Νευρο-Ασαφής Υπολογιστική
Neuro-Fuzzy Computing

Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 18η

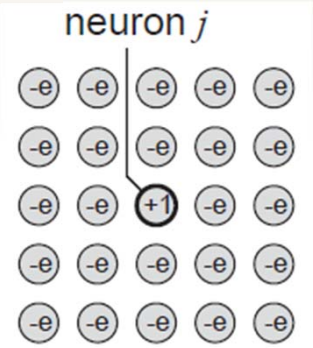
Competitive layers in biology

Competitive layers in biology

- In biological neural networks, neurons are typically arranged in two-dimensional layers, in which they are densely interconnected through lateral feedback
- Often weights vary as a function of the distance between the neurons they connect
- For example, the weights for Layer 2 of the Hamming network are assigned as follows:

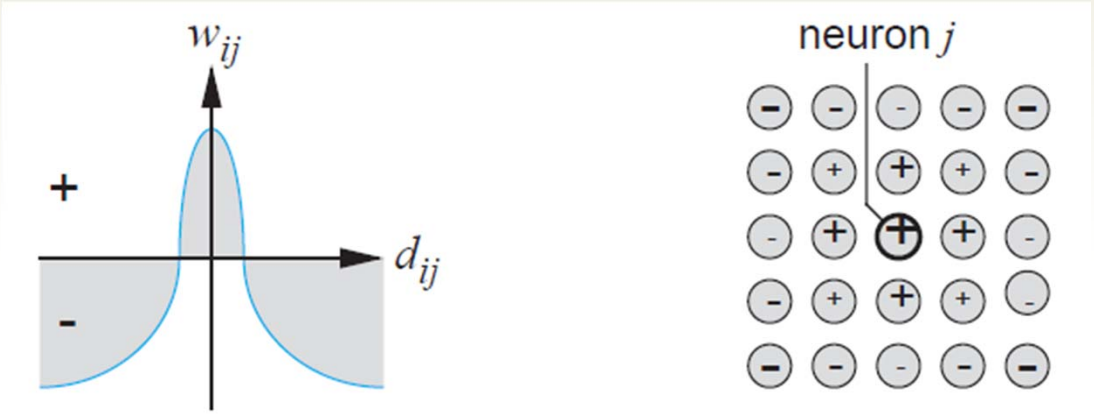
$$w_{ij} = \begin{cases} 1, & \text{if } i = j \\ -\epsilon & \text{if } i \neq j \end{cases} \qquad w_{ij} = \begin{cases} 1, & \text{if } d_{ij} = 0 \\ -\epsilon & \text{if } d_{ij} > 0 \end{cases}$$

- These equations will assign the weight values shown
  - Each neuron i is labeled with the value of the weight w_{ij}, which comes from it to the neuron marked j



Competitive layers in biology

- The term *on-center/off-surround* is often used to describe such a connection pattern between neurons
  - Each neuron reinforces itself (center), while inhibiting all other neurons (surround)
- It turns out that this is a crude approximation of biological competitive layers
  - In biology, a neuron reinforces not only itself, but also those neurons close to it. Typically, the transition from reinforcement to inhibition occurs smoothly as the distance between neurons increases [The Mexican-hat function]



Competitive layers in biology

- Biological competitive systems,
  - in addition to having a gradual transition between excitatory and inhibitory regions of the on-center/off-surround connection pattern
- [They] also have a weaker form of competition than the winner-take-all competition
  - Instead of a single active neuron (winner), biological networks generally have “bubbles” of activity that are centered around the most active neuron
  - This is caused
    - in part by the form of the on-center/off-surround connectivity pattern, and also
    - by nonlinear feedback connections

Self-Organizing Feature Maps

Self-Organizing Feature Maps

- In order to emulate the activity bubbles of biological systems, without having to implement the nonlinear on-center/off-surround feedback connections, Kohonen designed the following simplification:
  - His self-organizing feature map (SOFM) network first determines the winning neuron using the same procedure as the competitive layer
  - Next, the weight vectors for all neurons within a certain neighborhood of the winning neuron are updated using the Kohonen rule:

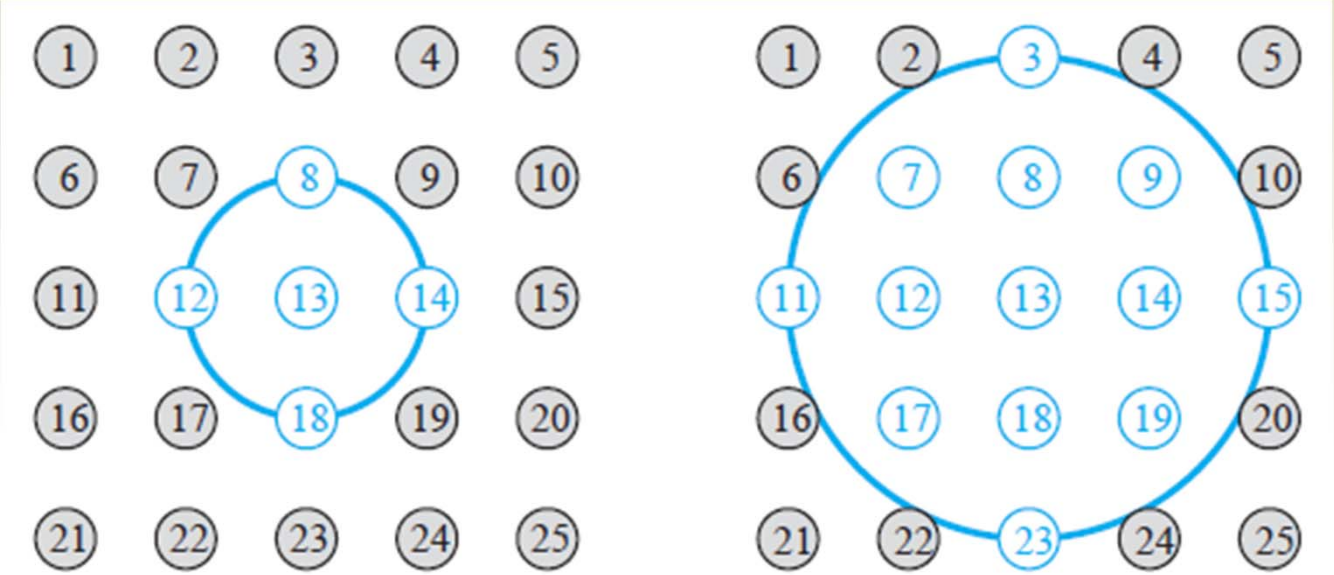
$$
\begin{aligned}
{}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) = \\
&= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) \qquad i \in N_{i^\star}(d)
\end{aligned}
$$

