# Competitive Networks

Instructor: Dr. Mohammad Rashedur Rahman

# Introduction

- When we applied a net that was trained to classify the input signal into one of the output categories, A, B, C, D, E, J, or K,
- The net sometimes responded that the signal was both a C and a K, or both an E and a K, or both a J and a K.
- In circumstances such as this, in which we know that only one of several neurons should respond, we can include additional structure in the network so that the net is forced to make a decision as to which one unit will respond. The mechanism by which this is achieved is called **competition**.
- This type of neural network is known as **competitive neural network**.

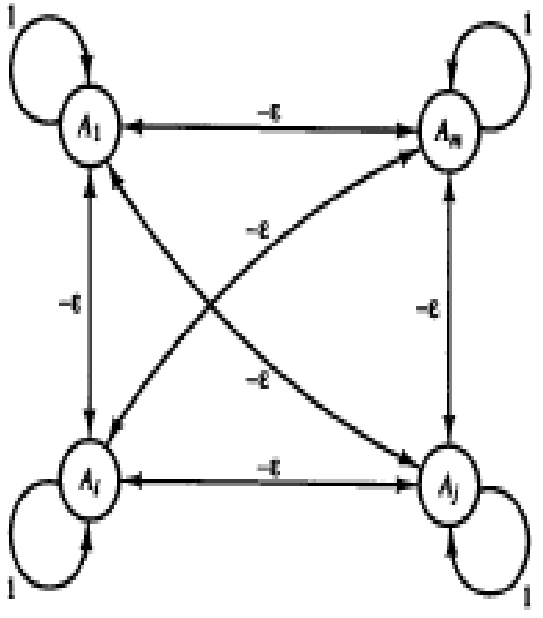


Figure 1.6 Competitive layer.

# Introduction (Cont..)

- Winner Take All. As the name suggests, only one neuron in the competing group will have a nonzero output signal when the competition is completed. A specific competitive net that performs Winner-Take-All competition is the MAXNET.
- MAXNET [Lippmann, 1987] is a specific example of a neural net based on competition. It can be used as a subnet to pick the node whose input is the largest.
- The m nodes in this subnet are completely interconnected, with symmetric weights. There is no training algorithm for the MAXNET; the weights are fixed.

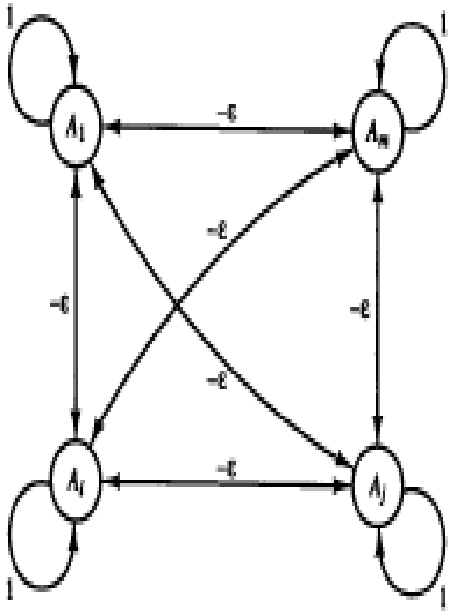


Figure 1.6 Competitive layer.

# Algorithm

The activation function for the MAXNET is

$$f(x) = \begin{cases} x & \text{if } x \geq 0; \\ 0 & \text{otherwise.} \end{cases}$$

The application procedure is as follows:

*Step 0.* Initialize activations and weights (set $0 < \epsilon < \frac{1}{m}$):

$a_j(0)$ input to node $A_j$,

$$w_{ij} = \begin{cases} 1 & \text{if } i = j; \\ -\epsilon & \text{if } i \neq j. \end{cases}$$

*Step 1.* While stopping condition is false, do Steps 2–4.

*Step 2.* Update the activation of each node: For $j = 1, \ldots, m$,

$$a_j(\text{new}) = f[a_j(\text{old}) - \epsilon \sum_{k \neq j} a_k(\text{old})].$$

*Step 3.* Save activations for use in next iteration:

$$a_j(\text{old}) = a_j(\text{new}), j = 1, \ldots, m.$$

*Step 4.* Test stopping condition:
If more than one node has a nonzero activation, continue;
otherwise, stop.

# MAXNET Example

The activation function for the MAXNET is

$$f(x) = \begin{cases} x & \text{if } x \geqslant 0; \\ 0 & \text{otherwise.} \end{cases}$$

The application procedure is as follows:

*Step 0.* Initialize activations and weights (set $0 < \epsilon < \frac{1}{m}$):

$a_j(0)$ input to node $A_j$,

$$w_{ij} = \begin{cases} 1 & \text{if } i = j; \\ -\epsilon & \text{if } i \neq j. \end{cases}$$

*Step 1.* While stopping condition is false, do Steps 2–4.

*Step 2.* Update the activation of each node: For $j = 1, \ldots, m$,

$$a_j(\text{new}) = f[a_j(\text{old}) - \epsilon \sum_{k \neq j} a_k(\text{old})].$$

*Step 3.* Save activations for use in next iteration:

$$a_j(\text{old}) = a_j(\text{new}), j = 1, \ldots, m.$$

*Step 4.* Test stopping condition:
If more than one node has a nonzero activation, continue; otherwise, stop.

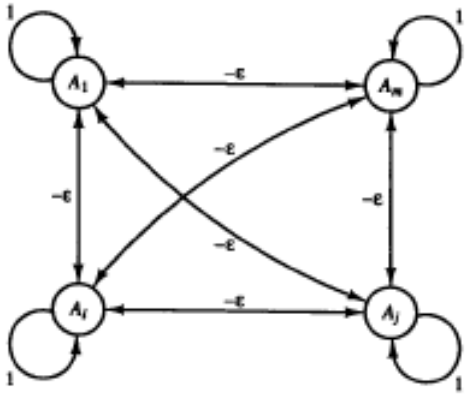



Figure 1.6 Competitive layer.

