Νευρο-Ασαφής Υπολογιστική
Neuro-Fuzzy Computing

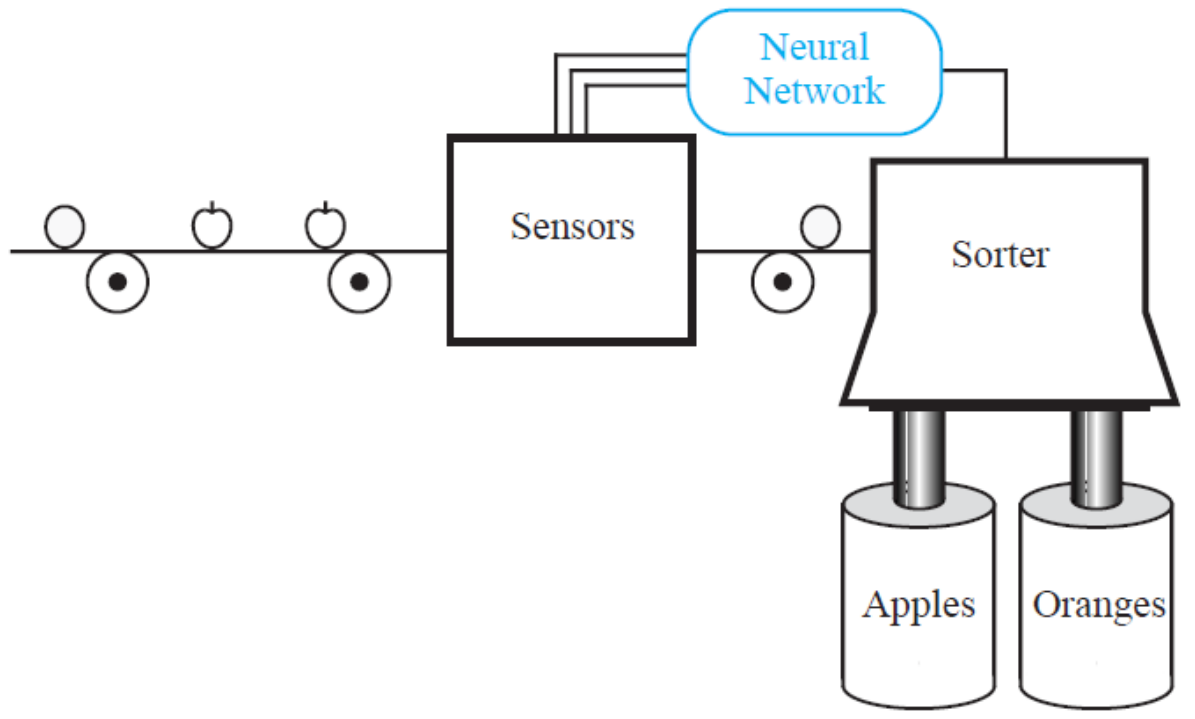
Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 17η

Hamming network (the instar)