where the neighborhood $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*:

$$N_i(d) = \{j: d_{ij} \leq d\}$$

Self-Organizing Feature Maps

- When a vector **p** is presented, the weights of the winning neuron *and* its neighbors will move toward **p**. The result is that, after many presentations, neighboring neurons will have learned vectors similar to each other
- To demonstrate the concept of a neighborhood, consider the two figures below:

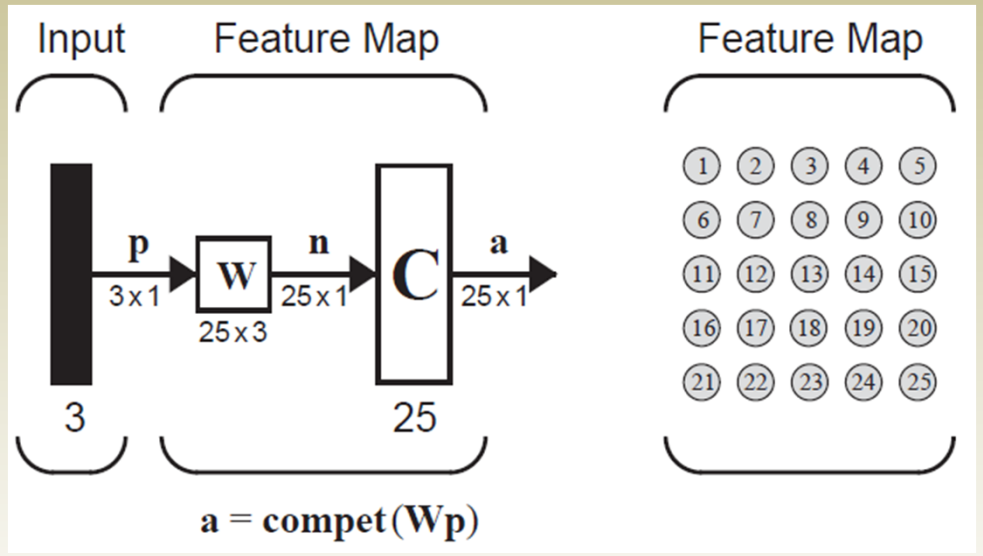


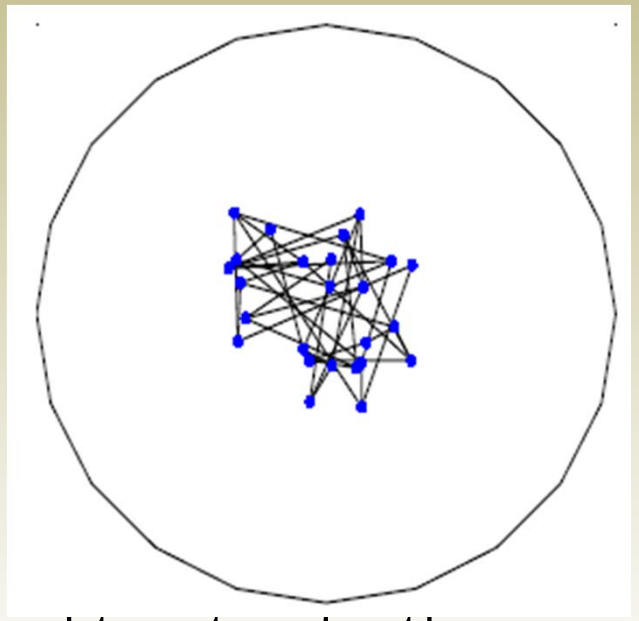
Self-Organizing Feature Maps

- The neurons in an SOFM do not have to be arranged in a two-dimensional pattern
- It is possible to use a one-dimensional arrangement, or even three or more dimensions
  - For a one-dimensional SOFM, a neuron will only have two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line)
- It is also possible to define distance in different ways
  - For instance, Kohonen has suggested rectangular and hexagonal neighborhoods for efficient implementation
- The performance of the network is not sensitive to the exact shape of the neighborhoods

Self-Organizing Feature Maps: Example

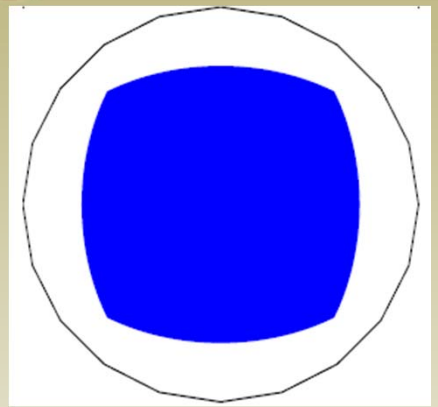
- Let's demonstrate the performance of an SOFM network Below we show a feature map and the two-dimensional topology of its neurons