**Example 4.1 Using a MAXNET**

Consider the action of a MAXNET with four neurons and inhibitory weights $\epsilon = 0.2$ when given the initial activations (input signals)

$$a_1(0) = 0.2 \quad a_2(0) = 0.4 \quad a_3(0) = 0.6 \quad a_4(0) = 0.8$$

The activations found as the net iterates are

$$a_1(1) = 0.0 \quad a_2(1) = 0.08 \quad a_3(1) = 0.32 \quad a_4(1) = 0.56$$

$$a_1(2) = 0.0 \quad a_2(2) = 0.0 \quad a_3(2) = 0.192 \quad a_4(2) = 0.48$$

$$a_1(3) = 0.0 \quad a_2(3) = 0.0 \quad a_3(3) = 0.096 \quad a_4(3) = 0.442$$

$$a_1(4) = 0.0 \quad a_2(4) = 0.0 \quad a_3(4) = 0.008 \quad a_4(4) = 0.422$$

$$a_1(5) = 0.0 \quad a_2(5) = 0.0 \quad a_3(5) = 0.0 \quad a_4(5) = 0.421$$

# Hamming Network

- A Hamming net [Lippmann, 1987; DARPA, 1988] is a maximum likelihood classifier net that can be used to determine which of several exemplar vectors is most similar to an input vector (an ***n***-tuple).
- The exemplar vectors determine the weights of the net.
- The measure of similarity between the input vector and the stored exemplar vectors is *n* minus the Hamming distance between the vectors.

# Hamming Network (Cont..)

- The Hamming distance between two vectors is the number of components in which the vectors differ. For bipolar vectors x and y,

  $x-y = a - d,$

  where *a* is the number of components in which the vectors agree and *d* is the number of components in which the vectors differ, i.e., the Hamming distance.
- However, if *n* is the number of components in the vectors, then

  $d = n - a$

  and

  $x-y = 2a - n,$

  or

  $2a = x-y + n.$

# Hamming Network (Cont..)

- By setting the weights to be one-half the exemplar vector and setting the value of the bias to n/2,
- The net will find the unit with the closest exemplar simply by finding the unit with the largest net input.
- The Hamming net uses MAXNET as a subnet to find the unit with the largest net input.

# Hamming Network (Cont..)

- The lower net consists of n input nodes, each connected to m output nodes (where m is the number of exemplar vectors stored in the net).
- The output nodes of the lower net feed into an upper net (MAXNET) that calculates the best exemplar match to the input vector. The input and exemplar vectors are bipolar.
- The sample architecture shown in Figure 4.4 assumes input vectors are 4-tuples, to be categorized as belonging to one of two classes.

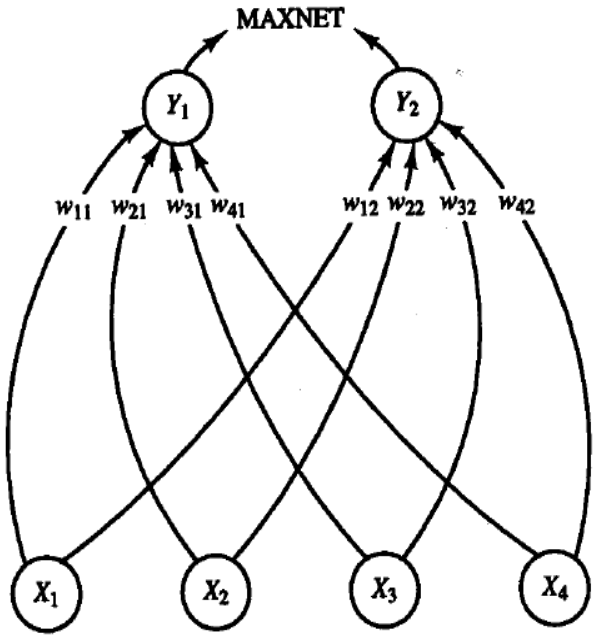



**Figure 4.4** Hamming net.

# Hamming Network (Algorithm)

**Application**

Given a set of $m$ bipolar exemplar vectors, $\mathbf{e}(1), \mathbf{e}(2), \ldots, \mathbf{e}(m)$, the Hamming net can be used to find the exemplar that is closest to the bipolar input vector $\mathbf{x}$. The net input $y\_in_j$ to unit $Y_j$, gives the number of components in which the input vector and the exemplar vector for unit $\mathbf{Y}_j$ $\mathbf{e}(j)$, agree ($n$ minus the Hamming distance between these vectors).

The nomenclature we use is as follows:

| | |
|---|---|
| $n$ | number of input nodes, number of components of any input vector; |
| $m$ | number of output nodes, number of exemplar vectors; |
| $\mathbf{e}(j)$ | the $j$th exemplar vector: |

$$\mathbf{e}(j) = (e_1(j), \ldots, e_i(j), \ldots, e_n(j)).$$

The application procedure for the Hamming net is:

*Step 0.* To store the $m$ exemplar vectors, initialize the weights:

$$w_{ij} = \frac{e_i(j)}{2}, (i = 1, \ldots, n; j = 1, \ldots, m).$$

And initialize the biases:

$$b_j = \frac{n}{2}, (j = 1, \ldots, m).$$

*Step 1.* For each vector $\mathbf{x}$, do Steps 2–4.