Example classification application



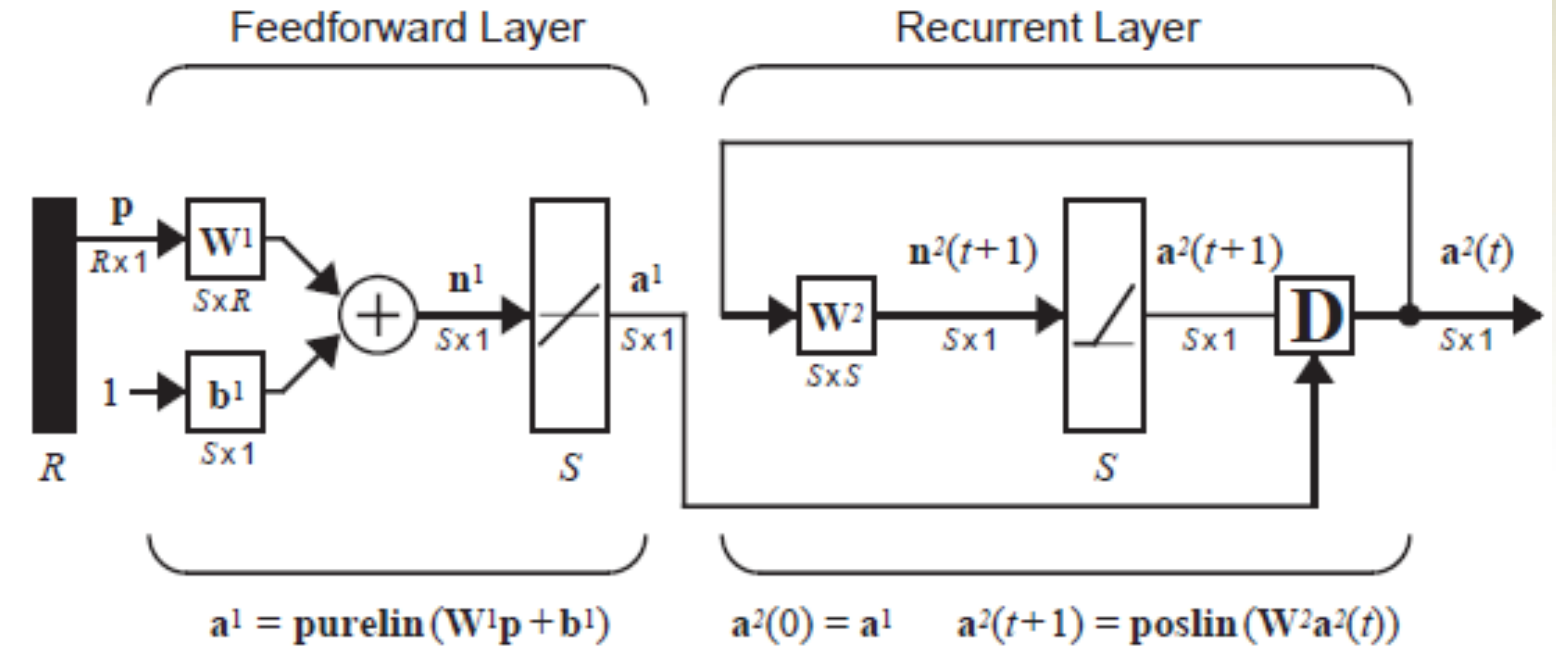
$$\mathbf{p} = \begin{bmatrix} \text{shape} \\ \text{texture} \\ \text{weight} \end{bmatrix}$$

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \quad \text{for orange}$$

$$\mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \quad \text{for apple}$$

Hamming network

- The Hamming network was designed explicitly to solve binary pattern recognition problems (where each element of the input vector has only two possible values)
- It uses both feedforward and recurrent (feedback) layers
- The number of neurons in the first layer is the same as the number of neurons in the second layer



Hamming network

- The objective of the Hamming network is to decide which prototype vector is closest to the input vector: this decision is indicated by the output of the recurrent layer
- There is one neuron in the recurrent layer for each prototype pattern
- When the recurrent layer converges, there will be only one neuron with nonzero output. This neuron indicates the prototype pattern that is closest to the input vector

Hamming network: Feedforward layer

- The feedforward layer performs a correlation, or inner product, between each of the prototype patterns and the input pattern
- In order for the feedforward layer to perform this correlation, the rows of the weight matrix in the feedforward layer, represented by the connection matrix , are set to the prototype patterns. For our example:

$$\mathbf{W}^1 = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix}$$

- The feedforward layer uses a linear transfer function, and each element of the bias vector is equal to R, where R is the number of elements in the input vector. For our example the bias vector would be:

$$\mathbf{b}^1 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

Hamming network: Feedforward layer

- With these choices for the weight matrix and bias vector, the output of the feedforward layer is:

$$\mathbf{a}^1 = \mathbf{W}^1\mathbf{p} + \mathbf{b}^1 = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \end{bmatrix}\mathbf{p} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^T\mathbf{p} + 3 \\ \mathbf{p}_2^T\mathbf{p} + 3 \end{bmatrix}$$