- The diagram on the right shows the initial weight vectors for the feature map. Each three-element weight vector is represented by a dot on the sphere. (The weights are normalized, therefore they will fall on the surface of a sphere.) Dots of neighboring neurons are connected by lines so you can see how the physical topology of the network is arranged in the input space

Self-Organizing Feature Maps: Example



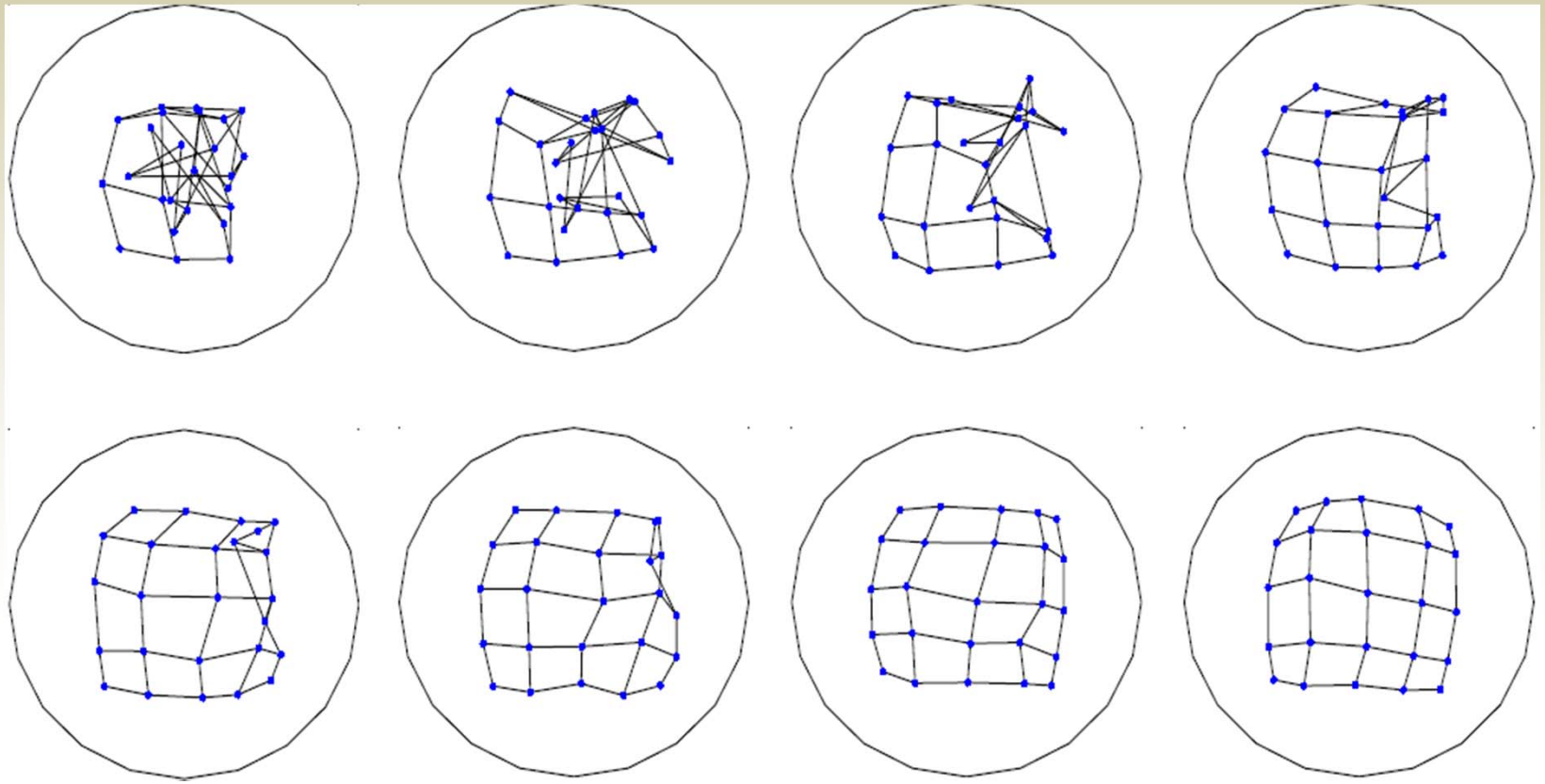
- The diagram to the left shows a square region on the surface of the sphere
- We will randomly pick vectors in this region and present them to the feature map
- Each time a vector is presented, the neuron with the closest weight vector will win the competition
- The winning neuron and its neighbors move their weight vectors closer to the input vector (and therefore to each other)
  - For this example we are using a neighborhood with a radius of 1

Self-Organizing Feature Maps: Example

- The weight vectors have two tendencies
  - first, they spread out over the input space as more vectors are presented
  - second, they move toward the weight vectors of neighboring neurons
- These two tendencies work together to rearrange the neurons in the layer so that they evenly classify the input space

Self-Organizing Feature Maps: Example

- The series of diagrams shows how the weights of the 25 neurons spread out over the active input space and organize themselves to match its topology (the input vectors were generated with equal probability from any point in the input space. Therefore, the neurons classify roughly equal areas of the input space.)