*Step 2.* Compute the net input to each unit $Y_j$:

$$y\_in_j = b_j + \sum_i x_i w_{ij}, (j = 1, \ldots, m).$$

*Step 3.* Initialize activations for MAXNET:

$$y_j(0) = y\_in_j, (j = 1, \ldots, m).$$

*Step 4.* MAXNET iterates to find the best match exemplar.

# Hamming Network Example

Given the exemplar vectors

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

and

$$\mathbf{e}(2) = (-1, -1, -1, 1),$$

the Hamming net can be used to find the exemplar that is closest to each of the bipolar input patterns, $(1, 1, -1, -1)$, $(1, -1, -1, -1)$, $(-1, -1, -1, 1)$, and $(-1, -1, 1, 1)$.

*Step 0.* Store the $m$ exemplar vectors in the weights:

$$\mathbf{W} = \begin{bmatrix} .5 & -.5 \\ -.5 & -.5 \\ -.5 & -.5 \\ -.5 & .5 \end{bmatrix}.$$

Initialize the biases:

$$b_1 = b_2 = 2.$$

*Step 1.* For the vector $\mathbf{x} = (1, 1, -1, -1)$, do Steps 2–4.

*Step 2.* $y\_in_1 = b_1 + \sum_i x_i w_{i1}$

$$= 2 + 1 = 3;$$

$$y\_in_2 = b_2 + \sum_i x_i w_{i2}$$

$$= 2 - 1 = 1.$$

These values represent the Hamming similarity because $(1, 1, -1, -1)$ agrees with $\mathbf{e}(1) = (1, -1, -1, -1)$ in the first, third, and fourth components and because $(1, 1, -1, -1)$ agrees with $\mathbf{e}(2) = (-1, -1, -1, 1)$ in only the third component.

*Step 3.* $y_1(0) = 3;$

$$y_2(0) = 1.$$

*Step 4.* Since $y_1(0) > y_2(0)$, MAXNET will find that unit $Y_1$ has the best match exemplar for input vector $\mathbf{x} = (1, 1, -1, -1)$.

For the vector $\mathbf{x} = (1, 1, -1, -1)$, do Steps 2–4.

# Hamming Network Example (Cont..)

*Step 1.* For the vector $\mathbf{x} = (1, -1, -1, -1)$, do Steps 2–4.

*Step 2.* $y\_in_1 = b_1 + \sum_i x_i w_{i1}$

$= 2 + 2 = 4;$

$y\_in_2 = b_2 + \sum_i x_i w_{i2}$

$= 2 + 0 = 2.$

Note that the input vector agrees with $\mathbf{e}(1)$ in all four components and agrees with $\mathbf{e}(2)$ in the second and third components.

*Step 3.* $y_1(0) = 4;$

$y_2(0) = 2.$

*Step 4.* Since $y_1(0) > y_2(0)$, MAXNET will find that unit $Y_1$ has the best match exemplar for input vector $\mathbf{x} = (1, -1, -1, -1)$.

*Step 1.* For the vector $\mathbf{x} = (-1, -1, -1, 1)$, do Steps 2–4.

*Step 2.* $y\_in_1 = b_1 + \sum_i x_i w_{i1}$

$= 2 + 0 = 2;$

$y\_in_2 = b_2 + \sum_i x_i w_{i2}$

$= 2 + 2 = 4.$

The input vector agrees with $\mathbf{e}(1)$ in the second and third components and agrees with $\mathbf{e}(2)$ in all four components.

*Step 3.* $y_1(0) = 2;$

$y_2(0) = 4.$

*Step 4.* Since $y_2(0) > y_1(0)$, MAXNET will find that unit $Y_2$ has the best match exemplar for input vector $\mathbf{x} = (-1, -1, -1, 1)$.

# Kohonen Self-Organizing Maps (SOM)

- The self-organizing neural networks described in this section, also called topology-preserving maps, assume a **topological structure** among the cluster units.
- This property is observed in the brain, but is not found in other artificial neural net-works.
- There are ***m*** cluster units, arranged in a one- or two-dimensional array; the input signals are ***n***-tuples [Kohonen, l989a].
- The weight vector for a cluster unit serves as an exemplar of the input patterns associated with that cluster.
- During the self-organization process, the cluster unit whose weight vector matches the input pattern most closely (typically, the square of the minimum Euclidean distance) is chosen as the winner.

# SOM (Cont...)

- The winning unit and its neighboring units (in terms of the topology of the cluster units) update their weights. The weight vectors of neighboring units are not in general, close to the input pattern.
- For example, for a linear array of cluster units, the neighborhood of radius R around ***cluster unit J*** consists of all units j such that

  **max(1, J - R) <=j <= min(J + R, m).**

  Here m is number of clusters

- The architecture of the Kohonen self-organizing map is shown in Figure 4.5. Neighborhoods of the unit designated by # of radii R = 2, l, and 0 in a one-dimensional topology (with 10 cluster units) are shown in Figure 4.6.

