



Αναγνώριση Προτύπων

Ταξινομητές Νευρωνικών Δικτύων

Χριστόδουλος Χαμζάς

2011

Τμήμα του περιεχομένου των παρουσιάσεων προέρχονται από τις παρουσιάσεις του αντίστοιχου διδασκτέου μαθήματος του καθ. Σέργιο Θεοδωρίδη,, Τμ. Πληροφορικής και Τηλεπικοινωνιών, Πανεπιστήμιο Αθηνών

Γιατί μη γραμμικοί ταξινομητές;



- ❖ Πρόβλημα 2 κατηγοριών: Αν ο αριθμός των προτύπων είναι μικρότερος από τον αριθμό των συνιστωσών κάθε προτύπου, τότε υπάρχει πάντα υπερεπίπεδο που τα διαχωρίζει.
- ❖ Επομένως οι γραμμικοί ταξινομητές είναι χρήσιμοι:
 - σε προβλήματα πολύ μεγάλης διαστατικότητας
 - Σε προβλήματα μέτριας διαστατικότητας, όπου υπάρχει ένας σχετικά μικρός αριθμός προτύπων εκπαίδευσης
- ❖ Επιπλέον ο αριθμός των συνιστωσών ενός προτύπου μπορεί να αυξηθεί αυθαίρετα με την προσθήκη νέων συνιστωσών που είναι μη γραμμικές συναρτήσεις των αρχικών συνιστωσών (π.χ. πολυώνυμα)
- ❖ **Υπάρχουν πολλά προβλήματα που δεν μπορούν να επιλυθούν με γραμμικούς ταξινομητές**
- ❖ Στις προηγούμενες μεθόδους, η κύρια δυσκολία είναι η εύρεση της μη γραμμικής συνάρτησης
- ❖ Μια κατηγορία μη γραμμικών ταξινομητών είναι και τα πολυστρωματικά νευρωνικά δίκτυα
- ❖ Σε αυτά η μορφή της μη γραμμικής συνάρτησης διαχωρισμού **μαθαίνεται** από τα δεδομένα εκμάθησης

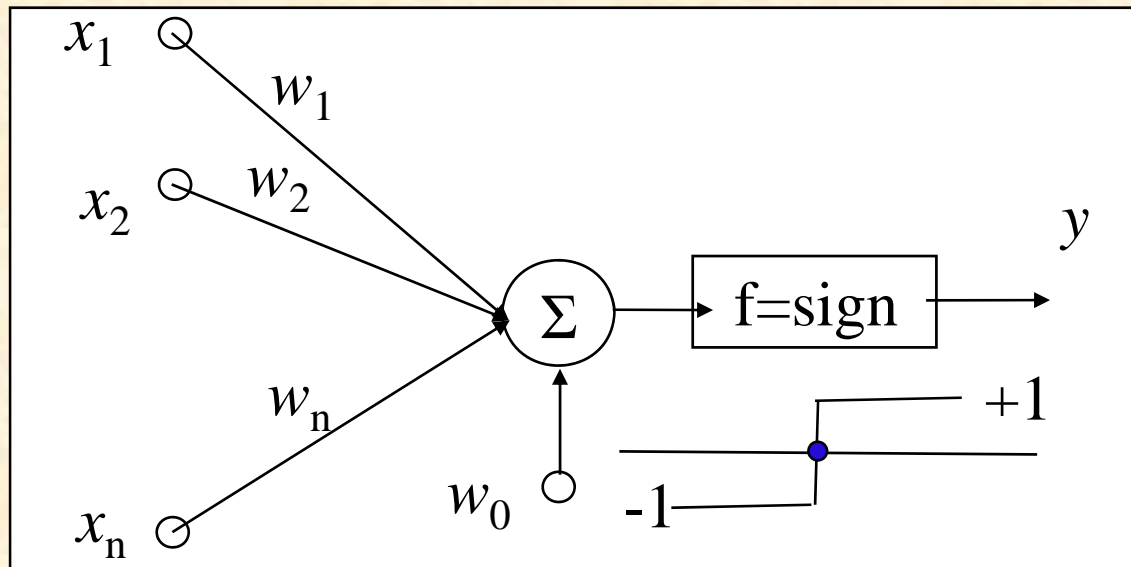


Το απλό perceptron (γραμμικός ταξινομητής)

- ❖ **Αρχιτεκτονική:** Μονοστρωματικό δίκτυο με N εισόδους και M νευρώνες διατεταγμένους σε ένα στρώμα νευρώνων. Συναπτικές συνδέσεις συνδέουν όλους τους νευρώνες με όλες τις εισόδους.
- ❖ **Νευρώνες:** Τύπου McCulloch-Pitts με hard limiter και προσαρμοζόμενο κατώφλι ενεργοποίησης

$$y_i = \text{sign}\left(\sum_j w_{ij} x_j - w_{0i}\right)$$

- ❖ Δεδομένου ότι οι έξοδοι είναι ανεξάρτητες μεταξύ τους, μπορούμε να τις μελετήσουμε και ανεξάρτητα, θεωρώντας κάθε νευρώνα του perceptron χωριστά:



$$y = \text{sign}\left(\sum_j w_j x_j - w_0\right)$$

$$= \text{sign}(\mathbf{w} \cdot \mathbf{x} - w_0)$$



Το πρόβλημα: Δίνεται ένα σύνολο P προτύπων

$$\{\mathbf{x}^\mu, \mu = 1, 2, \dots, P\} \subseteq \mathbb{R}^n$$

διαμερισμένο σε 2 κατηγορίες

$$C_1 = \{\mathbf{x}^\mu, \mu = 1, 2, \dots, K\} \quad C_2 = \{\mathbf{x}^\mu, \mu = K + 1, K + 2, \dots, P\}$$

