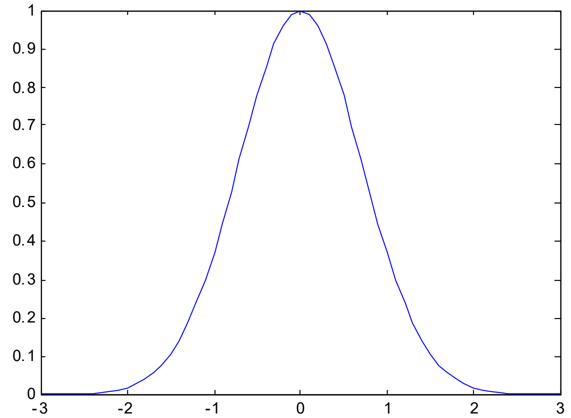
Artificial Neural Networks
Part II

Beatrice Lazzerini
Department of Information Engineering
University of Pisa
ITALY

Radial functions

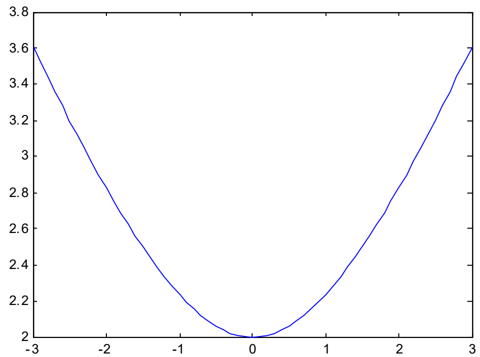
- *Radial functions*: their response decreases (or increases) monotonically with distance from a central point.
- E.g., Gaussian

$$h(x) = exp\left(-\frac{(x-t)^2}{2\sigma^2}\right)$$



- Its parameters are its *center* t and its *radius* (or *spread*) σ
- Small spread ➔ very selective
- Large spread ➔ not very selective

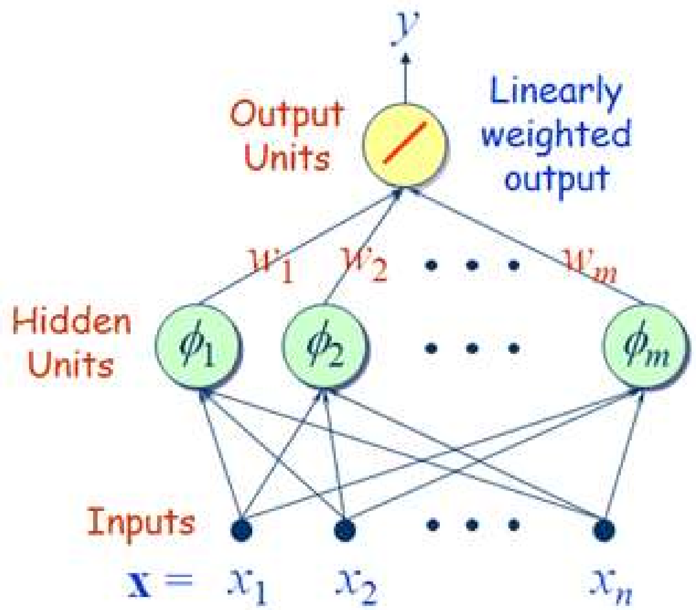
- Examples of RBFs
  - Multiquadric $\varphi(r) = (r^2 + c^2)^{1/2}$ $\quad c > 0$, $r \in R$



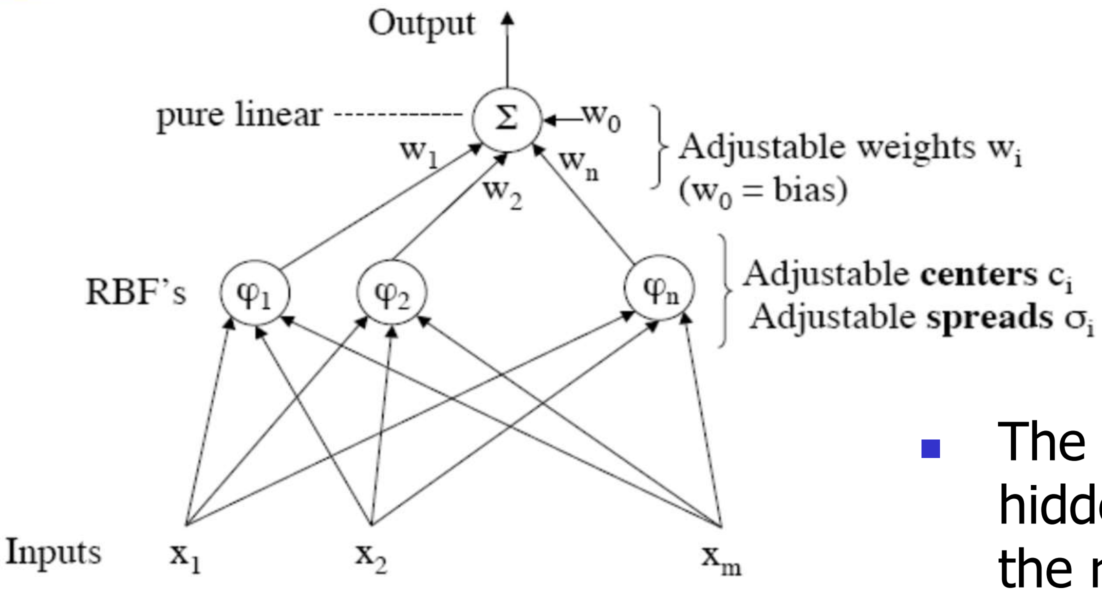
  - Inverse multiquadric $\varphi(r) = (r^2 + c^2)^{-1/2}$
  - Gaussian $\varphi(r) = exp\left(-\dfrac{r^2}{2\sigma^2}\right)$ $\quad \sigma > 0$

Radial Basis Function (RBF) network

- An RBF network in its simplest form is a two-layer network:
  - the input layer to hidden layer mapping is **nonlinear**,
  - the hidden layer to output layer mapping is **linear**.
- Usually (but not always) # hidden units > # input units.
- A nonlinear problem cast into a high-dimensional space nonlinearly is more likely to be linearly separable.