- Note that the outputs of the feedforward layer are equal to the inner products of each prototype pattern with the input, plus R. For two vectors of equal length (norm), their inner product will be largest when the vectors point in the same direction, and will be smallest when they point in opposite directions
- By adding R to the inner product we guarantee that the outputs of the feedforward layer can never be negative. This is required for proper operation of the recurrent layer

Hamming network: Feedforward layer

- This network is called the Hamming network because the neuron in the feedforward layer with the largest output will correspond to the prototype pattern that is closest in Hamming distance to the input pattern
  - The Hamming distance between two vectors is equal to the number of elements that are different. It is defined only for binary vectors
    - You may try to prove that the outputs of the feedforward layer are equal to $2R$ minus twice the Hamming distances from the prototype patterns to the input pattern

Hamming network: Recurrent layer

- The recurrent layer of the Hamming network is what is known as a “competitive” layer
- The neurons in this layer are initialized with the outputs of the feedforward layer, which indicate the correlation between the prototype patterns and the input vector
- Then the neurons compete with each other to determine a winner. After the competition, only one neuron will have a nonzero output
- The winning neuron indicates which category of input was presented to the network (for our example the two categories are apples and oranges). The equations that describe the competition are:

$$\mathbf{a}^2(0) = \mathbf{a}^1 \quad \text{and} \quad \mathbf{a}^2(t+1) = \mathbf{poslin}(\mathbf{W}^2\mathbf{a}^2(t))$$

Hamming network: Recurrent layer

- The weight matrix $\mathbf{W}^2$ has the form:

$$\mathbf{W}^2 = \begin{bmatrix} 1 & -\epsilon \\ -\epsilon & 1 \end{bmatrix}$$

  where ε is some number less than 1/(S-1), and S is the number of neurons in the recurrent layer

- An iteration of the recurrent layer proceeds as follows:

$$\mathbf{a}^2(t+1) = \mathbf{poslin}\left(\begin{bmatrix} 1 & -\epsilon \\ -\epsilon & 1 \end{bmatrix}\mathbf{a}^2(t)\right) = \mathbf{poslin}\left(\begin{bmatrix} a_1^2(t) - \epsilon a_2^2(t) \\ a_2^2(t) - \epsilon a_1^2(t) \end{bmatrix}\right)$$

- Each element is reduced by the same fraction of the other. The larger element will be reduced by less, and the smaller element will be reduced by more, therefore the difference between large and small will be increased. The effect of the recurrent layer is to zero out all neuron outputs, except the one with the largest initial value (which corresponds to the prototype pattern that is closest in Hamming distance to the input)

Hamming network: Recurrent layer

- Let us try an oblong orange: $\mathbf{p} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$
- The output of the feedforward layer will be:

$$\mathbf{a}^1 = \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} (1+3) \\ (-1+3) \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

  which will be the initial condition for the recurrent layer

- The weight matrix for the recurrent layer will be the $\mathbf{W}^2$ as given before with ε=1/2
- The first iteration of the recurrent layer produces:

$$\mathbf{a}^2(1) = \mathbf{poslin}(\mathbf{W}^2\mathbf{a}^2(0)) = \begin{cases} \mathbf{poslin}\left(\begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix} \right) \\ \mathbf{poslin}\left(\begin{bmatrix} 3 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \end{cases}$$

Hamming network: Recurrent layer

- The second iteration produces:

$$\mathbf{a}^2(2) = \mathbf{poslin}(\mathbf{W}^2\mathbf{a}^2(1)) = \begin{cases} \mathbf{poslin}\left(\begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}\begin{bmatrix} 3 \\ 0 \end{bmatrix}\right) \\ \mathbf{poslin}\left(\begin{bmatrix} 3 \\ -1.5 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \end{cases}$$

- Since the outputs of successive iterations produce the same result, the network has converged
- Prototype pattern number one, the *orange*, is chosen as the correct match, since neuron number one has the only nonzero output. (Recall that the first element of $\mathbf{a}^1$ was $\mathbf{p}^T{}_1\mathbf{p}+3$.)
- This is the correct choice, since the Hamming distance from the *orange* prototype to the input pattern is 1, and the Hamming distance from the *apple* prototype to the input pattern is 2