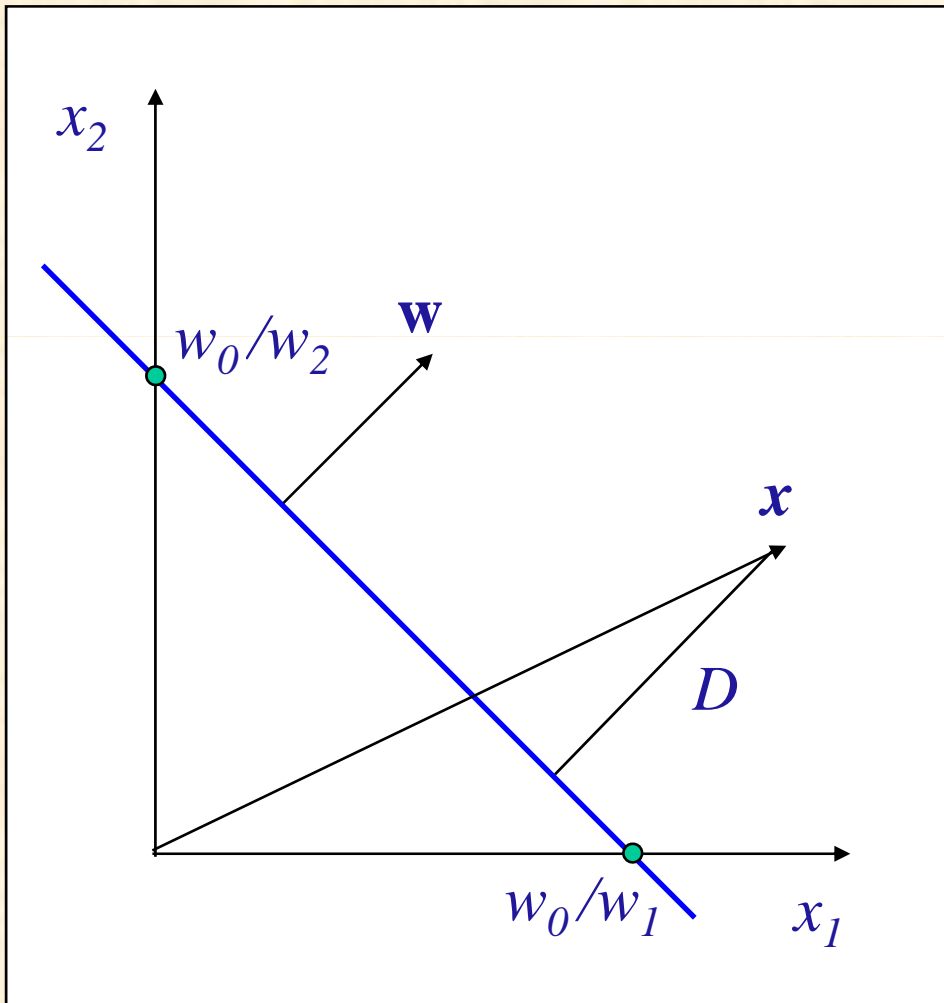
που είναι γραμμικά διαχωρίσιμες, δηλαδή υπάρχει ένα διάνυσμα $\hat{\mathbf{w}}$ ώστε

$$\hat{\mathbf{w}} \cdot \mathbf{x}^\mu > 0 \quad \forall \mathbf{x}^\mu \in C_1, \quad \hat{\mathbf{w}} \cdot \mathbf{x}^\mu < 0 \quad \forall \mathbf{x}^\mu \in C_2$$

Το ζητούμενο είναι να βρεθεί ένα τέτοιο διάνυσμα που να διαχωρίζει γραμμικά τις 2 κατηγορίες με επαναληπτικό τρόπο.



Η γεωμετρία του προβλήματος



$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$

- ❖ Διάνυσμα βαρών κάθετο στο διαχωριστικό υπερεπίπεδο
- ❖ Απόσταση προτύπου \mathbf{x} από το διαχωριστικό υπερεπίπεδο:

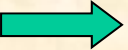

$$D = \frac{|g(\mathbf{x})|}{\|\mathbf{w}\|} = \frac{|\mathbf{w}^T \mathbf{x} + w_0|}{\|\mathbf{w}\|}$$