RBF network

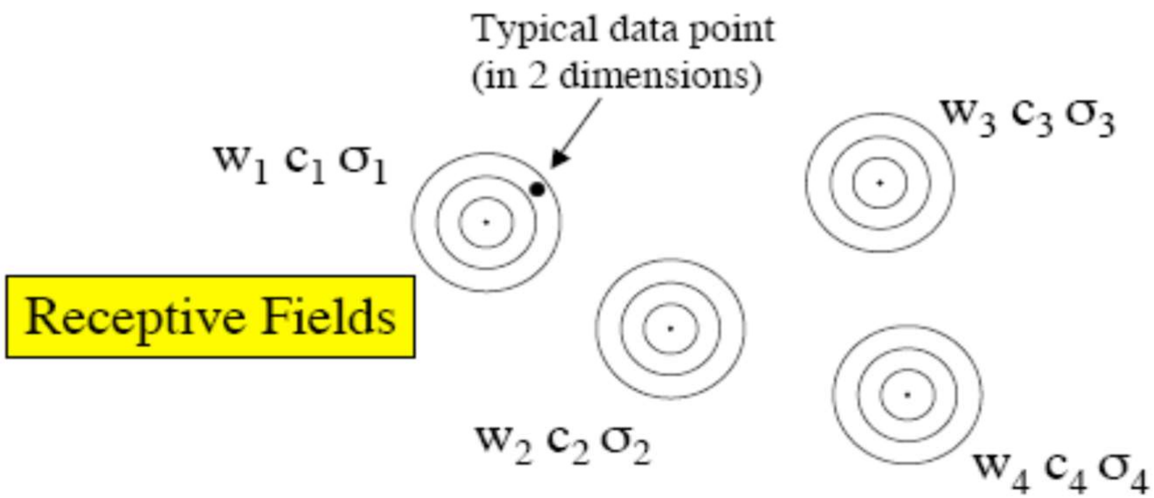


- The weights connecting to a hidden unit define the center of the radial-basis function for that hidden unit

- The net input to a hidden unit is the Euclidean norm

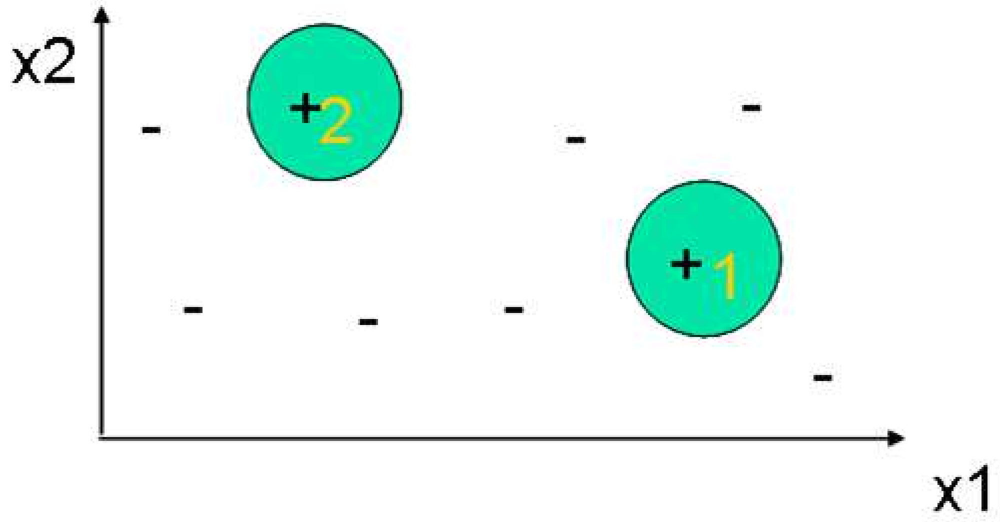
$$net_j = \left\| \mathbf{x} - \mathbf{w}_j \right\| = \left[\sum_{i=1}^{m} (x_i - w_{ij})^2 \right]^{1/2}$$

Output = $\Sigma\, w_i\, \varphi_i(\mathbf{x})$ where $\mathbf{x}$ is the input vector



- A function is approximated as a linear combination of radial-basis functions.
- An RBF responds only to a small region of the input space where the function itself is centered.
- An RBF network can be used not only for function fitting but also for classification.

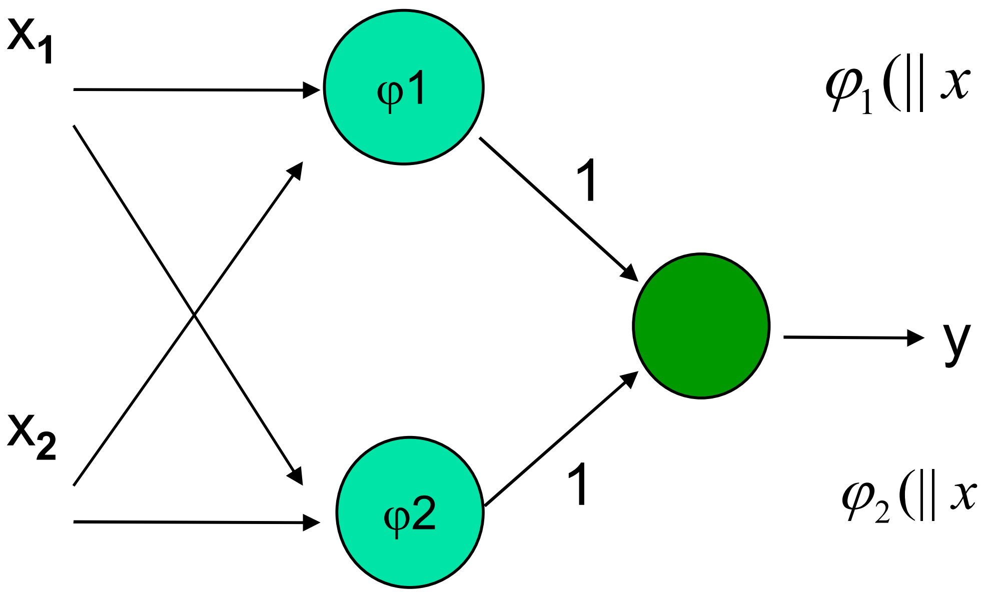
Example 1



- Examples inside circles 1 and 2 are of class +, examples outside both circles are of class –
- Can we separate the two classes using RBF networks?