Hamming network: Recurrent layer

- Recall that second-layer weights $\mathbf{W}^2$ are set so that the diagonal elements are 1, and the off-diagonal elements have a small negative value:
- This matrix produces *lateral inhibition*, in which the output of each neuron has an inhibitory effect on all of the other neurons. To illustrate this effect, substitute weight values of 1 and –ε for the appropriate elements of $\mathbf{W}^2$, and then we get for a single neuron:

$$a_i^2(t+1) = poslin\left(a_i^2(t) - \epsilon \sum_{j \neq i} a_j^2(t)\right)$$

Hamming network: Recurrent layer

- At each iteration, each neuron's output will decrease in proportion to the sum of the other neurons' outputs (with a minimum output of 0)
- The output of the neuron with the largest initial condition will decrease more slowly than the outputs of the other neurons. Eventually that neuron will be the only one with a positive output
- At this point the network has reached steady state
  - The index of the second-layer neuron with a stable positive output is the index of the prototype vector that best matched the input
- This is called a *winner-take-all competition*, since only one neuron will have a nonzero output

Competitive learning

Competitive layer

- The second-layer neurons in the Hamming network are said to be in *competition* because each neuron excites itself and inhibits all the other neurons
- To simplify our discussions in the remainder of this chapter, we will define a transfer function that does the job of a recurrent competitive layer:

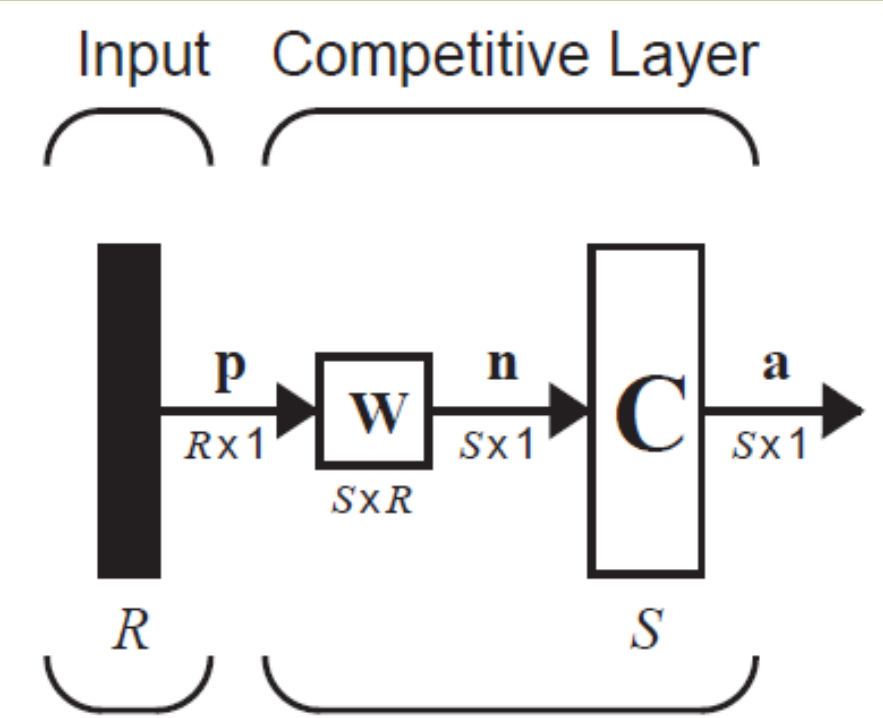
a = compet(n)

- It works by finding the index *i** of the neuron with the largest net input, and setting its output to 1 (with ties going to the neuron with the lowest index). All other outputs are set to 0:

$$a_i = \begin{cases} 1 & i = i^\star \\ 0 & i \neq i^\star \end{cases} \text{ where } n_{i^\star} \geq n_i, \forall i \text{ and } i^\star \leq i, \forall n_i = n_{i^\star}$$