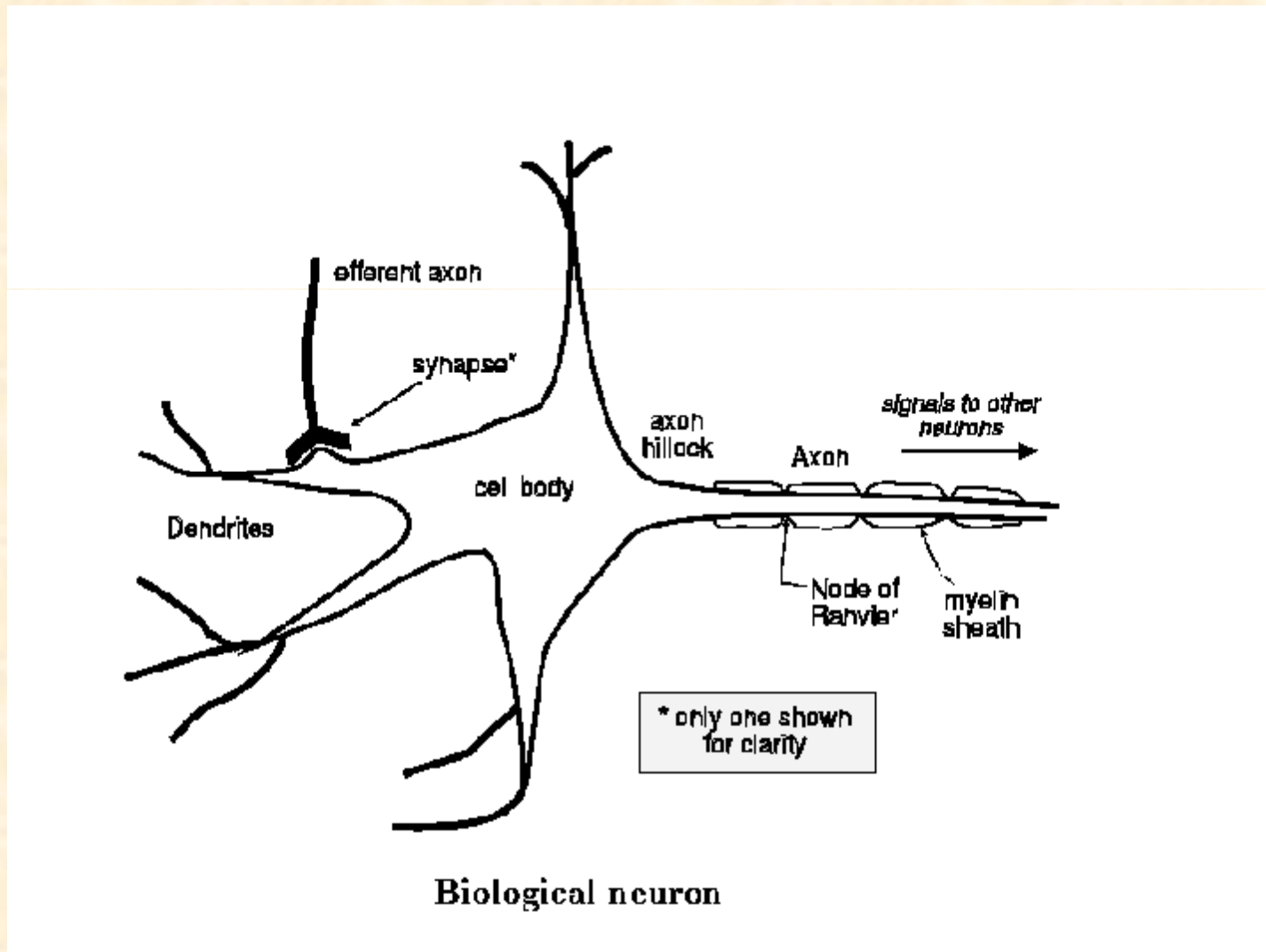


Motivation

Natural Systems perform very complex information processing tasks with completely different "hardware" than conventional (von Neumann) computers

The 100-step program constraint [Jerry Feldman]

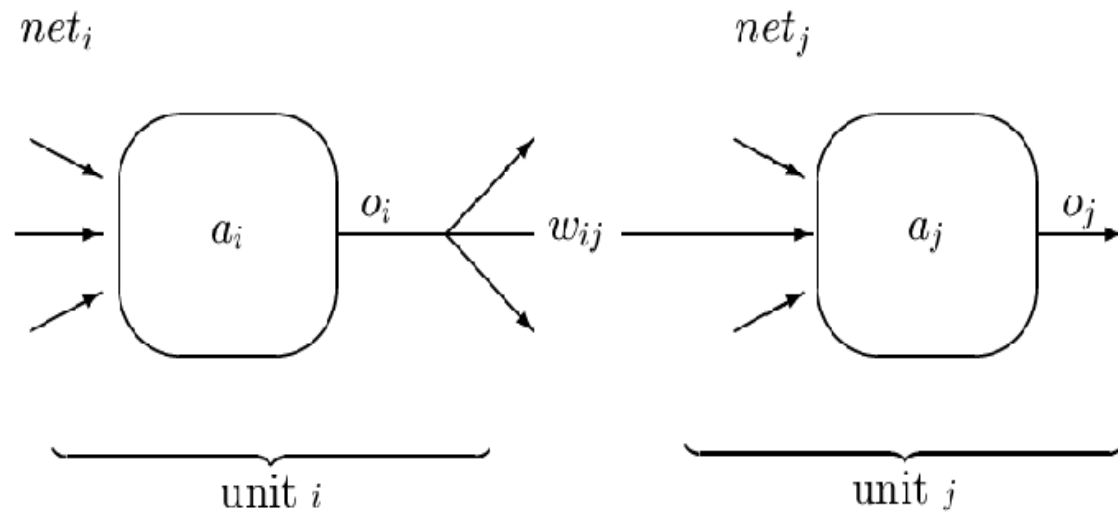
- Neurons operate in $\approx 1ms$
- Humans do sophisticated processing in $\approx 0.1s$
-  Only 100 serial steps
-  *massive parallelism*





Artificial Neural Network (ANN)

Aspects of ANNs





Definition:

An *Artificial Neural Network* (ANN) is an information processing device consisting of a large number of highly interconnected processing elements. Each processing element (*unit*) performs only very simple computations.