Improving Feature Maps

- One method to improve the performance of the feature map is to vary the size of the neighborhoods during training
  - Initially, the neighborhood size, d, is set large. As training progresses, is gradually reduced, until it only includes the winning neuron. This speeds up self-organizing and makes twists in the map very unlikely
- The learning rate can also be varied over time. An initial rate of 1 allows neurons to quickly learn presented vectors. During training, the learning rate is decreased asymptotically toward 0, so that learning becomes stable
- Another alteration that speeds self-organization is to have the winning neuron use a larger learning rate than the neighboring neurons

Improving Feature Maps

- Finally, both competitive layers and feature maps often use an alternative expression for net input
- Instead of using the inner product, they can directly compute the distance between the input vector and the prototype vectors
- The advantage of using the distance is that input vectors do not need to be normalized
- This alternative net input expression is introduced in the next section on LVQ networks

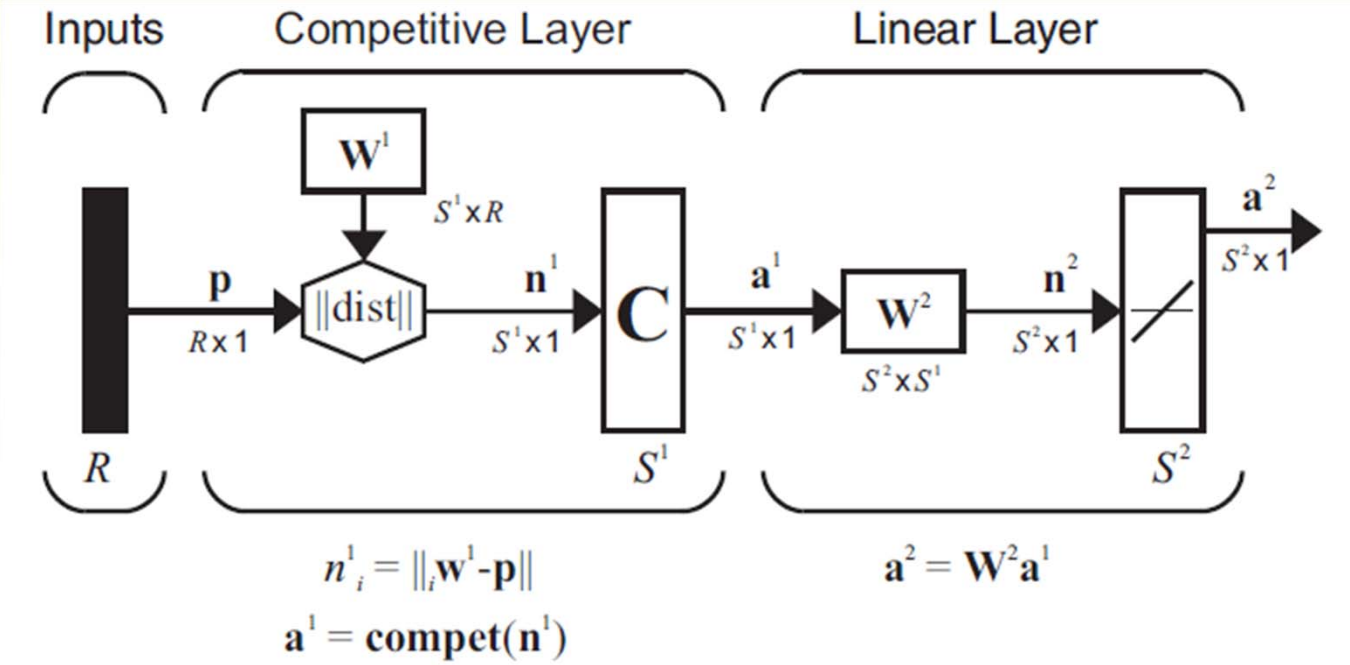
Learning Vector Quantization (LVQ)

Learning Vector Quantization

- The learning vector quantization (LVQ) network is a hybrid network
  - It uses both unsupervised and supervised learning to form classifications
- In the LVQ network, each neuron in the first layer is assigned to a class, with several neurons often assigned to the same class. Each class is then assigned to one neuron in the second layer
  - The number of neurons in the first layer, S^1, will therefore always be at least as large as the number of neurons in the second layer, S^2, and will usually be larger

Learning Vector Quantization

- As with the competitive network, each neuron in the first layer of the LVQ network learns a prototype vector, which allows it to classify a region of the input space
- Instead of computing the proximity of the input and weight vectors by using the inner product, we will simulate the LVQ networks by calculating the distance directly



Learning Vector Quantization

- One advantage of calculating the distance directly is that vectors need not be normalized
  - When the vectors are normalized, the response of the network will be the same, whether the inner product is used or the distance is directly calculated
- The net input of the first layer of the LVQ will be:

$$n_i^1 = -||_i\mathbf{w}^1 - \mathbf{p}||$$

or in vector form:

$$\mathbf{n}^1 = -\begin{bmatrix} ||_1\mathbf{w} - \mathbf{p}|| \\ ||_2\mathbf{w} - \mathbf{p}|| \\ \vdots \\ ||_{S^1}\mathbf{w} - \mathbf{p}|| \end{bmatrix}$$

- The output of the 1st layer of the LVQ is: $\mathbf{a^1}=\mathbf{compet}(\mathbf{n^1})$
  - Therefore the neuron whose weight vector is closest to the input vector will output a 1, and the other neurons will output 0