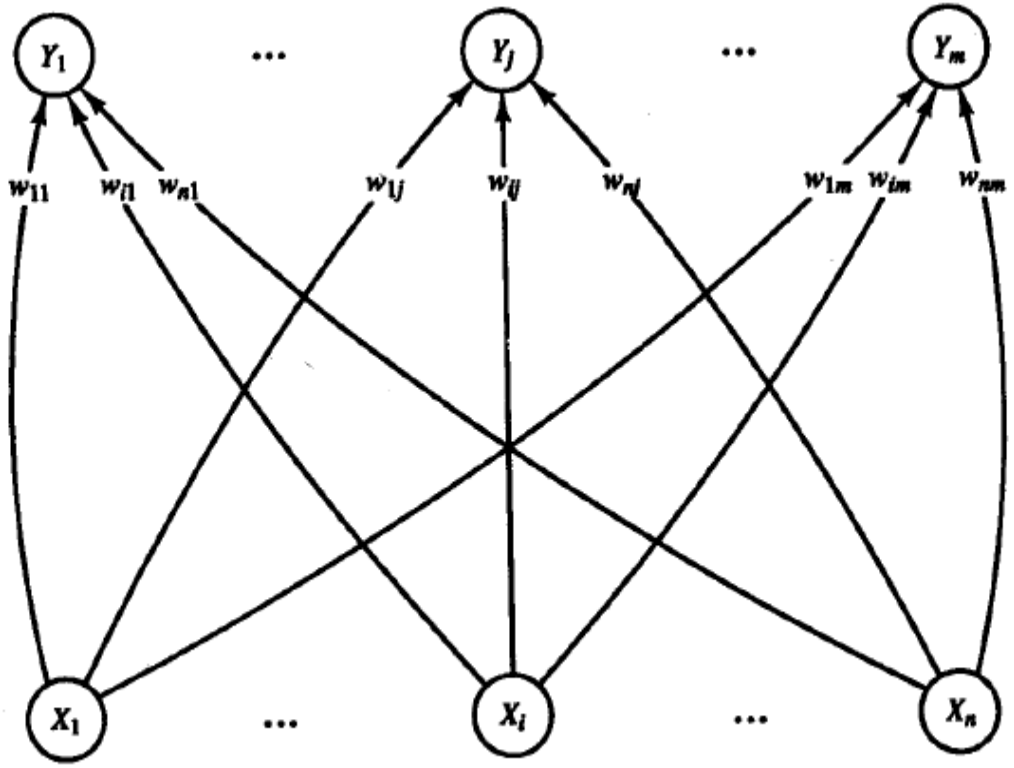



**Figure 4.5** Kohonen self-organizing map.

* * * { * ( * [#] * ) * } * *

{ } R = 2 ( ) R = 1 [ ] R = 0

**Figure 4.6** Linear array of cluster units.

# SOM (Cont…)

- The neighborhoods of radii R = 2, 1 and 0 are shown in Figure 4.7 for a rectangular grid and in Figure 4.8 for a hexagonal grid (each with 49 units).
- In each illustration, the winning unit is indicated by the symbol "#" and the other units are denoted by
- Note that each unit has eight nearest neighbors in the rectangular grid, but only six in the hexagonal grid.
- Winning units that are close to the edge of the grid will have some neighborhoods that have fewer units than that shown in the respective figure.
- Neighborhoods do not "wrap around" from one side of the grid to the other; "missing" units are simply ignored.

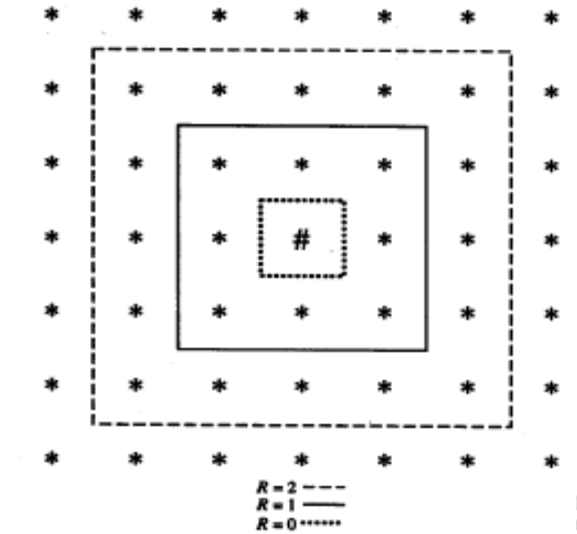



Figure 4.7 Neighborhoods for rectangular grid.

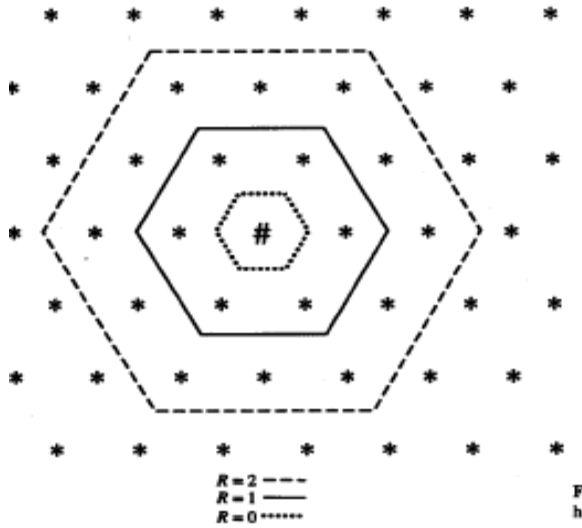
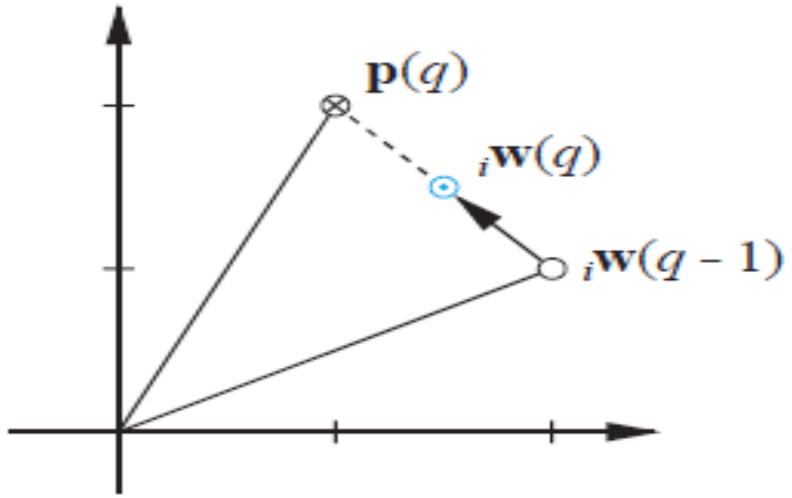



Figure 4.8 Neighborhoods for hexagonal grid.