Remarks:

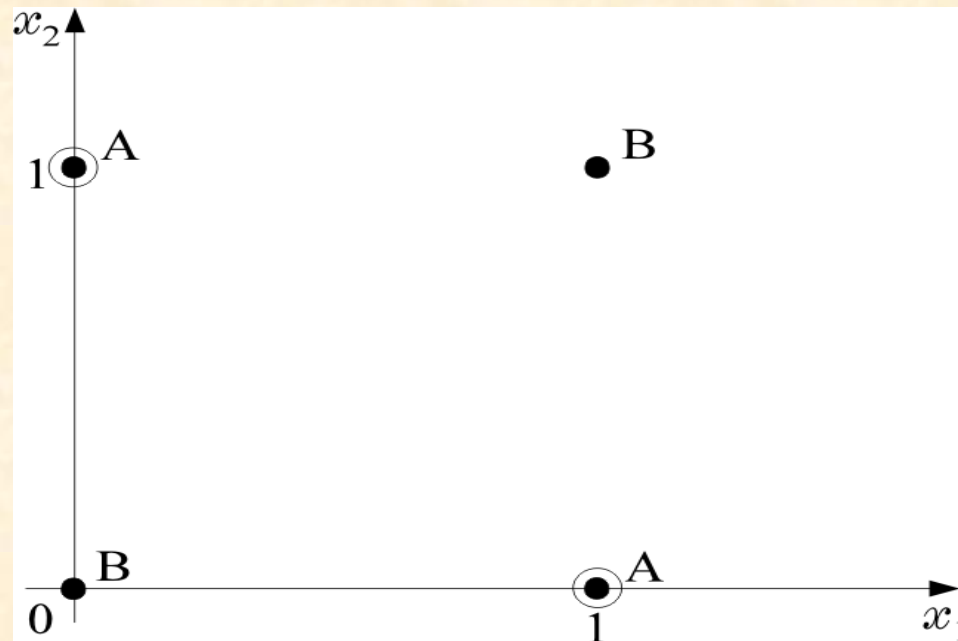
- Each unit computes a single *activation-value*
- Environmental interaction through a subset of units
- Behavior depends on interconnection structure
- Structure may adapt by *learning*