Competitive layer

- Replacing the recurrent layer of the Hamming network with a competitive transfer function on the first layer will simplify our presentations



Competitive layer

- As with the Hamming network, the prototype vectors are stored in the rows of **W**. The net input **n** calculates the distance between the input vector **p** and each prototype ${}_{i}\mathbf{w}$ (assuming vectors have normalized lengths of L)
- The net input n_i of each neuron is proportional to the angle θ_i between **p** and the prototype vector ${}_{i}\mathbf{w}$

$$\mathbf{n} = \mathbf{W}\mathbf{p} = \begin{bmatrix} {}_{1}\mathbf{w}^T \\ {}_{2}\mathbf{w}^T \\ \vdots \\ {}_{S}\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} {}_{1}\mathbf{w}^T\mathbf{p} \\ {}_{2}\mathbf{w}^T\mathbf{p} \\ \vdots \\ {}_{S}\mathbf{w}^T\mathbf{p} \end{bmatrix} = \begin{bmatrix} L^2 cos\theta_1 \\ L^2 cos\theta_2 \\ \vdots \\ L^2 cos\theta_S \end{bmatrix}$$

- The competitive transfer function assigns an output of 1 to the neuron whose weight vector points in the direction closest to the input vector:

a = compet(n)

Competitive learning

- We can now design a competitive network classifier by setting the rows of to the desired prototype vectors
- However, we would like to have a learning rule that could be used to train the weights in a competitive network, without knowing the prototype vectors
- One such learning rule is the *instar rule*:

$$ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha a_i(q)(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) $$

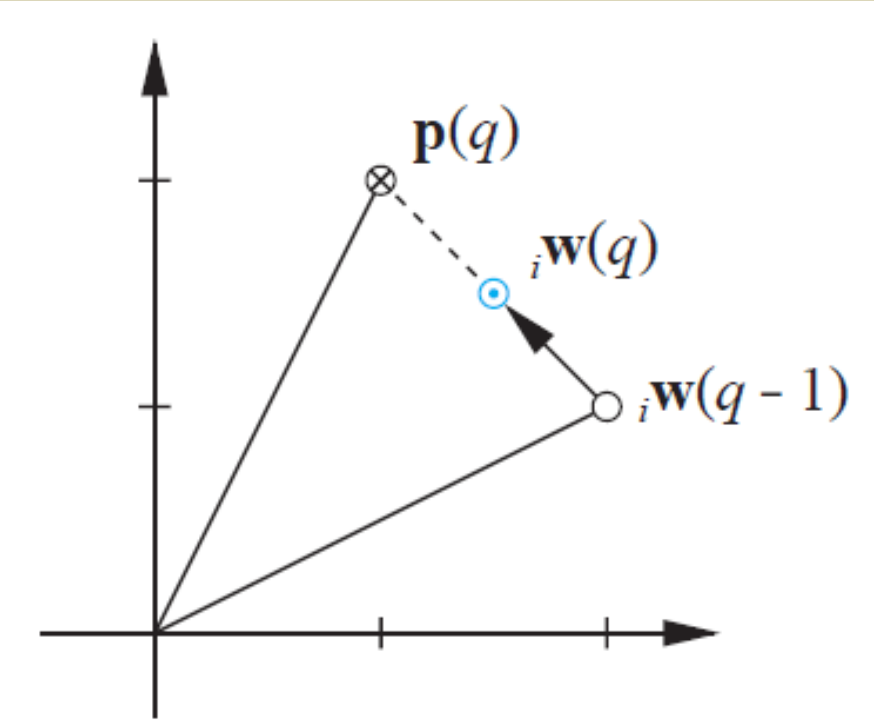
- For the competitive network, **a** is only nonzero for the winning neuron ($i = i^*$). Therefore, we can get the same results using the *Kohonen rule*:

$$ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) = $$
$$ = (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) $$

and $ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) $

Competitive learning

- Thus, the row of the weight matrix that is closest to the input vector (or has the largest inner product with the input vector) moves toward the input vector
- It moves along a line between the old row of the weight matrix and the input vector, as shown below:

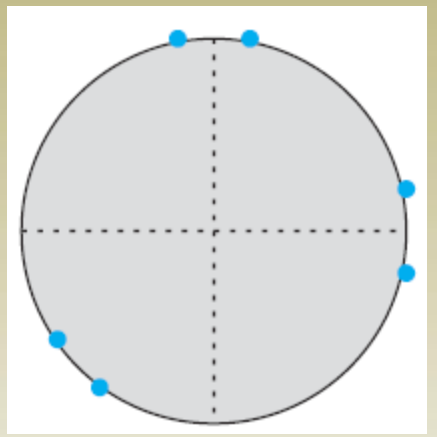


Competitive learning example

- Let's use the six vectors to demonstrate how a competitive layer learns to classify vectors

$$\mathbf{p}_1 = \begin{bmatrix} -0.1961 \\ 0.9806 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 0.9806 \\ 0.1961 \end{bmatrix}$$

$$\mathbf{p}_4 = \begin{bmatrix} 0.9806 \\ -0.1961 \end{bmatrix}, \mathbf{p}_5 = \begin{bmatrix} -0.5812 \\ -0.8137 \end{bmatrix}, \mathbf{p}_6 = \begin{bmatrix} -0.8137 \\ -0.5812 \end{bmatrix}$$

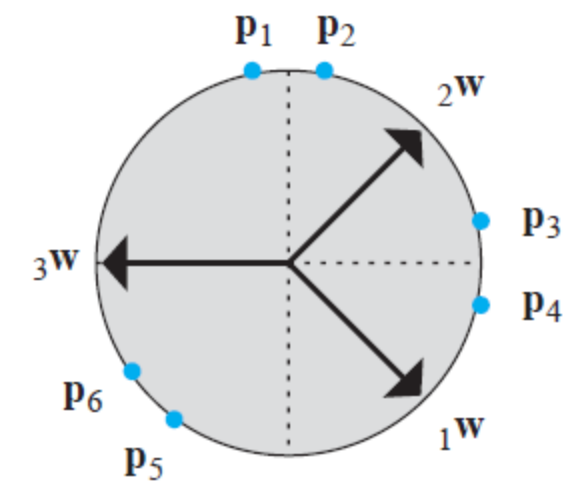


- Let our competitive network have three neurons, and therefore it can classify vectors into three classes
- The "randomly" chosen normalized initial weights are:

$$_1\mathbf{w} = \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix}, _2\mathbf{w} = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}, _3\mathbf{w} = \begin{bmatrix} -1.0000 \\ 0.0000 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} _1\mathbf{w}^T \\ _2\mathbf{w}^T \\ _3\mathbf{w}^T \end{bmatrix}$$

Competitive learning example

- The data vectors are shown at left, with the weight vectors displayed as arrows. Let's present the vector $\mathbf{p}_2$ to the network:



$$\mathbf{a} = \mathbf{compet}(\mathbf{W}\mathbf{p_2}) =$$

$$= \mathbf{compet}\left(\begin{bmatrix} 0.7071 & -0.7071 \\ 0.7071 & 0.7071 \\ -1.0000 & 0.0000 \end{bmatrix}\begin{bmatrix} 0.1961 \\ 0.8906 \end{bmatrix}\right)$$

$$= \mathbf{compet}\left(\begin{bmatrix} -0.5547 \\ 0.8321 \\ -0.1961 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

- The second neuron's weight vector was closest to $\mathbf{p}_2$, so it won the competition (i*=2) and output a 1