Example 1: a solution

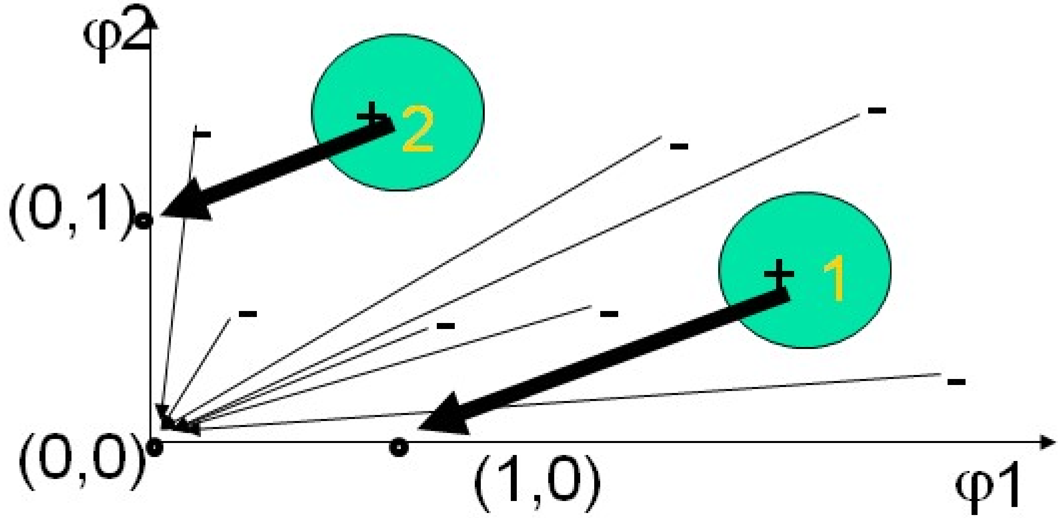
Choose as centers t1, t2 the centers of the two circles.
Let r1, r2 be the radii of the two circles, and x = (x1,x2) an example.



$$\varphi_1(\| x - t_1 \|) = \begin{cases} 1 \text{ if } \| x - t_1 \| <= r1 \\ 0 \text{ if } \| x - t_1 \| > r1 \end{cases}$$

$$\varphi_2(\| x - t_2 \|) = \begin{cases} 1 \text{ if } \| x - t_2 \| <= r2 \\ 0 \text{ if } \| x - t_2 \| > r2 \end{cases}$$

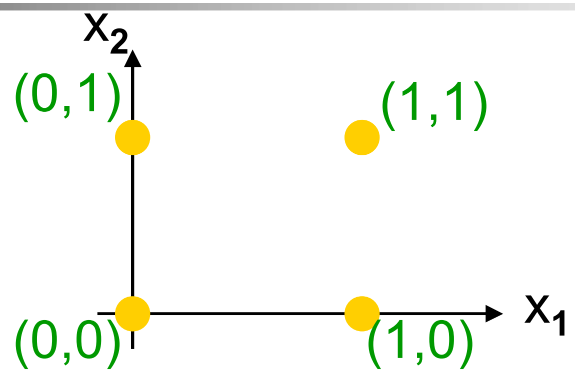
Example 1: a solution



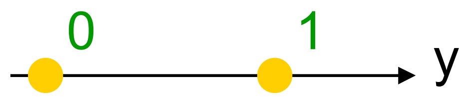
- Geometrically: examples are mapped from the input space <x1,x2> to the feature space $<\varphi_1,\varphi_2>$:
- examples in circle 1 are mapped to (1,0),
- examples in circle 2 are mapped to (0,1),
- examples outside both circles are mapped to (0,0).
- The two classes become linearly separable in the $<\varphi_1,\varphi_2>$ feature space.

Example 2: the XOR problem

- Input space:



- Output space:



- Construct an RBF pattern classifier with Gaussian activation functions such that:

  (0,0) and (1,1) are mapped to 0, class C1

  (1,0) and (0,1) are mapped to 1, class C2

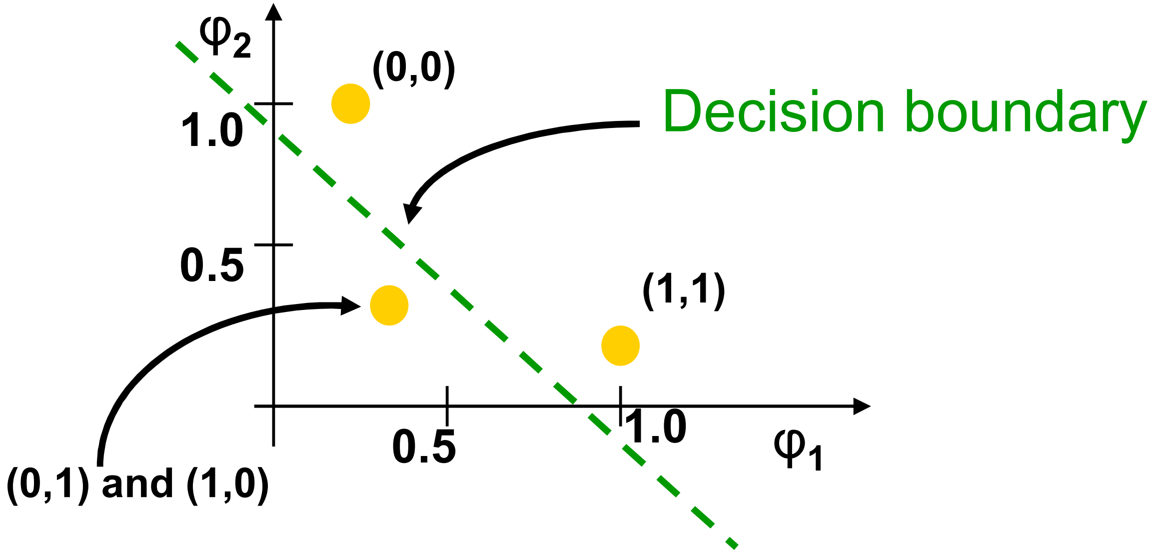
Example 2: a solution

We choose centers $t_1 = (1,1) \quad t_2 = (0,0)$

$$\varphi_1(\| x - t_1 \|) = e^{-\|x-t_1\|^2}$$
$$\varphi_2(\| x - t_2 \|) = e^{-\|x-t_2\|^2}$$

Input x	$\varphi_1(x)$	$\varphi_2(x)$
(1,1)	1	0.1353
(0,1)	0.3678	0.3678
(1,0)	0.3678	0.3678
(0,0)	0.1353	1

Example 2: a solution



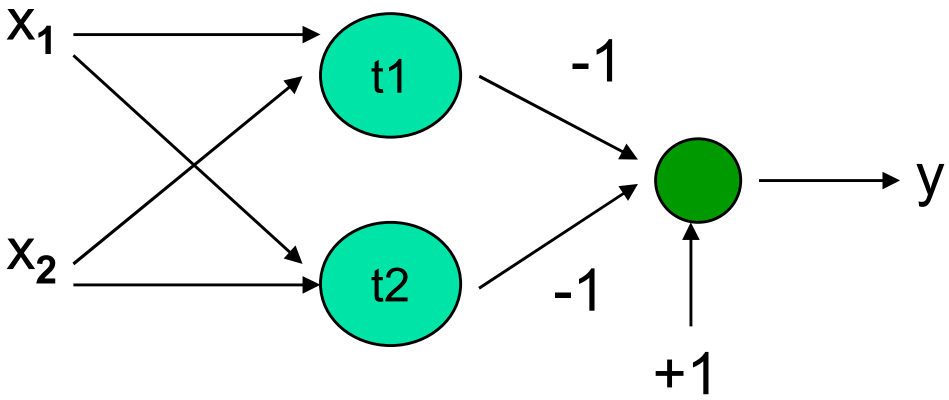
- When mapped into the feature space (hidden layer), C1 and C2 become linearly separable.
- So a linear classifier with $\varphi_1(x)$ and $\varphi_2(x)$ as inputs can be used to solve the XOR problem.

