **s** to x layer
4. present **t** to y layer
5. while equilibrium is not reached
6. compute $f(y_{in.j})$
7. compute $f(x_{in.j})$

BAM: Example Application 1

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 x 3 patterns) to the following bipolar codes:

(-1, 1)



(1,1)

The weight matrices are:

(to store A → −11)

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix}$$

(C → 11)

$$\begin{bmatrix} -1 & -1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

(*W*, to store both)

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

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:

BAM: Example Application 1

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 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.

BAM: Example Application 2a

We are given a **y** vector as input, one that is noisy version of one of the training **y** vectors. We are not given an **x** vector.

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

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

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

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

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

BAM: Example Application 2a

We are given a **y** vector as input, one that is noisy version of one of the training **y** vectors. We are not given an **x** vector.

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.

BAM: Example Application 2b

We are given a **y** vector as input, one that is noisy version of one of the training **y** vectors. We are given an **x** vector which is a noisy version of the associated **y** vector, but it is clearly distinguishable from the other **y** vector. .

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,

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

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

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

where the nonzero components are those that distinguish A from

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

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

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

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

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

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

which is pattern A.

BAM: Example Application 2c

We are given a **y** vector as input, one that is noisy version of one of the training **y** vectors. We are given an **x** vector which is a noisy version of the associated **x** vector and this noisy version has similarity with the other **y** vector also. .

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 y = (0 1) and **x** = (0 0 −1 0 0 1 0 1 0 0 0 0 0 0 0 0). Then

$$(0,1)W^T =$$

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

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

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

which is not quite pattern A.

So we try iterating, sending the **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.

About BAMs

- The number of different bits in two binary or bipolar vectors is called the Hamming distance between the two.
- The average Hamming distance between two equal length vectors is the total Hamming distance divided by the number of bits in the vectors.
- Encoding is better when the average Hamming distance of the inputs is similar to the average Hamming distance of the outputs.

Continuous BAMs

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 $(s(p), t(p))$, $p = 1, 2, \ldots, P$, 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

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