Learning Vector Quantization

- Thus far, the LVQ network behaves exactly like the competitive network (at least for normalized vectors)
- However:
  - In the competitive network, the neuron with the nonzero output indicates which class the input vector belongs to
  - For the LVQ network, the winning neuron indicates a subclass
    - There may be several different neurons (subclasses) that make up each class
- The 2nd layer of the LVQ network is used to combine subclasses into a single class
  - This is done with the $\mathbf{W}^2$ matrix
    - The columns of $\mathbf{W}^2$ represent subclasses, and the rows represent classes. $\mathbf{W}^2$ has a single 1 in each column, with the other elements set to zero. The row in which the 1 occurs indicates which class the appropriate subclass belongs to

$$(w_{ki}^2 = 1) \rightarrow \text{subclass } i \text{ is part of class } k$$

Learning Vector Quantization

- The process of combining subclasses to form a class allows the LVQ network to create complex class boundaries
- A standard competitive layer has the limitation that it can only create decision regions that are convex
- The LVQ network overcomes this limitation

LVQ Learning

- The learning in the LVQ network combines competitive learning with supervision
- As with all supervised learning algorithms, it requires a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

- Each target vector must contain only zeros, except for a single 1. The row in which the 1 appears indicates the class to which the input vector belongs
  - For example, if we have a problem where we would like to classify a particular three-element vector into the second of four classes, we can express this as

$$\left\{\mathbf{p}_1 = \begin{bmatrix} \sqrt{1/2} \\ 0 \\ \sqrt{1/2} \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}\right\}$$

LVQ Learning

- Before learning can occur, each neuron in the first layer is assigned to an output neuron
- This generates the matrix $\mathbf{W}^2$
- Typically, equal numbers of hidden neurons are connected to each output neuron, so that each class can be made up of the same number of convex regions
- All elements of are set to zero, except for the following:

  If hidden neuron i is to be assigned to class k, then set $w_{ki}^2 = 1$

- Once $\mathbf{W}^2$ is defined, it will never be altered. The hidden weights $\mathbf{W}^1$ are trained with a variation of the Kohonen rule

LVQ Learning

- The LVQ learning rule proceeds as follows:
  - At each iteration, an input vector $\mathbf{p}$ is presented to the network, and the distance from $\mathbf{p}$ to each prototype vector is computed
  - The hidden neurons compete, neuron i^* wins the competition, and the i^*-th element of $\mathbf{a}^1$ is set to 1
  - Next, $\mathbf{a}^1$ is multiplied by $\mathbf{W}^2$ to get the final output $\mathbf{a}^2$, which also has only one nonzero element, k^*, indicating that is being assigned to class k^*

LVQ Learning

- The Kohonen rule is used to improve the hidden layer of the LVQ network in two ways
- First, if **p** is classified correctly, then we move the weights ${}_{i^*}\mathbf{w}^1$ of the winning hidden neuron toward **p**:

$$ {}_{i^\star}\mathbf{w}^1(q) = {}_{i^\star}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^\star}\mathbf{w}^1(q-1)) \text{ if } a^2_{k^\star} = t_{k^\star} = 1 $$

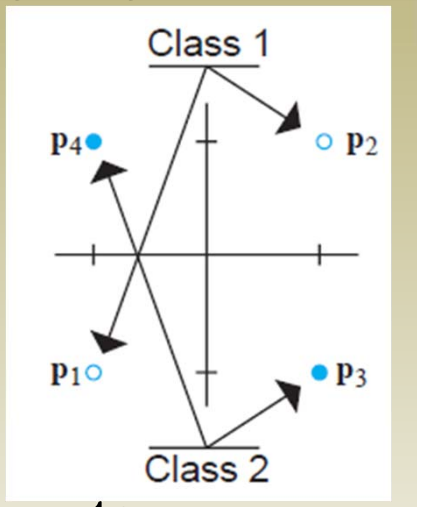
- Second, if **p** was classified incorrectly, then we know that the wrong hidden neuron won the competition, and therefore we move its weights ${}_{i^*}\mathbf{w}^1$ away from **p**:

$$ {}_{i^\star}\mathbf{w}^1(q) = {}_{i^\star}\mathbf{w}^1(q-1) - \alpha(\mathbf{p}(q) - {}_{i^\star}\mathbf{w}^1(q-1)) \text{ if } a^2_{k^\star} = 1 \neq t_{k^\star} = 0 $$

- The result will be that each hidden neuron moves toward vectors that fall into the class for which it forms a subclass and away from vectors that fall into other classes

LVQ Learning: Example

- We would like to train an LVQ network to solve the following classification problem:



$$\text{class 1:}\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix}1\\1\end{bmatrix}\right\}, \text{class 2:}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\-1\end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}\right\}$$

- We begin by assigning target vectors to each input:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix}1\\0\end{bmatrix}\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}1\\0\end{bmatrix}\right\}$$

$$\left\{\mathbf{p}_3 = \begin{bmatrix}1\\-1\end{bmatrix}, \mathbf{t}_3 = \begin{bmatrix}0\\1\end{bmatrix}\right\}, \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix}0\\1\end{bmatrix}\right\}$$

LVQ Learning: Example

- We now must choose how many subclasses will make up each of the two classes
- If we let each class be the union of two subclasses, we will end up with four neurons in the hidden layer. The output layer weight matrix will be

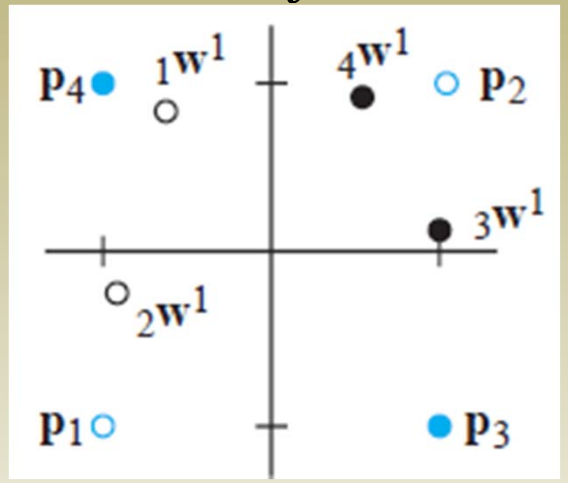
$$\mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$\mathbf{W}^2$ connects hidden neurons 1 and 2 to output neuron 1. It connects hidden neurons 3 and 4 to output neuron 2.