Example 2: a solution

$$\varphi_1(\| x - t_1 \|) = e^{-\|x-t_1\|^2}$$

with $t_1 = (1,1)$ and $t_2 = (0,0)$

$$\varphi_2(\| x - t_2 \|) = e^{-\|x-t_2\|^2}$$



$$y = -e^{-\|x-t_1\|^2} - e^{-\|x-t_2\|^2} + 1$$

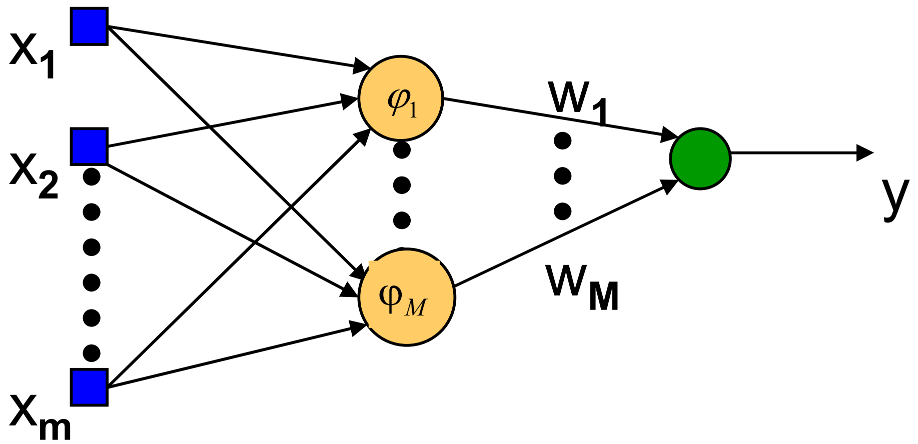
If $y > 0$ then class 1 otherwise class 0

Determination of parameters

- What should we learn for an RBF network with a given architecture and Gaussian activation functions?
  - The **centers** of the radial-basis functions.
  - The **spreads** of the radial-basis functions.
  - The **weights** from the hidden layer to the output layer.

Learning strategies

- The different layers of an RBF network perform different tasks, so it is reasonable to separate the optimization of the hidden and output layers of the network using different techniques.
- There are different learning strategies depending on how the centers of the radial-basis functions are specified.
- Basically, we can identify three approaches.



Learning Algorithm 1
Fixed centers selected at random

- The locations of the **centers** can be *randomly* chosen from the training data set.
- The **spreads** are chosen by *normalization*. More precisely, the standard deviation of the Gaussian functions is fixed by normalization according to the spread of the centers

$$\sigma = \frac{d}{\sqrt{2M}}$$

where M is the number of centers and d is the maximum distance between the chosen centers.

- Such a choice for the standard deviation simply ensures that the Gaussian functions are not too peaked or too flat; both of these extremes must be avoided.

Learning Algorithm 1

- The activation function for hidden neurons becomes:

$$\varphi_i\left(\left\|\mathbf{x}-\mathbf{t}_i\right\|\right)=\exp\left(-\frac{M}{d^2}\left\|\mathbf{x}-\mathbf{t}_i\right\|^2\right), \qquad i=1,\dots,M$$

- The linear **weights** in the output layer are computed by means of the *pseudo-inverse method*.
  - For an example (x_i, d_i) consider the output of the network

$$y(x_i)=w_1\varphi_1(\| x_i - t_1 \|)+\ldots+w_M\varphi_M(\| x_i - t_M \|)$$

  - We would like $y(x_i)=d_i$ for each example, that is

$$w_1\varphi_1(\| x_i - t_1 \|)+\ldots+w_M\varphi_M(\| x_i - t_M \|)=d_i$$

Learning Algorithm 1

- This can be re-written in matrix form for one example

$$\left[\varphi_1(\| x_i - t_1 \|) \;\ldots\; \varphi_M(\| x_i - t_M \|)\right][w_1 \ldots w_M]^T = d_i$$

and

$$\begin{bmatrix} \varphi_1(\| x_1 - t_1 \|) \ldots \varphi_M(\| x_1 - t_M \|) \\ \ldots \\ \varphi_1(\| x_N - t_1 \|) \ldots \varphi_M(\| x_N - t_M \|) \end{bmatrix} [w_1 \ldots w_M]^T = [d_1 \ldots d_N]^T$$

for all N examples.

Learning Algorithm 1

- Let

$$\Phi = \begin{bmatrix} \varphi_1(\| x_1 - t_1 \|) & \dots & \varphi_M(\| x_1 - t_M \|) \\ & \dots & \\ \varphi_1(\| x_N - t_1 \|) & \dots & \varphi_M(\| x_N - t_M \|) \end{bmatrix}$$

- Then we can write

$$\Phi \begin{bmatrix} w_1 \\ \dots \\ w_M \end{bmatrix} = \begin{bmatrix} d_1 \\ \dots \\ d_N \end{bmatrix}$$

- If Φ^+ is the pseudo-inverse of matrix Φ we obtain the weights using the following formula

$$[w_1 \dots w_M]^T = \Phi^+ [d_1 \dots d_N]^T$$

Learning Algorithm 1: summary

- Choose the **centers** randomly from the training set.
- Calculate the **spread** for the radial-basis functions using normalization.
- Find the **weights** using the pseudo-inverse method.

Learning Algorithm 2
Self-organized selection of centers

- A clustering algorithm is used to find the **centers**. This allows you to place the centers of the radial-basis functions in those regions of the input space where significant data are present.
- **Spreads** are chosen by normalization.
- A supervised learning rule (e.g., *least mean square* (LMS) algorithm) is applied to calculate the linear **weights** of the output layer. The outputs of the hidden units in the RBF network serve as the inputs of the LMS algorithm.