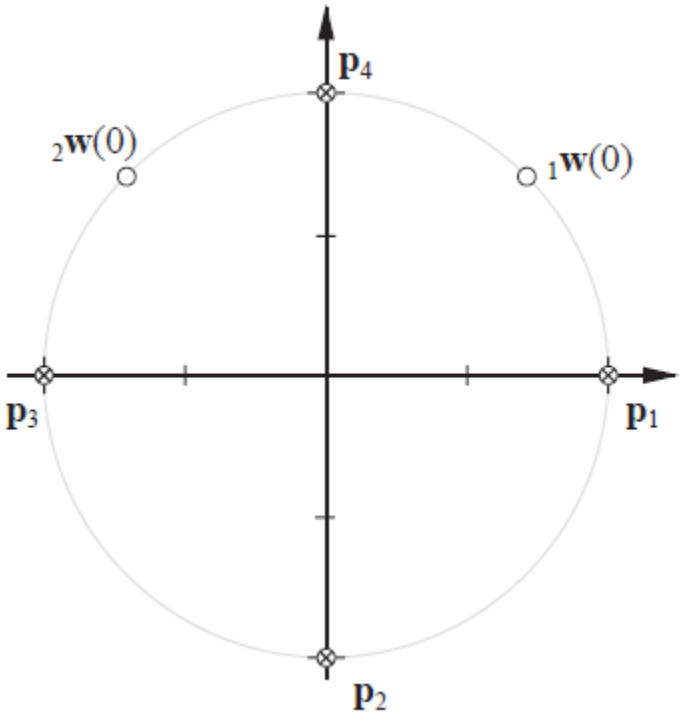
# Graphical Representation



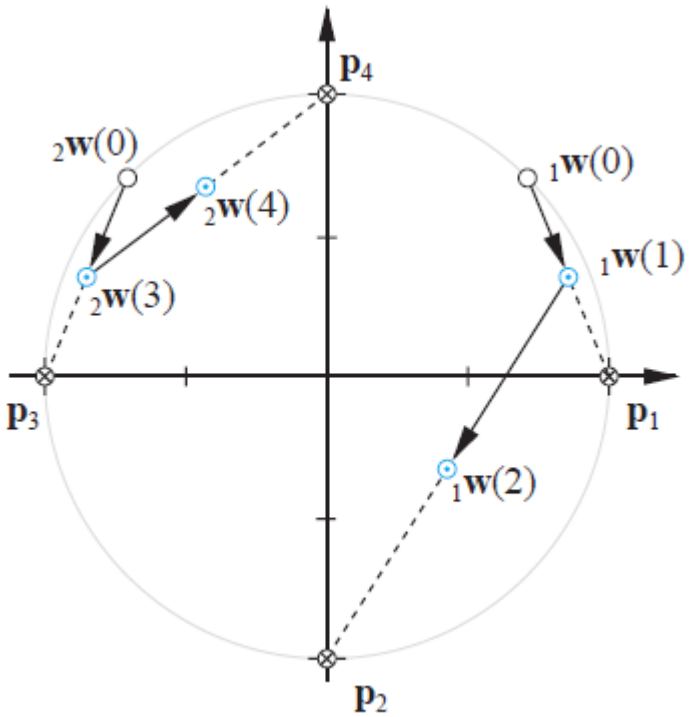

$$_{i^*}\mathbf{w}(q) = {}_{i^*}\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}(q-1))$$

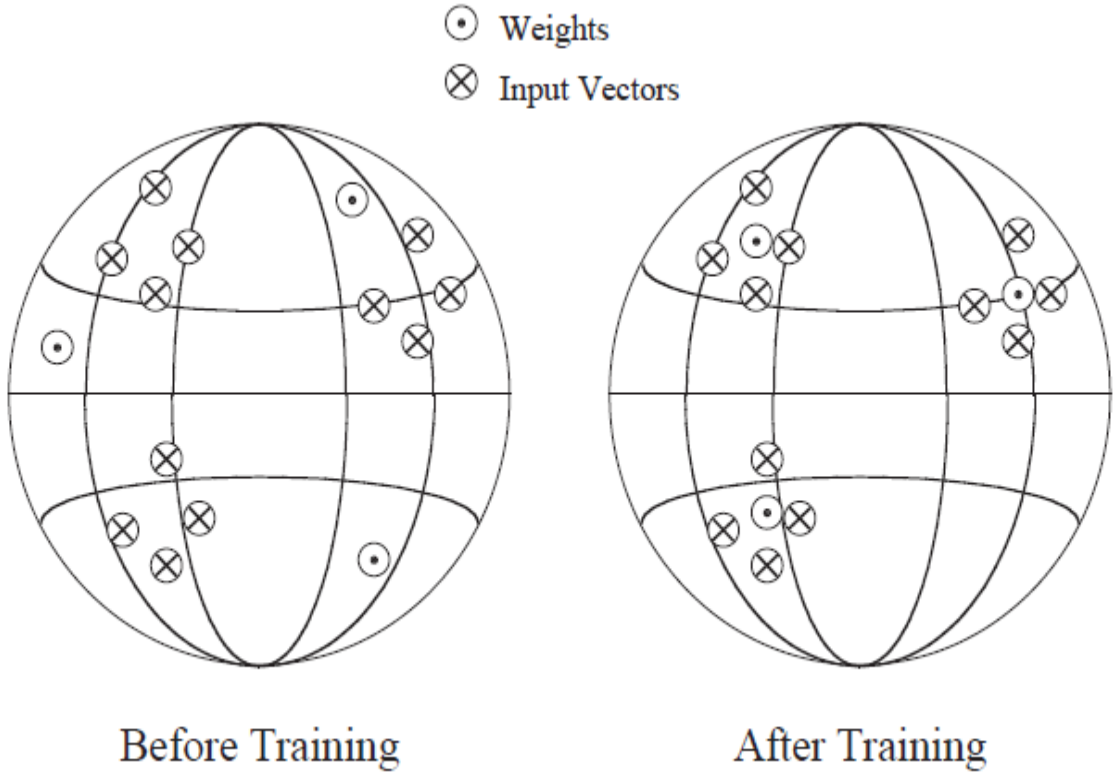
$$_{i^*}\mathbf{w}(q) = (1-\alpha){}_{i^*}\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