Non Linear Classifiers

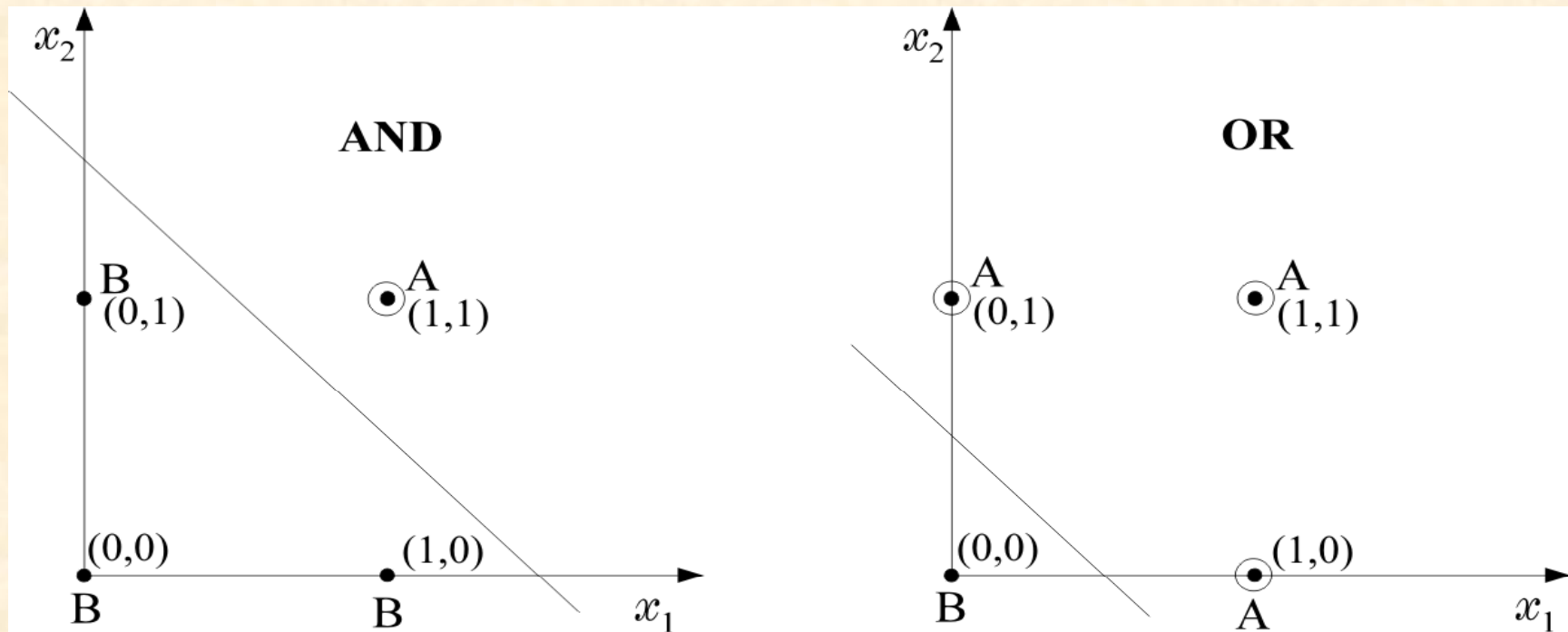
❖ The XOR problem

x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B





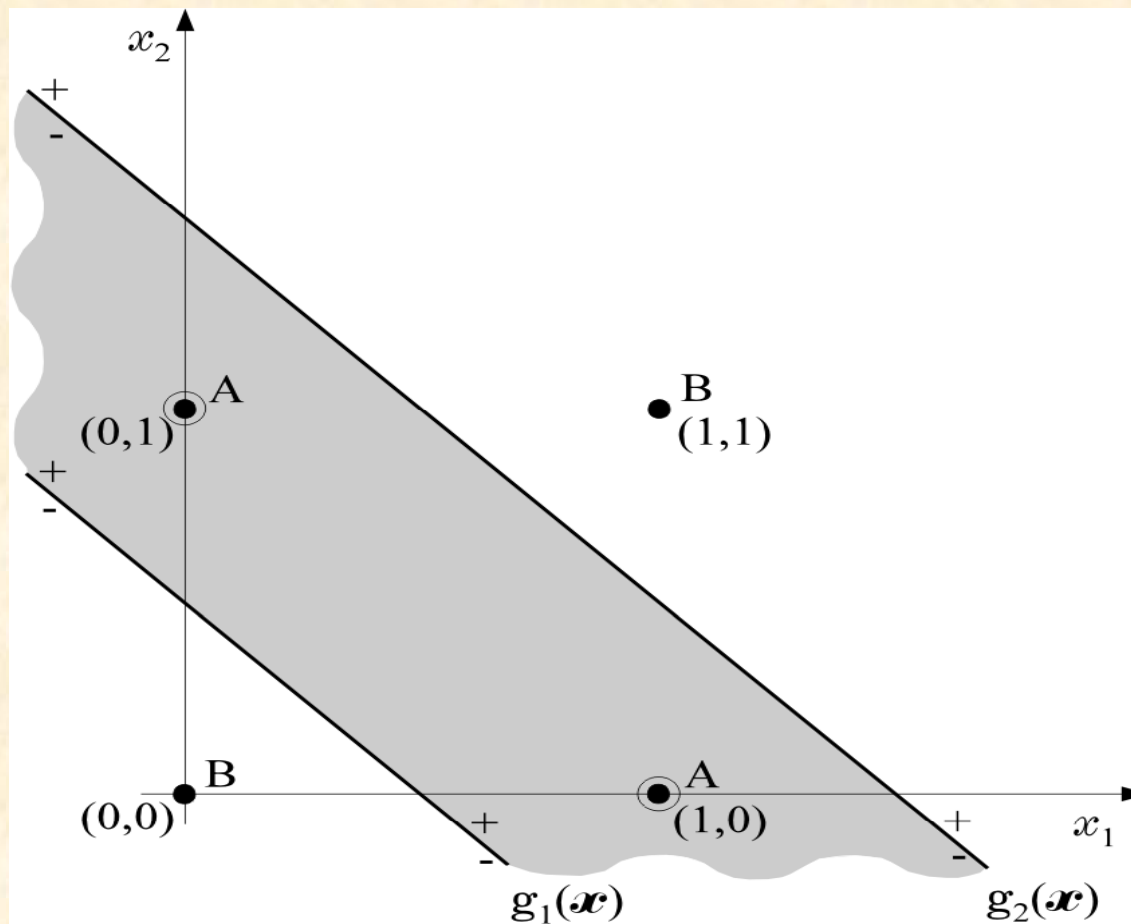
- ❖ There is no single line (hyperplane) that separates class A from class B. On the contrary, AND and OR operations are linearly separable problems





❖ The Two-Layer Perceptron

- For the XOR problem, draw **two**, instead, of one lines





- Then class B is located **outside** the shaded area and class A **inside**. This is a **two-phase** design.
- Phase 1: Draw two lines (hyperplanes)

$$g_1(\underline{x}) = g_2(\underline{x}) = 0$$

Each of them is realized by a perceptron. The outputs of the perceptrons will be

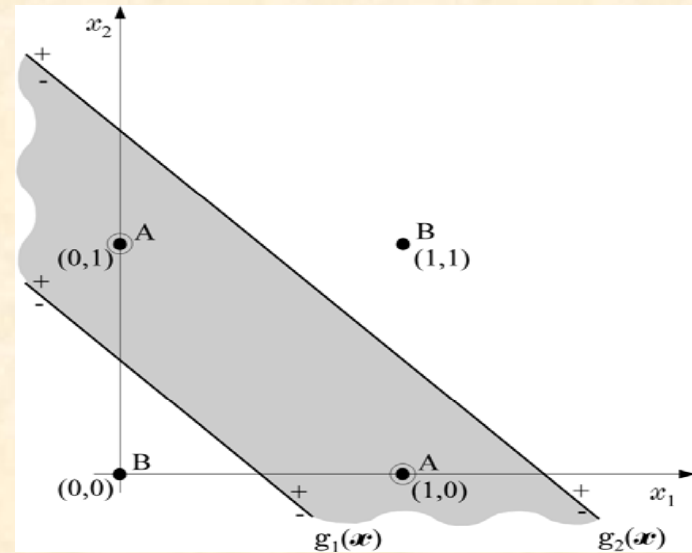
$$y_i = f(g_i(\underline{x})) = \begin{cases} 0 \\ 1 \end{cases} \quad i = 1, 2$$

depending on the position of \underline{x} .

- Phase 2: Find the position of \underline{x} *w.r.t.* **both** lines, based on the values of y_1, y_2 .