Learning Algorithm 3
Supervised selection of centers

- In the third approach, update formulas are used to simultaneously train weights, centers and spreads iteratively using gradient descent.

- In practice, the squared error $\mathrm{E} = \frac{1}{2}\sum_{j=1}^{N} e_j^2$ is minimized,

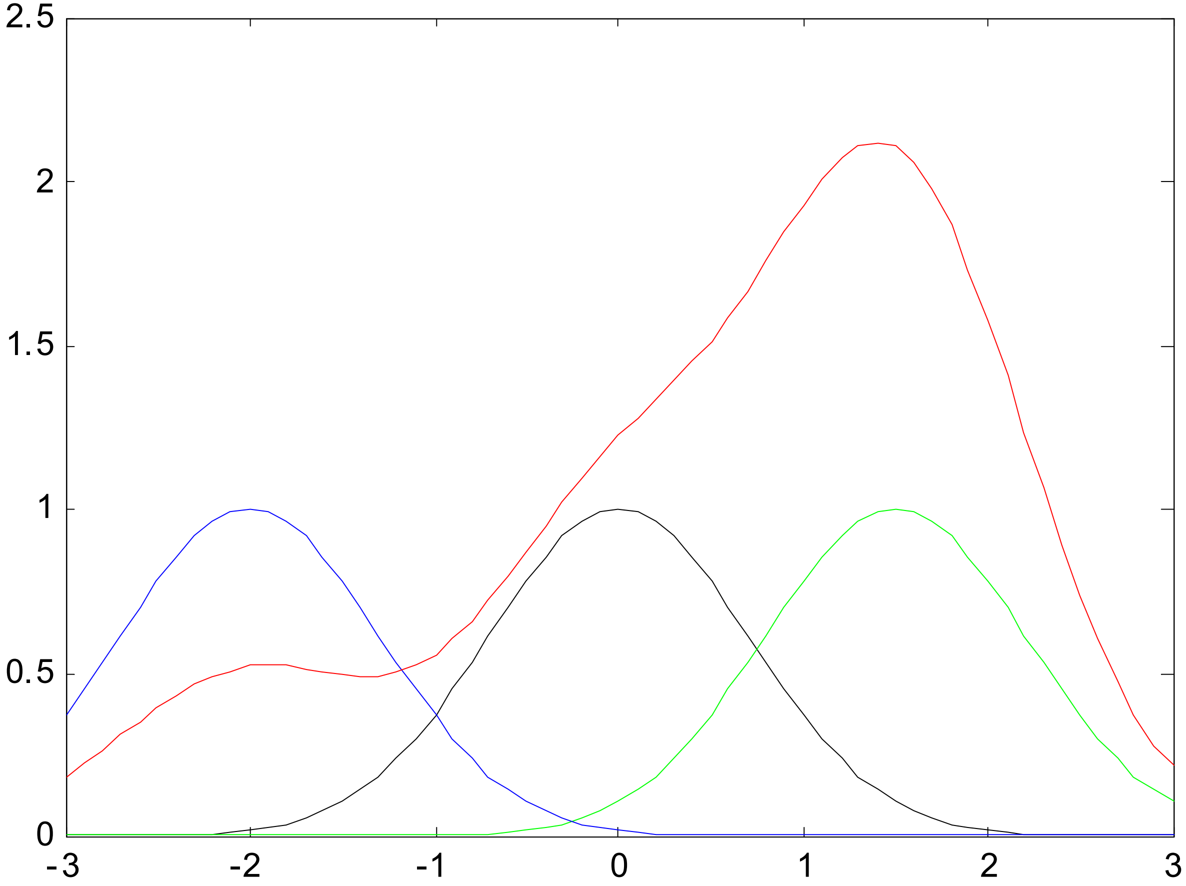
  where N is the number of training examples and e_j is the error, defined by

$$e_j = d_j - y(x_j) = d_j - \sum_{i=1}^{M} w_i \varphi_i\left(\left\| x_j - t_i \right\|\right)$$

- The results of the minimization of the error are the updates for
  - centers $$\Delta t_j = -\eta_{t_j} \frac{\partial E}{\partial t_j}$$
  - spread $$\Delta \sigma_j = -\eta_{\sigma_j} \frac{\partial E}{\partial \sigma_j}$$
  - weights $$\Delta w_{ij} = -\eta_{ij} \frac{\partial E}{\partial w_{ij}}$$

- p=[-3:.1:3];
- f1=radbas(p);
- f2=radbas(p-1.5);
- f3=radbas(p+2);
- f=f1+f2*2+f3*0.5;