# Graphical Representation (Cont..)



# Graphical Representation (Cont..)



# Algorithm: SOM

*Step 0.* Initialize weights $w_{ij}$. (Possible choices are discussed below.)
Set topological neighborhood parameters.
Set learning rate parameters.

*Step 1.* While stopping condition is false, do Steps 2–8.

*Step 2.* For each input vector **x**, do Steps 3–5.

*Step 3.* For each $j$, compute:

$$D(j) = \sum_i (w_{ij} - x_i)^2.$$

*Step 4.* Find index $J$ such that $D(J)$ is a minimum.

*Step 5.* For all units $j$ within a specified neighborhood of $J$, and for all $i$:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})].$$

*Step 6.* Update learning rate.

*Step 7.* Reduce radius of topological neighborhood at specified times.

*Step 8.* Test stopping condition.

# Choice of Radius and Learning Rate in SOM

- The learning rate a is a slowly decreasing function of time (or training epochs). Kohonen (1989a, p. 133) indicates that a linearly decreasing function is satisfactory for practical computations; a geometric decrease would produce similar results.
- The radius of the neighborhood around a cluster unit also decreases as the clustering process progresses.
- The formation of a map occurs in two phases: the initial formation of the correct order and the final convergence.
- The second phase takes much longer than the first and requires a small value for the learning rate. Many iterations through the training set may be necessary, at least in some applications [Kohonen,1989a].
- Random values may be assigned for the initial weights.
- If some information is available concerning the distribution of clusters that might be appropriate for a particular problem, the initial weights can be taken to reflect that prior knowledge. the weights are initialized to random values (chosen from the same range of values as the components of the input vectors).

# Example on SOM

Let the vectors to be clustered be

$$(1, 1, 0, 0); (0, 0, 0, 1); (1, 0, 0, 0); (0, 0, 1, 1).$$

The maximum number of clusters to be formed is

$$m = 2.$$

Suppose the learning rate (geometric decrease) is

$$\alpha(0) = .6,$$

With only two clusters available, the neighborhood of node $J$ (Step 4) is s
only one cluster updates its weights at each step (i.e., $R = 0$).

*Step 0.* Initial weight matrix:

$$\begin{bmatrix} .2 & .8 \\ .6 & .4 \\ .5 & .7 \\ .9 & .3 \end{bmatrix}.$$

Initial radius:

$$R = 0.$$

Initial learning rate:

$$\alpha(0) = 0.6.$$

*Step 1.* Begin training.

*Step 2.* For the first vector, (1, 1, 0, 0), do Steps 3–5.

*Step 3.*
$$D(1) = (.2 - 1)^2 + (.6 - 1)^2 + (.5 - 0)^2 + (.9 - 0)^2 = 1.86;$$
$$D(2) = (.8 - 1)^2 + (.4 - 1)^2 + (.7 - 0)^2 + (.3 - 0)^2 = 0.98.$$

*Step 4.* The input vector is closest to output node 2, so

$$J = 2.$$

*Step 5.* The weights on the winning unit are updated:

$$w_{J2}(\text{new}) = w_{J2}(\text{old}) + .6\,[x_i - w_{J2}(\text{old})]$$
$$= .4\, w_{J2}(\text{old}) + .6\, x_i.$$

This gives the weight matrix

$$\begin{bmatrix} .2 & .92 \\ .6 & .76 \\ .5 & .28 \\ .9 & .12 \end{bmatrix}.$$

*Step 2.* For the second vector, (0, 0, 0, 1), do Steps 3–5.

*Step 3.*
$$D(1) = (.2 - 0)^2 + (.6 - 0)^2 + (.5 - 0)^2 + (.9 - 1)^2 = 0.66;$$
$$D(2) = (.92 - 0)^2 + (.76 - 0)^2 + (.28 - 0)^2 + (.12 - 1)^2 = 2.2768.$$

*Step 4.* The input vector is closest to output node 1, so

$$J = 1.$$

# Example on SOM (Cont..)

*Step 5.* Update the first column of the weight matrix:

$$\begin{bmatrix} .08 & .92 \\ .24 & .76 \\ .20 & .28 \\ .96 & .12 \end{bmatrix}.$$

*Step 2.* For the third vector, (1, 0, 0, 0), do Steps 3–5.

*Step 3.*

$$D(1) = (.08 - 1)^2 + (.24 - 0)^2 + (.2 - 0)^2 + (.96 - 0)^2 = 1.8656;$$

$$D(2) = (.92 - 1)^2 + (.76 - 0)^2 + (.28 - 0)^2 + (.12 - 0)^2 = 0.6768.$$

*Step 4.* The input vector is closest to output node 2, so

$$J = 2.$$

*Step 5.* Update the second column of the weight matrix:

$$\begin{bmatrix} .08 & .968 \\ .24 & .304 \\ .20 & .112 \\ .96 & .048 \end{bmatrix}.$$

*Step 2.* For the fourth vector, (0, 0, 1, 1), do Steps 3–5.