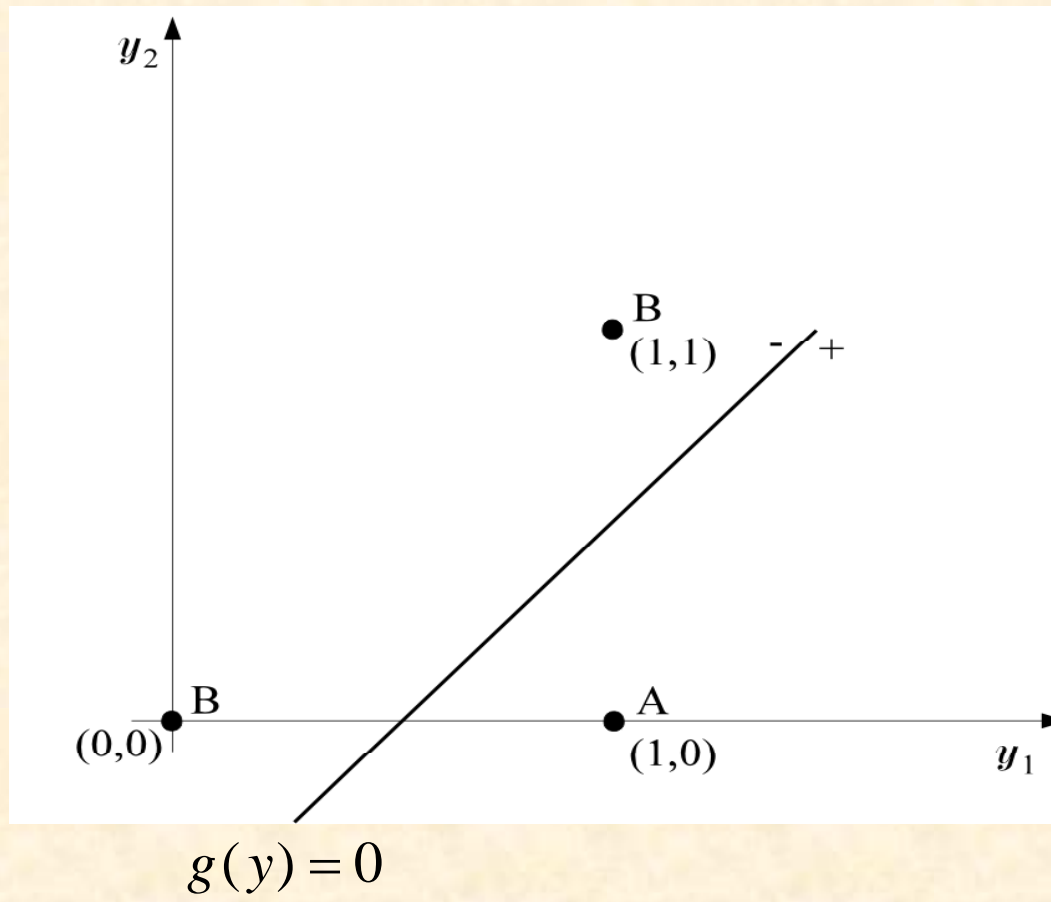
1 st phase				2 nd phase
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)



- Equivalently: The computations of the first phase perform a mapping $\underline{x} \rightarrow \underline{y} = [y_1, y_2]^T$



The decision is now performed on the **transformed** \underline{y} data.

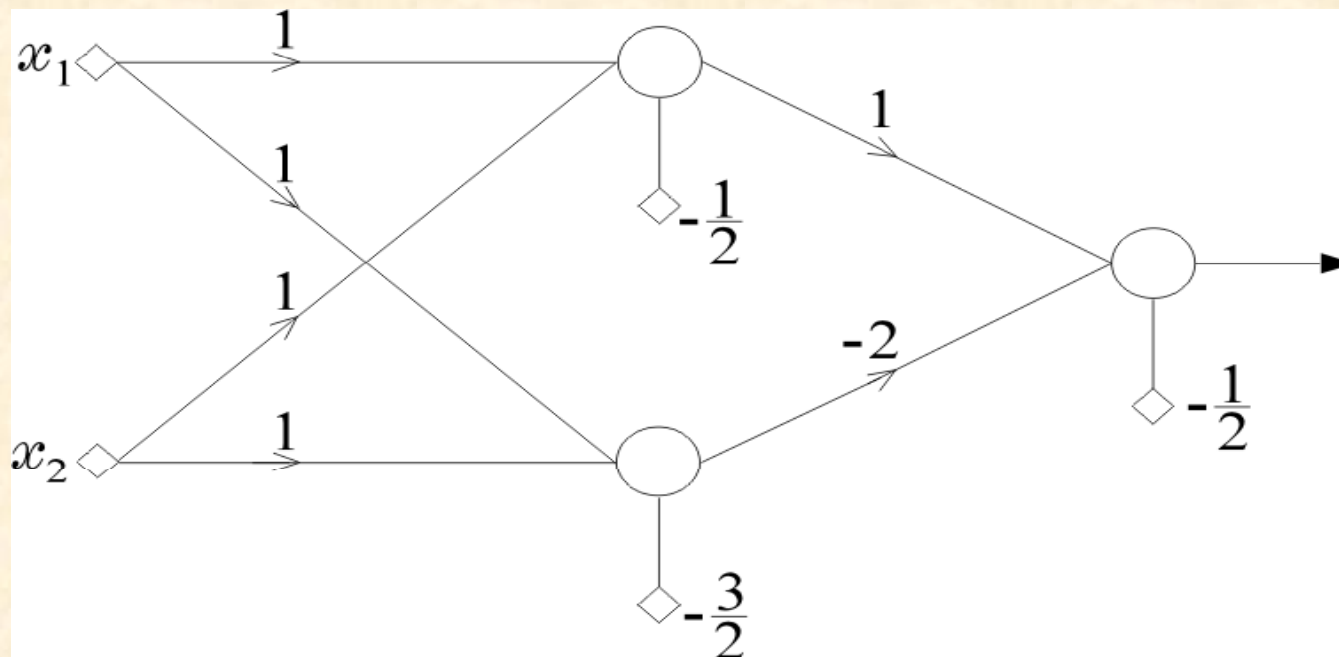


This can be performed via a second line, which can also be realized by a perceptron.