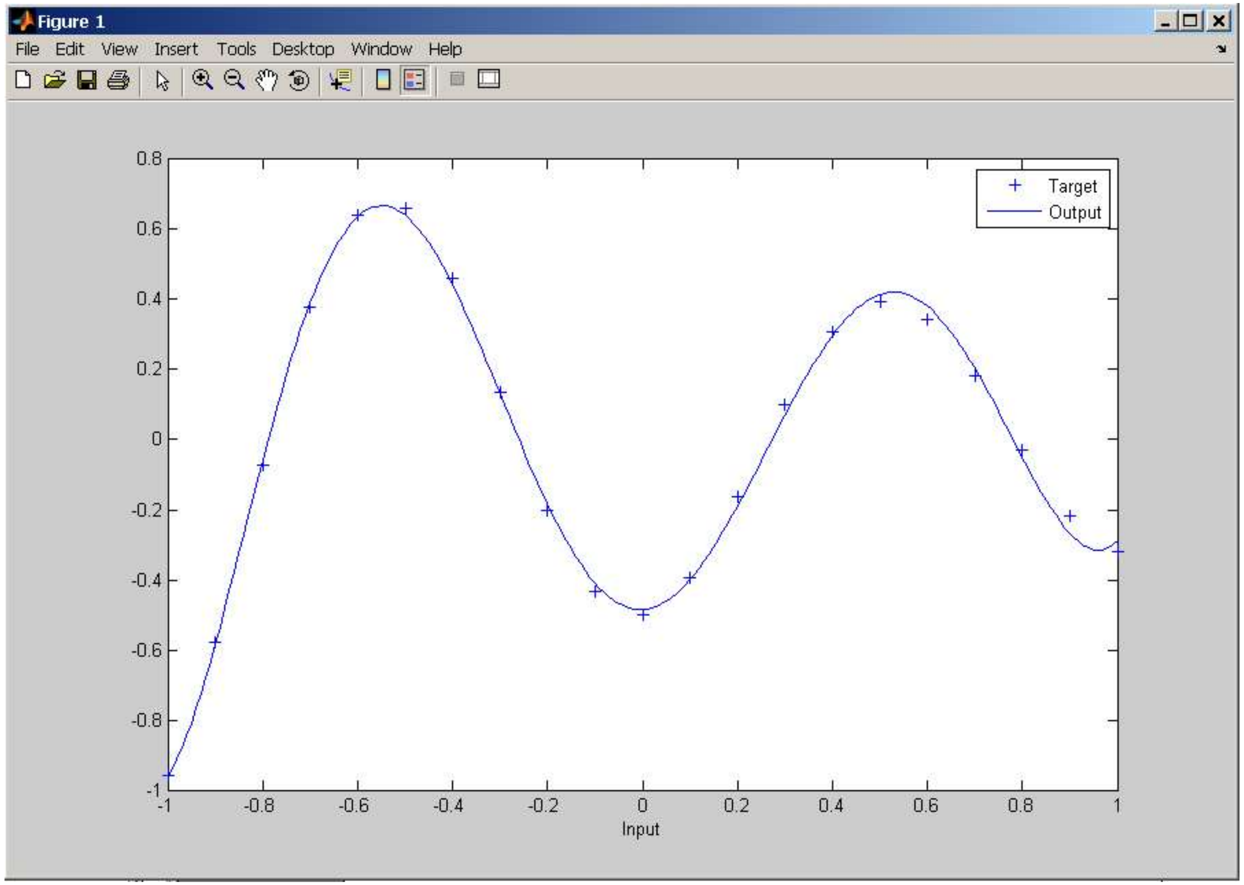
$$radbas(n) = e^{-n^2}$$



Example: RBF network for function approximation

- An RBF network that approximates a function defined by 21 data points (-1:.1:1)

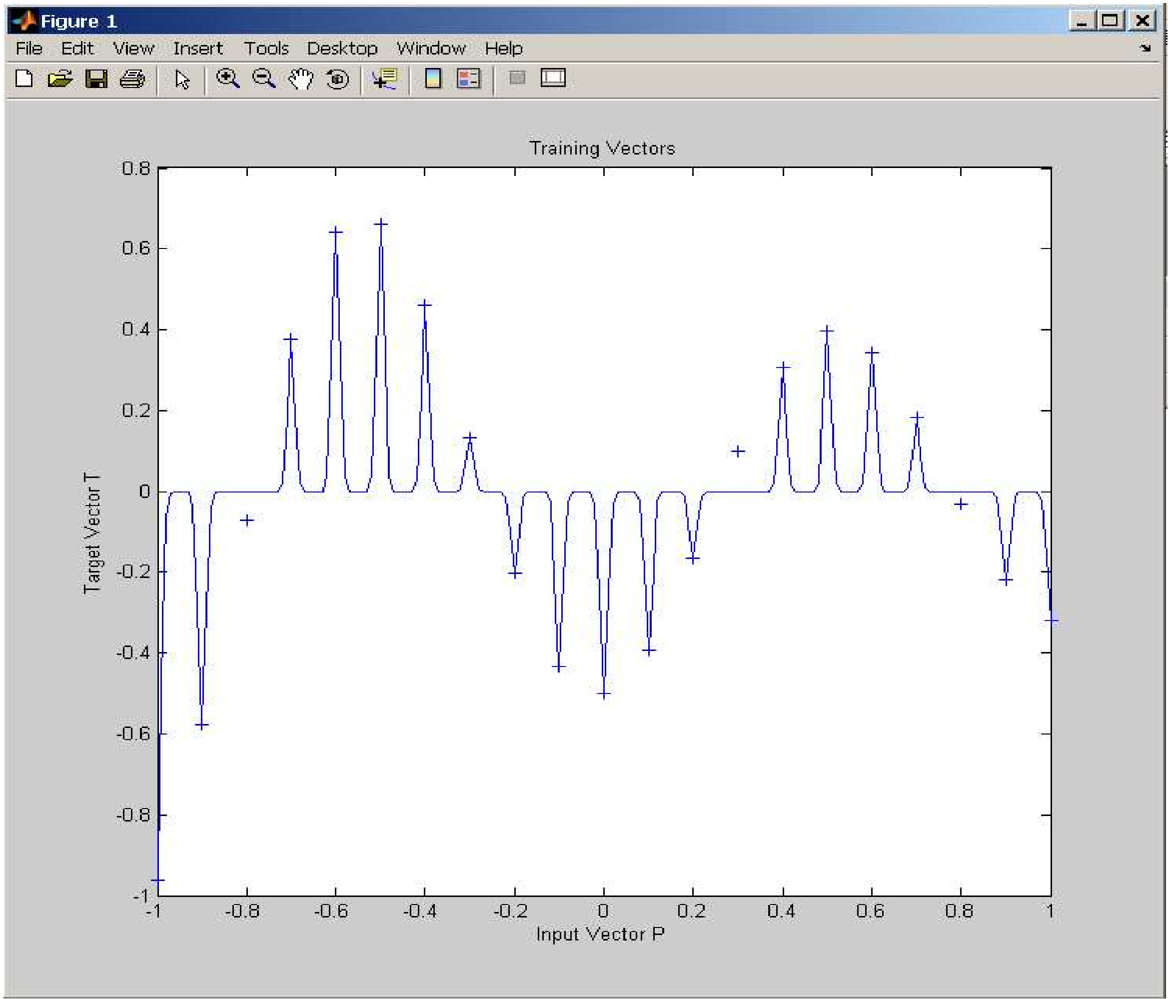
- **<u>First case</u>**
- spread constant = 1