*Step 3.*

$$D(1) = (.08 - 0)^2 + (.24 - 0)^2 + (.2 - 1)^2 + (.96 - 1)^2 = 0.7056;$$

$$D(2) = (.968 - 0)^2 + (.304 - 0)^2 + (.112 - 1)^2 + (.048 - 1)^2 = 2.724.$$

*Step 4.*

$$J = 1.$$

*Step 5.* Update the first column of the weight matrix:

$$\begin{bmatrix} .032 & .968 \\ .096 & .304 \\ .680 & .112 \\ .984 & .048 \end{bmatrix}.$$

*Step 6.* Reduce the learning rate:

$$\alpha = .5\,(0.6) = .3$$

The weight update equations are now

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + .3\,[x_i - w_{ij}(\text{old})]$$

$$= .7w_{ij}(\text{old}) + .3x_i.$$

# Example on SOM (Cont..)

The weight matrix after the second epoch of training is

$$\begin{bmatrix} .016 & .980 \\ .047 & .360 \\ .630 & .055 \\ .999 & .024 \end{bmatrix}$$

Modifying the adjustment procedure for the learning rate so that it decreases geometrically from .6 to .01 over 100 iterations (epochs) gives the following results:

Iteration 0: Weight matrix: $\begin{bmatrix} .2 & .8 \\ .6 & .4 \\ .5 & .7 \\ .9 & .3 \end{bmatrix}$

Iteration 1: Weight matrix: $\begin{bmatrix} .032 & .970 \\ .096 & .300 \\ .680 & .110 \\ .980 & .048 \end{bmatrix}$

Iteration 2: Weight matrix: $\begin{bmatrix} .0053 & .9900 \\ -.1700 & .3000 \\ .7000 & .0200 \\ 1.0000 & .0086 \end{bmatrix}$

Iteration 10: Weight matrix: $\begin{bmatrix} 4.6e\text{-}7 & .3700 \\ .6300 & 5.4e\text{-}7 \\ 1.0000 & 2.3e\text{-}7 \end{bmatrix}$

Iteration 50: Weight matrix: $\begin{bmatrix} 1.9e\text{-}19 & 1.0000 \\ 5.7e\text{-}15 & .4700 \\ .5300 & 6.6e\text{-}15 \\ 1.0000 & 2.8e\text{-}15 \end{bmatrix}$

Iteration 100: Weight matrix: $\begin{bmatrix} 6.7e\text{-}17 & 1.0000 \\ 2.0e\text{-}16 & .4900 \\ .5100 & 2.3e\text{-}16 \\ 1.0000 & 1.0e\text{-}16 \end{bmatrix}$

These weight matrices appear to be converging to the matrix

$$\begin{bmatrix} 0.0 & 1.0 \\ 0.0 & 0.5 \\ 0.5 & 0.0 \\ 1.0 & 0.0 \end{bmatrix},$$

the first column of which is the average of the two vectors placed in cluster 1 and the second column of which is the average of the two vectors placed in cluster 2.

(1,1,0,0) and (1,0,0,0) will be grouped together and (0,0,0,1) and (0,0,1,1) will be grouped together

# LEARNING VECTOR QUANTIZATION (LVQ)

- Learning vector quantization (LVQ) [Kohonen l989a, l990a] is a pattern classification method in which each output unit represents a particular class or category. (Several output units should be used for each class.)
- The weight vector for an output unit is often referred to as a reference (or codebook) vector for the class that the unit represents. During training, the output units are positioned (by adjusting their weights through supervised training).
- It is assumed that a set of training patterns with known classifications is provided, along with an initial distribution of reference vectors (each of which represents a known classification).
- After training, an LVQ net classifies an input vector by assigning it to the same class as the output unit that has its weight vector (reference vector) closest to the input vector.

# LVQ Algorithm (Cont..)

The motivation for the algorithm for the LVQ net is to find the output unit that is closest to the input vector. Toward that end, if $\mathbf{x}$ and $\mathbf{w}_c$ belong to the same class, then we move the weights toward the new input vector; if $\mathbf{x}$ and $\mathbf{w}_c$ belong to different classes, then we move the weights away from this input vector.

The nomenclature we use is as follows:

| | |
|---|---|
| $\mathbf{x}$ | training vector $(x_1, \ldots, x_i, \ldots, x_n)$. |
| $T$ | correct category or class for the training vector. |
| $\mathbf{w}_j$ | weight vector for $j$th output unit $(w_{1j}, \ldots, w_{ij}, \ldots, w_{nj})$. |
| $C_j$ | category or class represented by $j$th output unit. |
| $\|\mathbf{x} - \mathbf{w}_j\|$ | Euclidean distance between input vector and (weight vector for) $j$th output unit. |

*Step 0.* Initialize reference vectors (several strategies are discussed shortly); initialize learning rate, $\alpha(0)$.

*Step 1.* While stopping condition is false, do Steps 2–6.

*Step 2.* For each training input vector $\mathbf{x}$, do Steps 3–4.

*Step 3.* Find $J$ so that $\|\mathbf{x} - \mathbf{w}_J\|$ is a minimum.

*Step 4.* Update $\mathbf{w}_J$ as follows:

if $T = C_J$, then

$$\mathbf{w}_J(\text{new}) = \mathbf{w}_J(\text{old}) + \alpha[\mathbf{x} - \mathbf{w}_J(\text{old})];$$

if $T \neq C_J$, then

$$\mathbf{w}_J(\text{new}) = \mathbf{w}_J(\text{old}) - \alpha[\mathbf{x} - \mathbf{w}_J(\text{old})].$$

*Step 5.* Reduce learning rate.