Each class will be made up of two convex regions

LVQ Learning: Example

- The row vectors in $\mathbf{W}^1$ are initially set to random values
- They can be seen here:



- The weights belonging to the two hidden neurons that define class 1 are marked with hollow circles. The weights defining class 2 are marked with solid circles. The values for these weights are:

$$ {}_1\mathbf{w}^1 = \begin{bmatrix} -0.543 \\ 0.840 \end{bmatrix}, \; {}_2\mathbf{w}^1 = \begin{bmatrix} -0.969 \\ -0.249 \end{bmatrix}, \; {}_3\mathbf{w}^1 = \begin{bmatrix} 0.997 \\ 0.094 \end{bmatrix}, \; {}_4\mathbf{w}^1 = \begin{bmatrix} 0.456 \\ 0.954 \end{bmatrix} $$

LVQ Learning: Example

- At each iteration of the training process, we present an input vector, find its response, and then adjust the weights. In this case we will begin by presenting $\mathbf{p}_3$

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1) = \mathbf{compet}\left(\begin{bmatrix} -\|{}_1\mathbf{w}^1 - \mathbf{p}_3\| \\ -\|{}_2\mathbf{w}^1 - \mathbf{p}_3\| \\ -\|{}_3\mathbf{w}^1 - \mathbf{p}_3\| \\ -\|{}_4\mathbf{w}^1 - \mathbf{p}_3\| \end{bmatrix}\right)$$

$$= \mathbf{compet}\left(\begin{bmatrix} -\|[-0.543\ \ 0.840]^T - [1\ -1]^T\| \\ -\|[-0.969\ -0.249]^T - [1\ -1]^T\| \\ -\|[0.997\ \ 0.094]^T - [1\ -1]^T\| \\ -\|[0.456\ \ 0.954]^T - [1\ -1]^T\| \end{bmatrix}\right) = \mathbf{compet}\left(\begin{bmatrix} -2.40 \\ -2.11 \\ -1.09 \\ -2.03 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

LVQ Learning: Example

- The third hidden neuron has the closest weight vector to $\mathbf{p}_3$
- In order to determine which class this neuron belongs to, we multiply $\mathbf{a}^1$ by $\mathbf{W}^2$

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

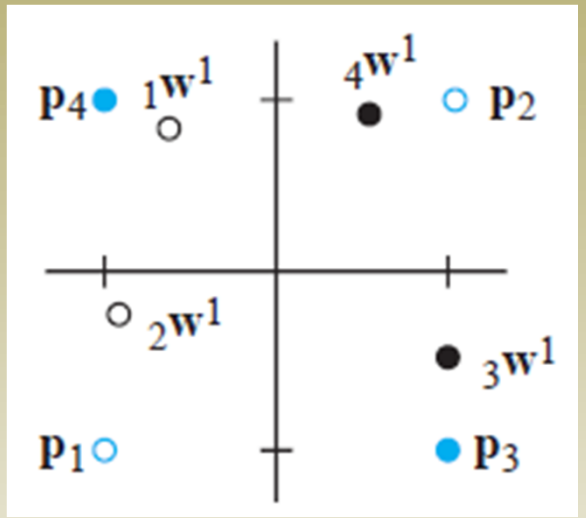
- This output indicates that $\mathbf{p}_3$ is a member of class 2. This is correct, so ${}_3\mathbf{w}^1$ is updated by moving it toward $\mathbf{p}_3$

$${}_3\mathbf{w}^1(1) = {}_3\mathbf{w}^1(0) + \alpha(\mathbf{p}_3 - {}_3\mathbf{w}^1(0))$$

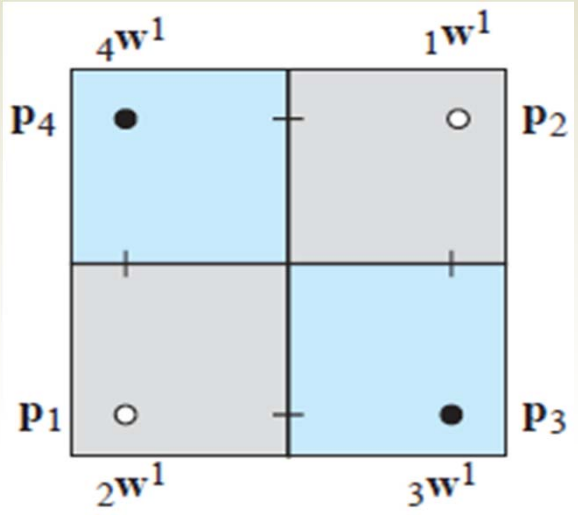
$$= \begin{bmatrix} 0.997 \\ 0.094 \end{bmatrix} + 0.5\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} - \begin{bmatrix} 0.997 \\ 0.094 \end{bmatrix} \right) = \begin{bmatrix} 0.998 \\ -0.453 \end{bmatrix}$$

LVQ Learning: Example

- The figure on the right shows the weights after ${}_3\mathbf{w}^1$ was updated on the first iteration