- Computations of the first phase perform a **mapping** that **transforms** the **nonlinearly** separable problem to a **linearly** separable one.

- The architecture





- This is known as the **two layer^(*)** perceptron with one **hidden** and **one output layer**. The activation functions are

$$f(.) = \begin{cases} 0 \\ 1 \end{cases}$$

- The neurons (nodes) of the figure realize the following lines (hyperplanes)

$$g_1(\underline{x}) = x_1 + x_2 - \frac{1}{2} = 0$$

$$g_2(\underline{x}) = x_1 + x_2 - \frac{3}{2} = 0$$

$$g(\underline{y}) = y_1 - 2y_2 - \frac{1}{2} = 0$$

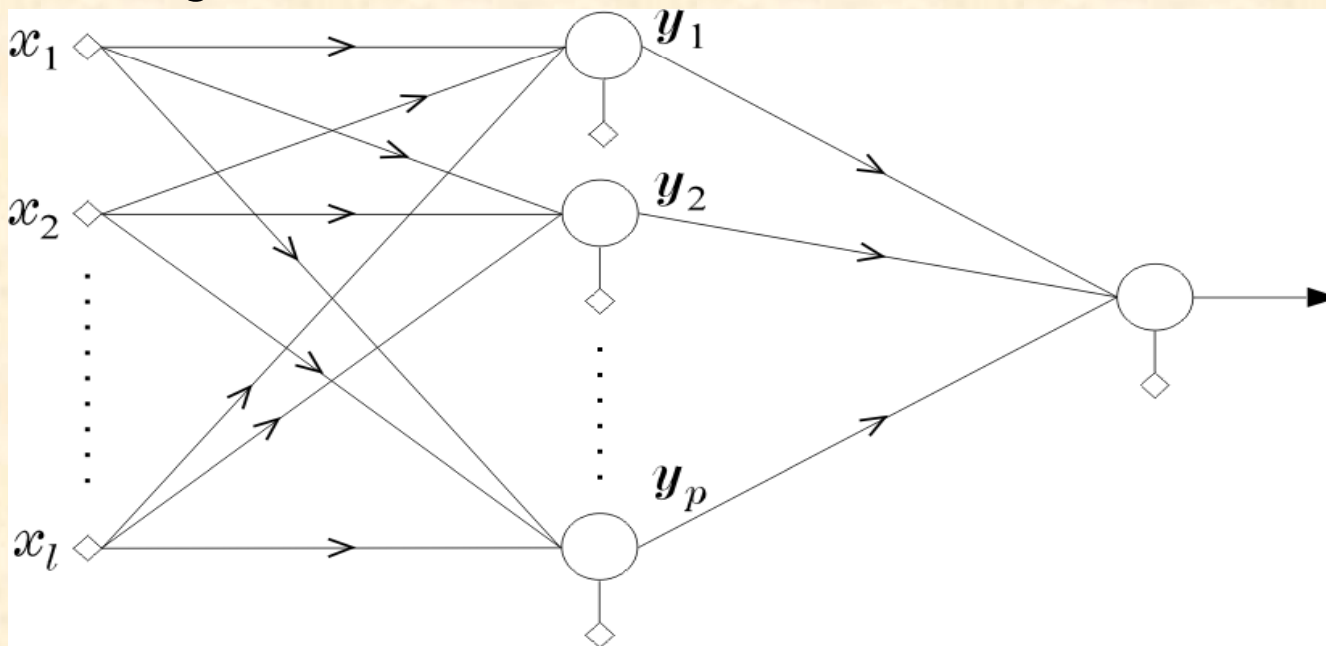
() NOTE: Duba, Hart and Stork, in their book they call it a three layer perceptron. In general in their notation, what we call N-layer they call it (N+1)-Layer*



❖ Classification capabilities of the two-layer perceptron

- The mapping performed by the first layer neurons is **onto the vertices** of the unit side square, e.g., $(0, 0), (0, 1), (1, 0), (1, 1)$.

- The more general case,



$$\underline{x} \in R^l$$

$$\underline{x} \rightarrow \underline{y} = [y_1, \dots, y_p]^T, y_i \in \{0, 1\} \quad i = 1, 2, \dots, p$$