Example: RBF network for function approximation

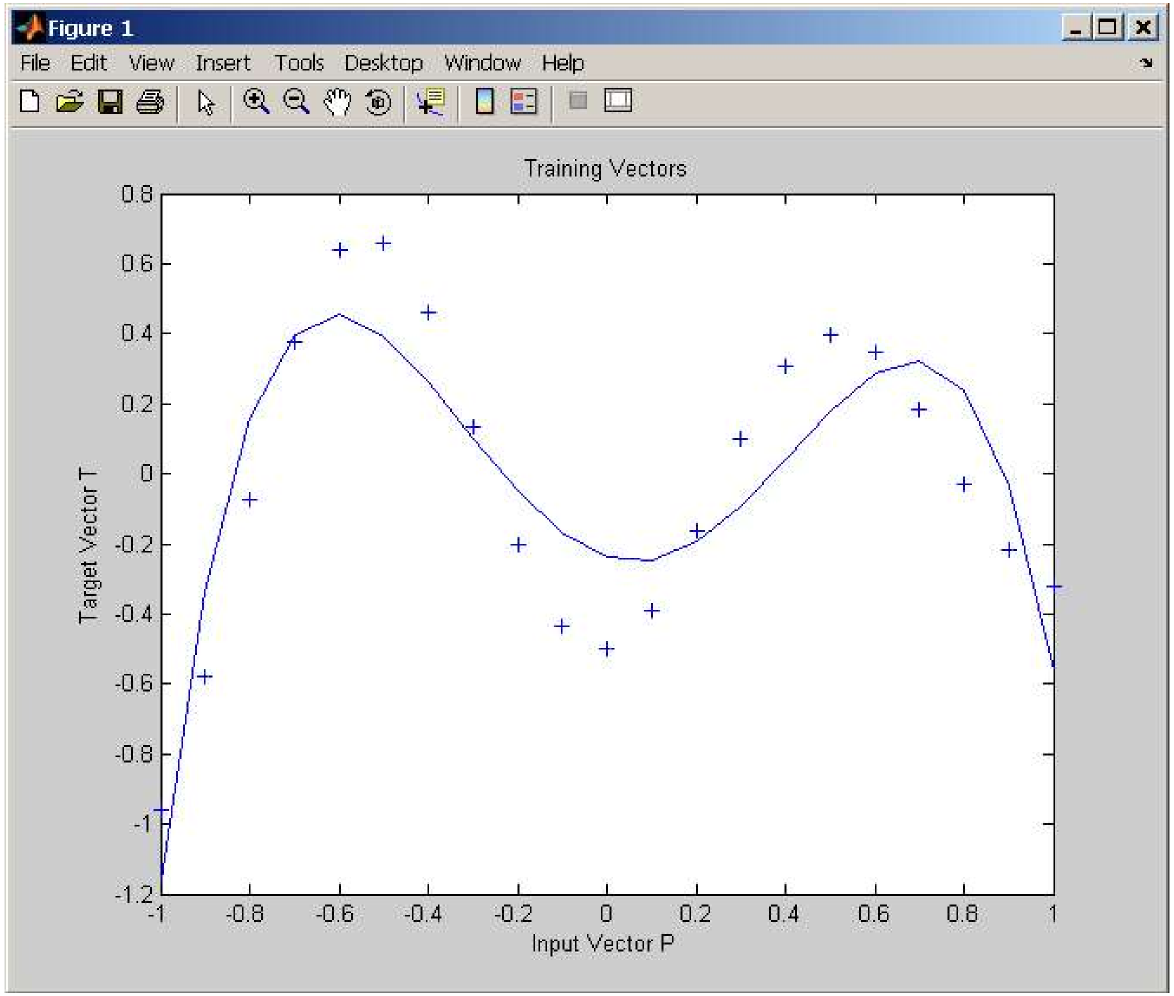
- **<u>Second case</u>**
- spread constant = .01 ➔ there are no two RBF neurons that have strong output for any given input.
- In other words, the solution does not generalize from the input/output vectors

  ➔ **overfitting**



Example: RBF network for function approximation

- **<u>Third case</u>**
- spread constant =100 ➔ the RBF neurons will output high values (≈ 1) for all inputs used to design the network.
- All neurons always output 1, so they cannot be used to generate different responses.



The moral of the story

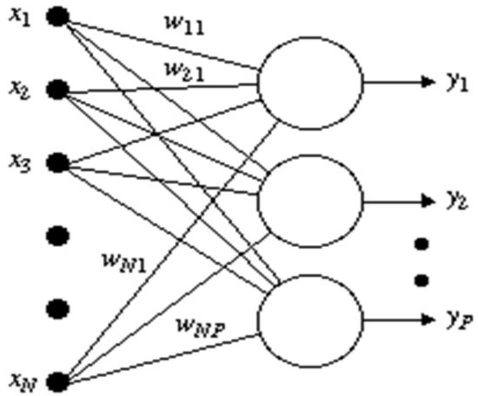
- Choose a spread larger than the distance between adjacent input vectors, so that you get a good generalization, but smaller than the distance across the entire input space.

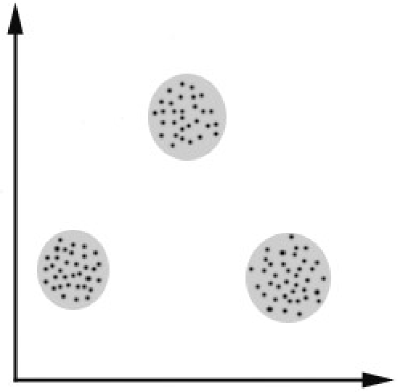
RBF Networks and MLPs

- RBF networks and MLPs are examples of nonlinear layered feedforward networks. They are both ***universal approximators*** ➔ there is always an RBF network that can accurately mimic a specified MLP, or vice versa. However, these two networks differ from each other. In particular,
  - MLPs construct *global* approximations to nonlinear input-output mapping. Consequently, they are able to generalize in regions of the input space where few or no training data are available.
  - RBF networks construct *local* approximations to nonlinear input-output mapping, with the result that these networks are able to learn quickly and have reduced sensitivity to the order of presentation of the training data.

Supervised and Unsupervised learning

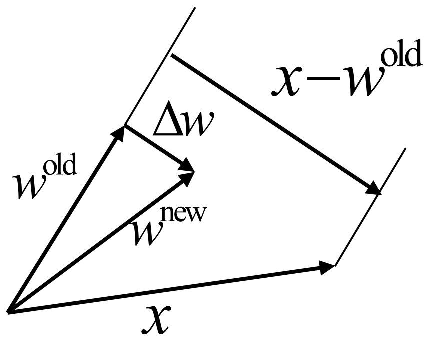
- **Supervised learning** is used when an external teacher provides a set of training examples of proper network behavior. The network learns how to map an input pattern to a desired pattern.
- **Unsupervised learning** is mainly based on *clustering* of input data. No prior knowledge is available regarding whether an input belongs to a particular class. The network, which is trained without a teacher, learns by detecting the similarity between the input patterns. Each cluster is represented by its *centroid* (the position that is the average of all points in the cluster). These centroids can be considered as *prototypes* for the clusters (as they represent the key features of a cluster).





- Examples of unsupervised neural networks: *competitive network* and *self-organizing map* (*SOM*), also called *self-organizing feature map* (*SOFM*).
- The number of input neurons matches the dimension of the input vectors.
- The output neurons (or *cluster units*) act as cluster prototypes: the *P* output neurons classify the input data into *P* clusters.
- Each input unit is fully connected to each output unit → the total number of weights connected to an output neuron equals the size of the input vectors. The weights of a cluster unit are considered to be the coordinates that describe the cluster's position in the input space.