- The figure on the right shows the weights after the algorithm has converged. The figure also indicates how the regions of the input space will be classified. The regions that will be classified as class 1 are shown in gray, and the regions that will be classified as class 2 are shown in blue



Improving LVQ Networks

- The LVQ network suffers from a couple of limitations:
  - First, as with competitive layers, occasionally a hidden neuron in an LVQ network can have initial weight values that stop it from ever winning the competition. The result is a dead neuron that never does anything useful
    - This problem is solved with the use of a "conscience" mechanism (will see it as an exercise in class)
  - Secondly, depending on how the initial weight vectors are arranged, a neuron's weight vector may have to travel through a region of a class that it doesn't represent, to get to a region that it does represent
    - Because the weights of such a neuron will be repulsed by vectors in the region it must cross, it may not be able to cross, and so it may never properly classify the region it is being attracted to
    - This is usually solved by applying the following modification to the Kohonen rule

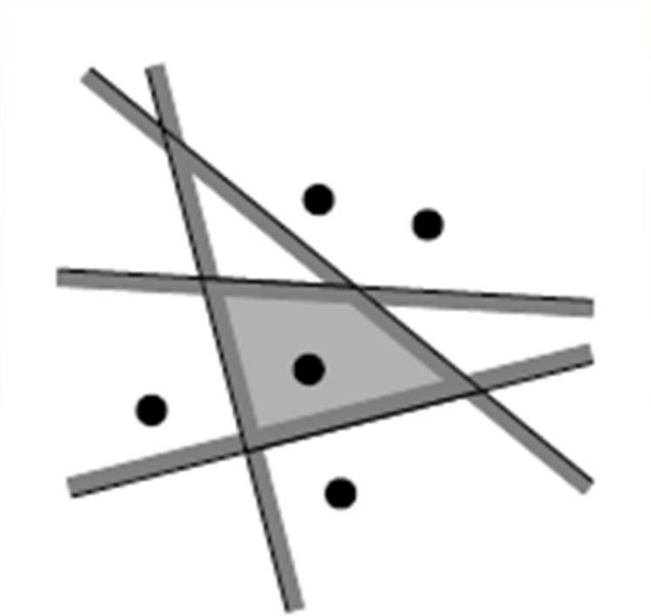
Improving LVQ Networks: The LVQ2

- If the winning neuron in the hidden layer incorrectly classifies the current input, we move its weight vector away from the input vector, as before
- However, we also adjust the weights of the closest neuron to the input vector that does classify it properly
  - The weights for this second neuron should be moved toward the input vector
- When the network correctly classifies an input vector, the weights of only one neuron are moved toward the input vector. However, if the input vector is incorrectly classified, the weights of two neurons are updated, one weight vector is moved away from the input vector, and the other one is moved toward the input vector
- The resulting algorithm is called *LVQ2*

Appendix:
The Voronoi diagram

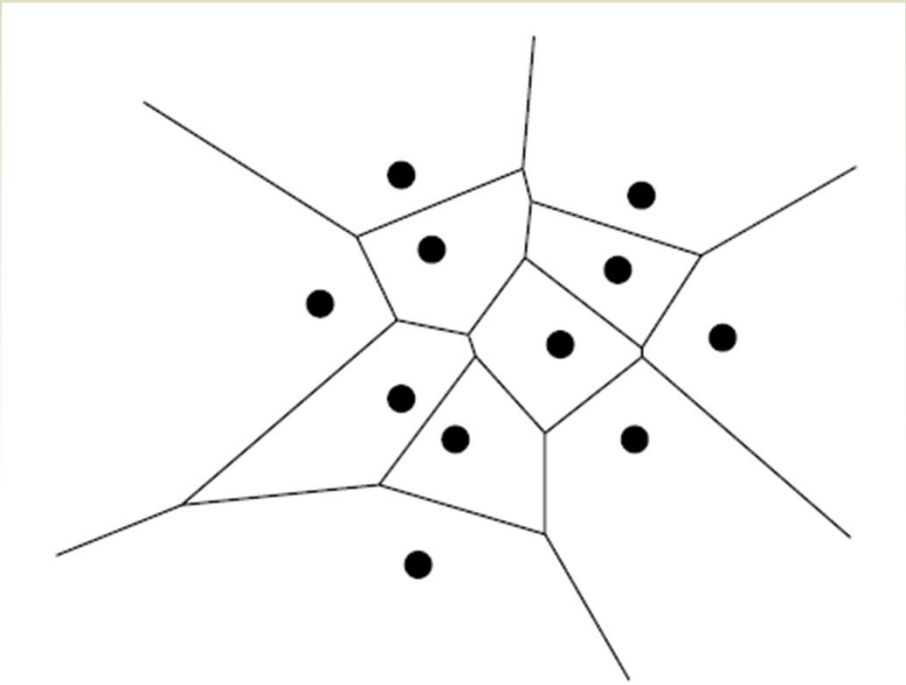
Definition of the Voronoi diagram

- The Post Office problem
- We are given a set of n points in the plane
- We define the Voronoi diagram of P as the subdivision of the plane into n cells, one for each site in P, with the property that a point q lies in the cell corresponding to a site p_i if and only if dist(q, p_i) < dist(q, p_j) for each $p_j \in$ P with j ≠ i