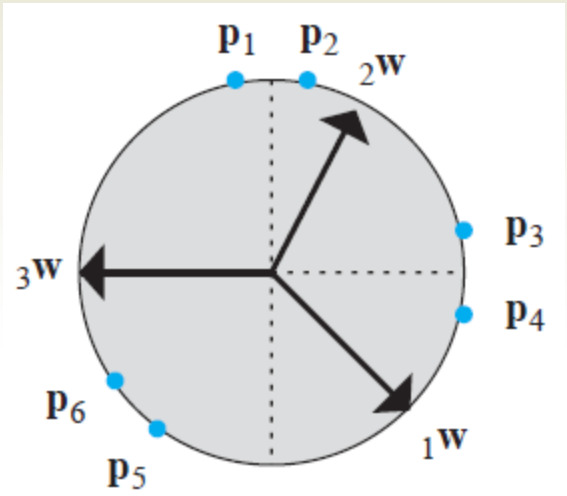
Competitive learning example

- We now apply the Kohonen learning rule to the winning neuron with a learning rate of $\alpha = 0.5$

$$ {}_2\mathbf{w}^{new} = {}_2\mathbf{w}^{old} + \alpha(\mathbf{p}_2 - {}_2\mathbf{w}^{old}) $$

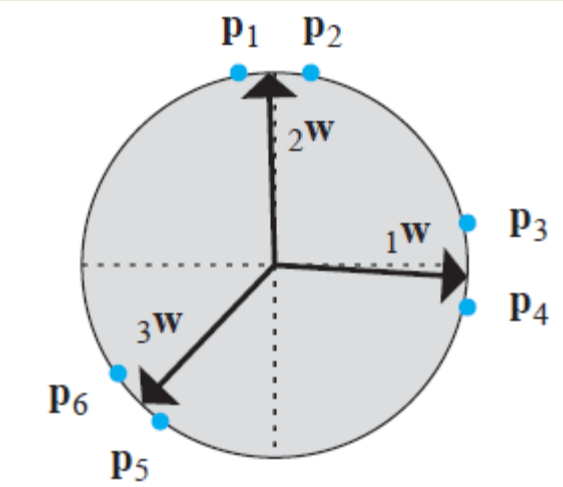
$$ = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} + 0.5\left(\begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix} - \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} \right) = \begin{bmatrix} 0.4516 \\ 0.8438 \end{bmatrix} $$

- The Kohonen rule moves ${}_2\mathbf{w}$ closer to $\mathbf{p}_2$, as can be seen in the diagram below



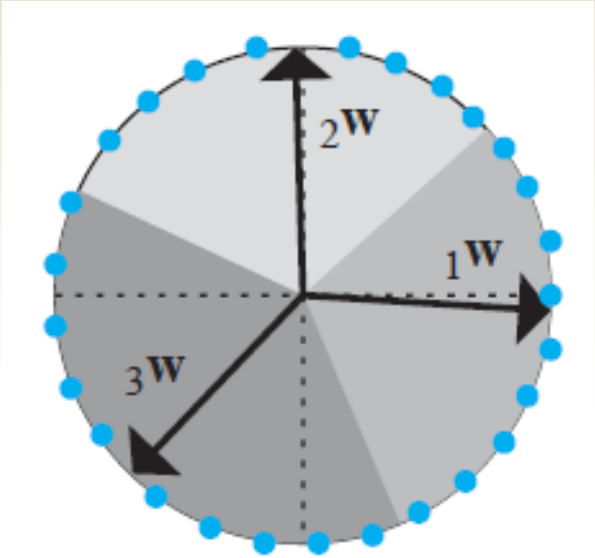
Competitive learning example

- If we continue choosing input vectors at random and presenting them to the network, then at each iteration the weight vector closest to the input vector will move toward that vector. Eventually, each weight vector will point at a different cluster of input vectors. Each weight vector becomes a prototype for a different cluster
- This problem is simple and we can predict which weight vector will point at which cluster
- The final weights will look



Competitive learning example

- Once the network has learned to cluster the input vectors, it will classify new vectors accordingly.
- The figure below uses shading to show which region each neuron will respond to
- The competitive layer assigns each input vector **p** to one of these classes by producing an output of 1 for the neuron whose weight vector is closest to **p**