*Step 6.* Test stopping condition:

The condition may specify a fixed number of iteratic
(i.e., executions of Step 1) or the learning rate reachin
sufficiently small value.

# LVQ Weight Initialization

- The simplest method of initializing the weight (reference) vectors is to take the first m training vectors and use them as weight vectors, the remaining vectors are then used for training.

- Another possible method for initializing the weights is to use k-means clustering or the self organizing maps to place the weights.

- Each weight vector is then calibrated by determining the input patterns that are closest to it, finding the class that the largest number of these input patterns belong to and assigning that class to the weight vector.

# LVQ Example

In this very simple example, two reference vectors will be used. The following input vectors represent two classes, 1 and 2:

| VECTOR | CLASS |
|---|---|
| (1, 1, 0, 0) | 1 |
| (0, 0, 0, 1) | 2 |
| (0, 0, 1, 1) | 2 |
| (1, 0, 0, 0) | 1 |
| (0, 1, 1, 0) | 2 |

The first two vectors will be used to initialize the two reference vectors. Thus, the first output unit represents class 1, the second class 2 (symbolically, $C_1 = 1$ and $C_2 = 2$). This leaves vectors (0, 0, 1, 1), (1, 0, 0, 0), and (0, 1, 1, 0) as the training vectors. Only one iteration (one epoch) is shown:

*Step 0.* Initialize weights:

$\mathbf{w}_1 = (1, 1, 0, 0)$;

$\mathbf{w}_2 = (0, 0, 0, 1)$.

Initialize the learning rate: $\alpha = .1$.

*Step 1.* Begin computations.

*Step 2.* For input vector $\mathbf{x} = (0, 0, 1, 1)$ with $T = 2$, do Steps 3–4.

*Step 3.* $J = 2$, since $\mathbf{x}$ is closer to $\mathbf{w}_2$ than to $\mathbf{w}_1$.

*Step 4.* Since $T = 2$ and $C_2 = 2$, update $\mathbf{w}_2$ as follows:

$$\mathbf{w}_2 = (0,0,0,1) + .1[(0,0,1,1) - (0,0,0,1)]$$

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

*Step 2.* For input vector $\mathbf{x} = (1, 0, 0, 0)$ with $T = 1$, do Steps 3–4.

*Step 3.* $J = 1$.

*Step 4.* Since $T = 1$ and $C_1 = 1$, update $\mathbf{w}_1$ as follows:

$$\mathbf{w}_1 = (1,1,0,0) + .1[(1,0,0,0) - (1,1,0,0)]$$

$$= (1,.9,0,0).$$

*Step 2.* For input vector $\mathbf{x} = (0, 1, 1, 0)$ with $T = 2$, do Steps 3–4.

*Step 3.* $J = 1$.

*Step 4.* Since $T = 2$, but $C_1 = 1$, update $\mathbf{w}_1$ as follows:

$$\mathbf{w}_1 = (1,.9,0,0) - .1[(0,1,1,0) - (1,.9,0,0)]$$

$$= (1.1,.89,-.1,0).$$

*Step 5.* This completes one epoch of training.
Reduce the learning rate.

*Step 6.* Test the stopping condition.

# LVQ2

- In the original LVQ only the reference vector that is close to the input vector is updated.
- The direction it is moved depends on whether the winning reference vector belongs to the same class as the input vector.
- In the improved algorithm like LVQ2 two vectors (a winner and a runner-up) learn if several conditions are satisfied.
- The idea is that if the input is approximately the same distance from both the winner and the runner-up then each of them should learn.

# LVQ2 (Cont..)

In the first modification, LVQ2, the conditions under which both vectors are modified are that:

1. The winning unit and the runner-up (the next closest vector) represent different classes.
2. The input vector belongs to the same class as the runner-up.
3. The distances from the input vector to the winner and from the input vector to the runner-up are approximately equal. This condition is expressed in terms of a window, using the following notation:

| | |
|---|---|
| $\mathbf{x}$ | current input vector; |
| $\mathbf{y}_c$ | reference vector that is closest to $\mathbf{x}$; |
| $\mathbf{y}_r$ | reference vector that is next to closest to $\mathbf{x}$ (the runner-up); |
| $\mathbf{d}_c$ | distance from $\mathbf{x}$ to $\mathbf{y}_c$; |
| $d_r$ | distance from $\mathbf{x}$ to $\mathbf{y}_r$. |

# LVQ 2 (Cont..)

To be used in updating the reference vectors, a window is defined as follows: The input vector $\mathbf{x}$ falls in the window if

$$\frac{d_c}{d_r} > 1 - \epsilon$$

and

$$\frac{d_r}{d_c} < 1 + \epsilon,$$

where the value of $\epsilon$ depends on the number of training samples; a value of .35 is typical [Kohonen, 1990a].

In LVQ2, the vectors $\mathbf{y}_c$ and $\mathbf{y}_r$ are updated if the input vector $\mathbf{x}$ falls in the window, $\mathbf{y}_c$ and $\mathbf{y}_r$ belong to different classes, and $\mathbf{x}$ belongs to the same class as $\mathbf{y}_r$. If these conditions are met, the closest reference vector and the runner up are updated:

$$\mathbf{y}_c(t+1) = \mathbf{y}_c(t) - \alpha(t)[\mathbf{x}(t) - \mathbf{y}_c(t)];$$

$$\mathbf{y}_r(t+1) = \mathbf{y}_r(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{y}_r(t)].$$