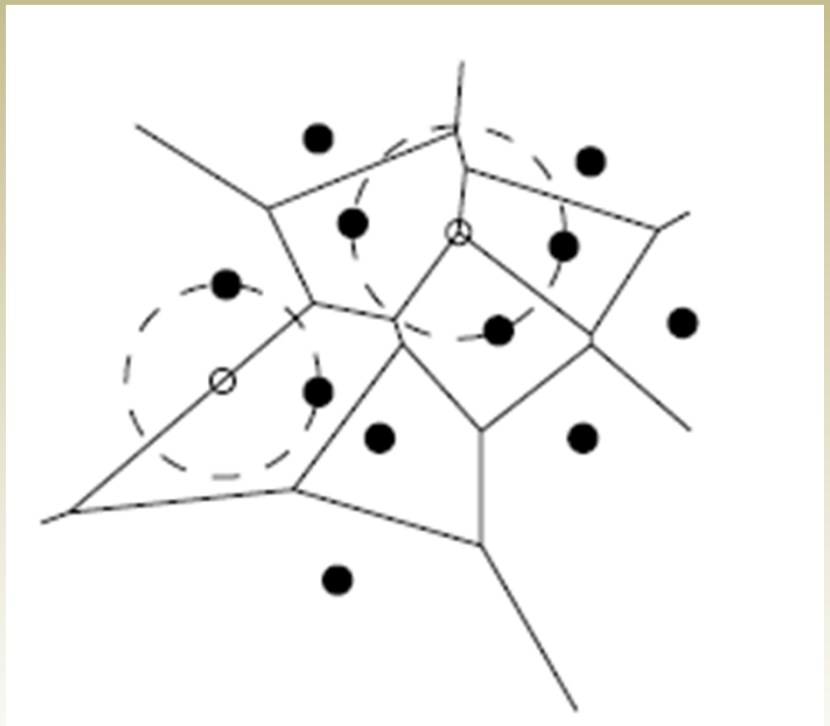
Structure of the Voronoi diagram

- What does the complete Voronoi diagram look like?
  - Each cell of the diagram is the intersection of a number of half-planes, so the Voronoi diagram is a planar subdivision whose edges are straight
  - Some edges are line segments and others are half-lines
  - Unless all sites are collinear there will be no edges that are full lines



Structure of the Voronoi diagram

- **Theorem**. For the Voronoi diagram Vor(P) of a set of points P the following holds:
  i. A point q is a vertex of Vor(P) if and only if its largest empty circle $C_P(q)$ contains three or more sites on its boundary.
  ii. The bisector between sites p_i and p_j defines an edge of Vor(P) if and only if there is a point q on the bisector such that $C_P(q)$ contains both p_i and p_j on its boundary but no other site



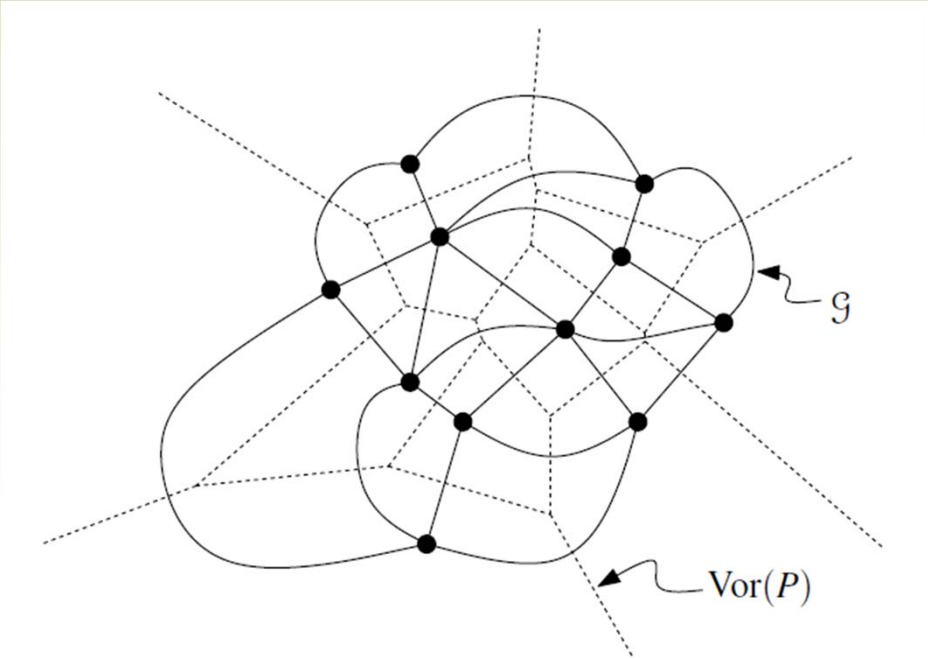
Computation of the Voronoi diagram

i. *Fortune*'s algorithm after its inventor—computes the Voronoi diagram in O(nlogn) time
ii. You may be tempted to look for an even faster algorithm, for example one that runs in linear time
iii. The problem of sorting n real numbers is reducible to the problem of computing Voronoi diagrams, so any algorithm for computing Voronoi diagrams must take Ω(nlogn) time in the worst case
iv. Hence, Fortune's algorithm is optimal

Appendix:
The Delaunay Graph

The dual of the Voronoi diagram

- The dual graph G of the Voronoi diagram
- This graph G has a node for every Voronoi cell—equivalently, for every site—and it has an arc between two nodes if the corresponding cells share an edge. Note that this means that G has an arc for every edge of Vor(P)
- As you can see, there is a one-to-one correspondence between the bounded faces of G and the vertices of Vor(P).



Definition of the Delaunay Graph

- Consider the straight-line embedding of *G*, where the node corresponding to the Voronoi cell V(p) is the point p, and the arc connecting the nodes of V(p) and V(q) is the segment pq—see Figure below
- We call this embedding the *Delaunay graph* of P, and we denote it by *DG*(P)