- The basic form of unsupervised learning is called *competitive learning*: all units compete to be the one to fire: the unit closest (i.e., most similar) to the training vector is the *winning unit* or the *winner* or the *winner-takes-all neuron*.
- Two common similarity measures:
  - Euclidean distance
  - Cosine distance
- The weights of the winning neuron are adapted to move the neuron even closer to the training vector.



$$w^{new} = w^{old} + \Delta w = w^{old} + \alpha (x - w^{old}), \quad \alpha \in (0,1)$$

➔ *Competitive network*

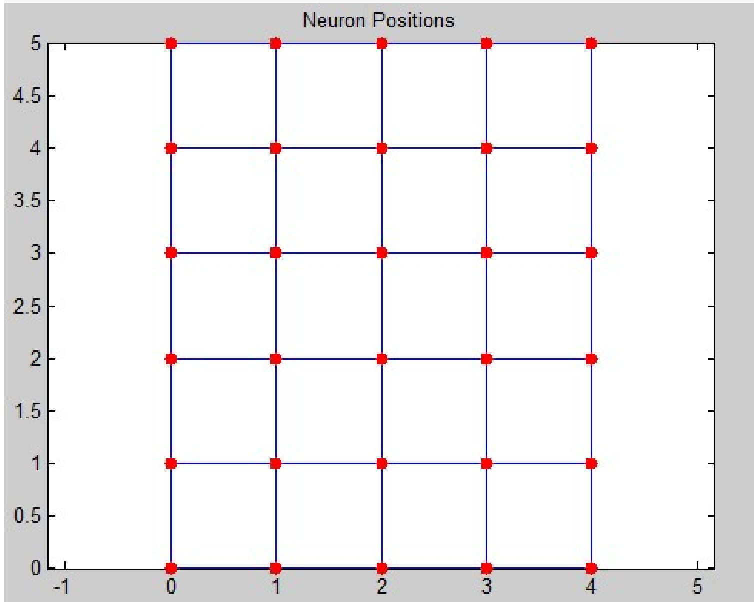
- Usually, the units within a given neighborhood of the winning neuron also update their weights.
- A neuron is a member of the updating neighborhood if it falls within a given radius that is centered on the winning neuron.
- The radius of the neighborhood is typically reduced during training.
- A learning rate determines the amount by which a neuron moves towards the training vector and, like the radius, is gradually reduced over time.

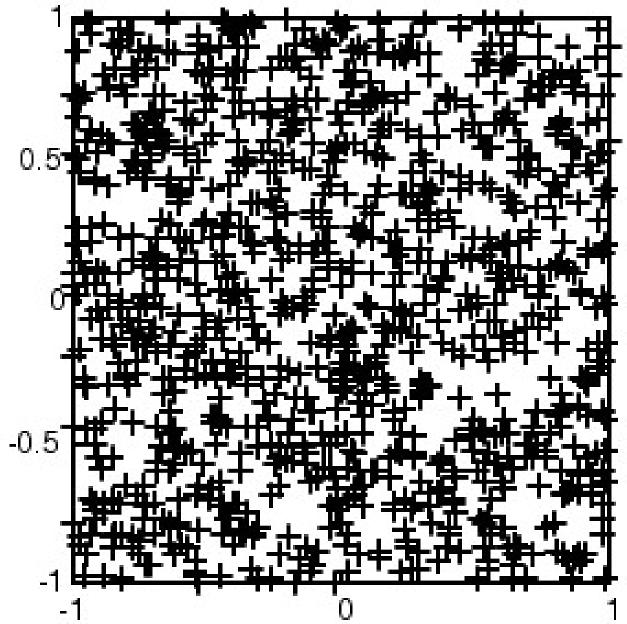
→ *SOM*

- Typically, the cluster units are arranged as a linear array or as a square grid. Other topologies can be used, e.g., triangular or hexagonal. The topology simply controls which units for a given radius need to be updated.
- At the end of the training, the cluster units provide a summary representation of the input space.
- SOMs learn both the *distribution* (as competitive layers do) and *topology* of the input vectors they are trained on.

Example

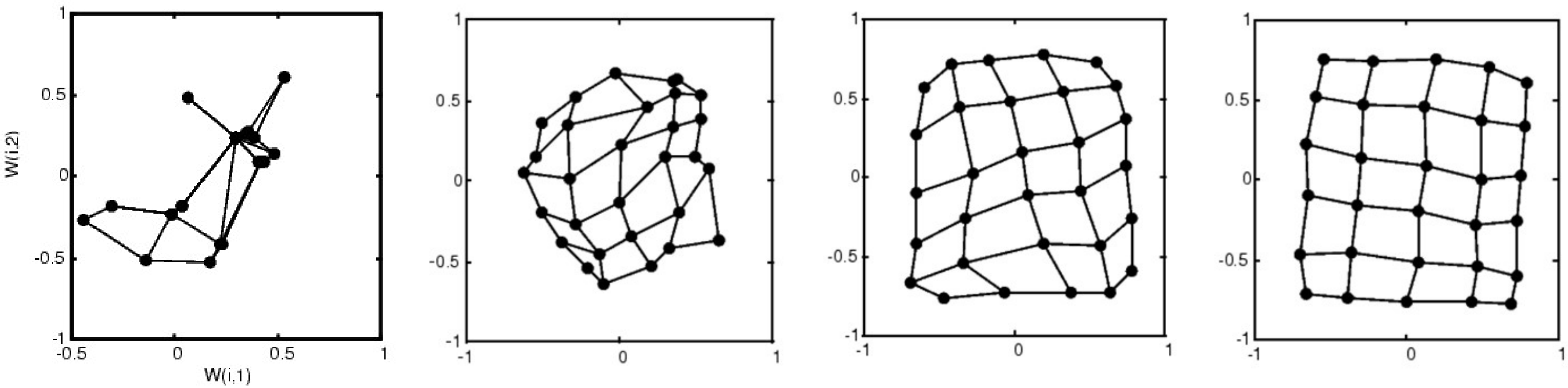
- Let us consider a map (from MATLAB) with a square topology of 30 neurons (5x6). The map is trained on 1000 data points drawn from a given square:





- Let us look at the development of the map over time: the map is drawn at different times during training by plotting the cluster units in the input space.

- The map is trained for 5000 presentation cycles (epochs)



after 40 cycles after 120 cycles after 500 cycles after 5000 cycles