Problems with Competitive layers

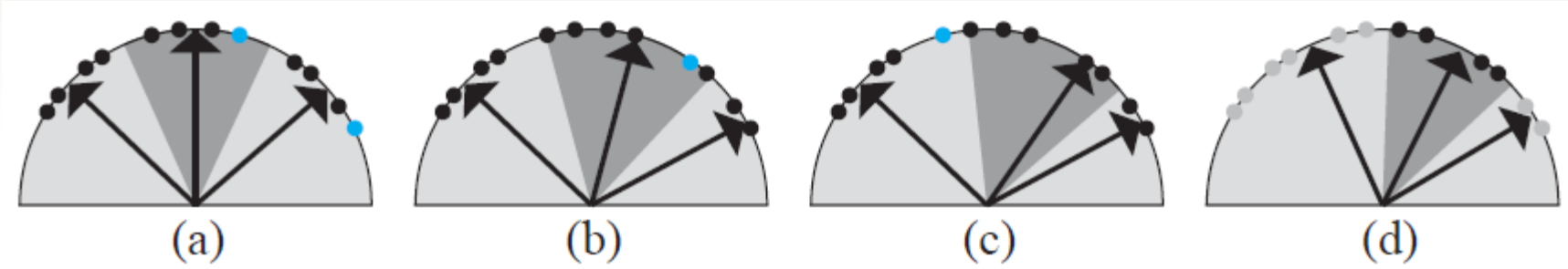
- Competitive layers make efficient adaptive classifiers, but they do suffer from a few problems
- The first problem is that the choice of learning rate forces a trade-off between the speed of learning and the stability of the final weight vectors
  - A learning rate near zero results in slow learning. However, once a weight vector reaches the center of a cluster it will tend to stay close to the center
  - In contrast, a learning rate near 1.0 results in fast learning. However, once the weight vector has reached a cluster, it will continue to oscillate as different vectors in the cluster are presented

Problems with Competitive layers

- Sometimes this trade-off between fast learning and stability can be used to advantage
- Initial training can be done with a large learning rate for fast learning
- Then the learning rate can be decreased as training progresses, to achieve stable prototype vectors
- Unfortunately, this technique will not work if the network needs to continuously adapt to new arrangements of input vectors

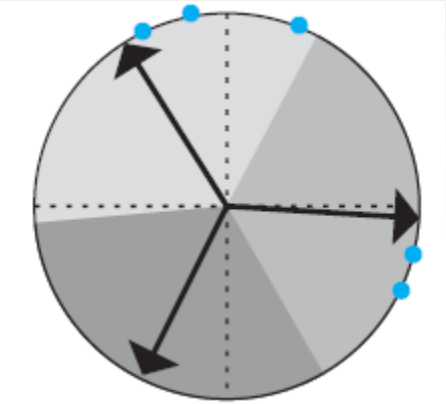
Problems with Competitive layers

- A more serious stability problem occurs when clusters are close together: In certain cases, a weight vector forming a prototype of one cluster may "invade" the territory of another weight vector, and therefore upset the current classification scheme
- The series of four diagrams illustrate this problem. Two input vectors (shown with blue circles in diagram (a)) are presented several times. The result is that the weight vectors representing the middle and right clusters shift to the right. Eventually one of the right cluster vectors is reclassified by the center weight vector. Further presentations move the middle vector over to the right until it "loses" some of its vectors, which then become part of the class associated with the left weight vector



Problems with Competitive layers

- A third problem with competitive learning is that occasionally a neuron's initial weight vector is located so far from any input vectors that it never wins the competition, and therefore never learns. The result is a "dead" neuron, which does nothing useful
- For example, the downward-pointing weight vector in the diagram to the left will never learn, regardless of the order in which vectors are presented. One solution to this problem consists of adding a negative bias to the net input of each neuron and then decreasing the bias each time the neuron wins. This will make it harder for a neuron to win the competition if it has won often. This mechanism is sometimes called a "conscience"



Problems with Competitive layers

- Finally, a competitive layer always has as many classes as it has neurons. This may not be acceptable for some applications, especially when the number of clusters is not known in advance
- In addition, for competitive layers, each class consists of a convex region of the input space. Competitive layers cannot form classes with nonconvex regions or classes that are the union of unconnected regions