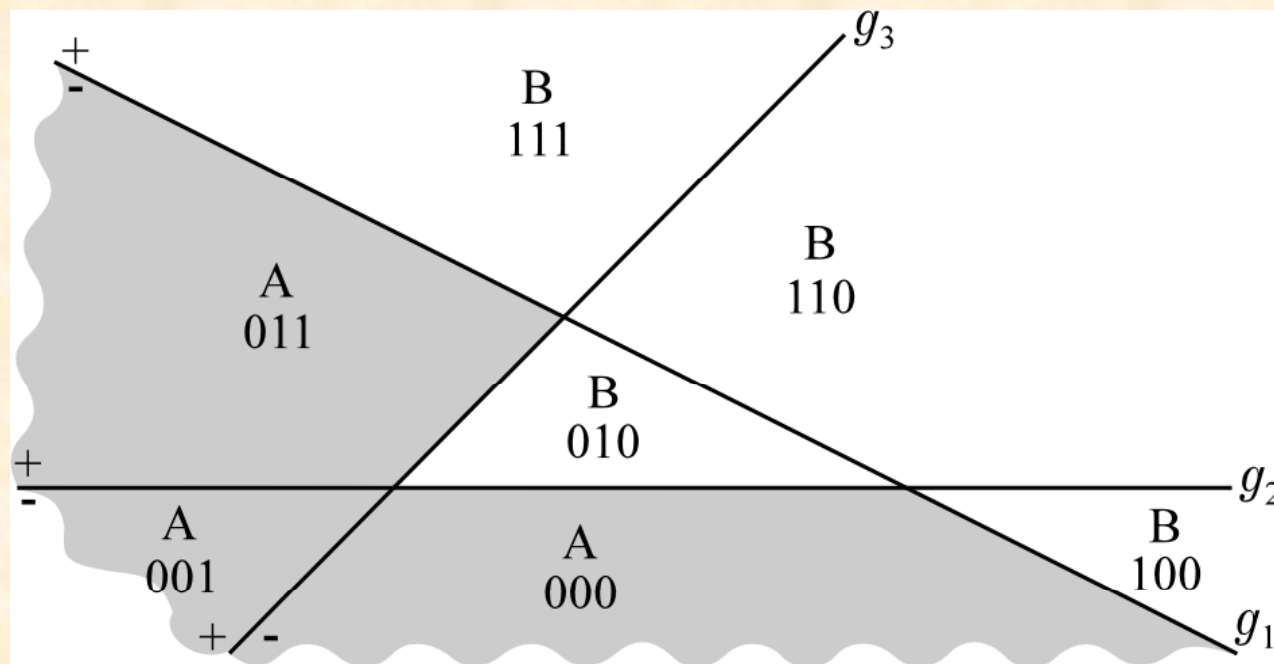


performs a mapping of a vector
onto the vertices of the unit side H_p hypercube

- The mapping is achieved with p neurons each realizing a hyperplane. The output of each of these neurons is 0 or 1 depending on the **relative position** of \underline{x} w.r.t. the hyperplane.



- Intersections of these hyperplanes form regions in the l -dimensional space. Each region corresponds to a vertex of the H_p unit hypercube.



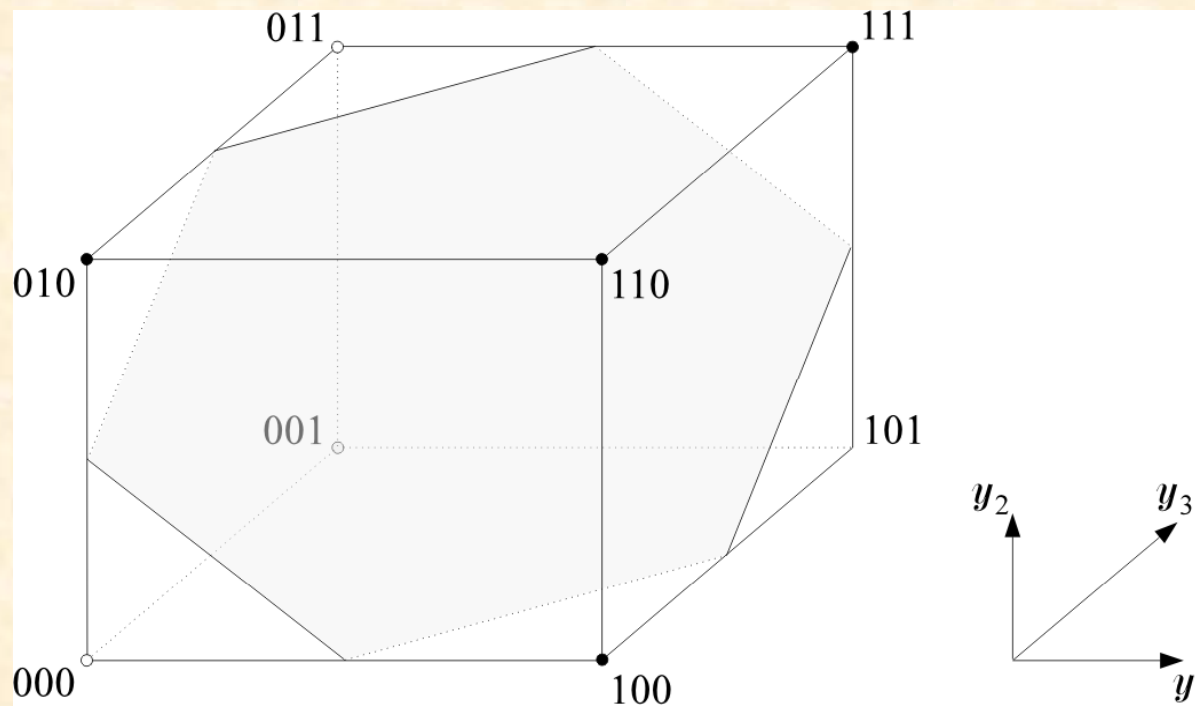


For example, the 001 vertex corresponds to the region which is located

to the (-) side of $g_1(\underline{x})=0$

to the (-) side of $g_2(\underline{x})=0$

to the (+) side of $g_3(\underline{x})=0$



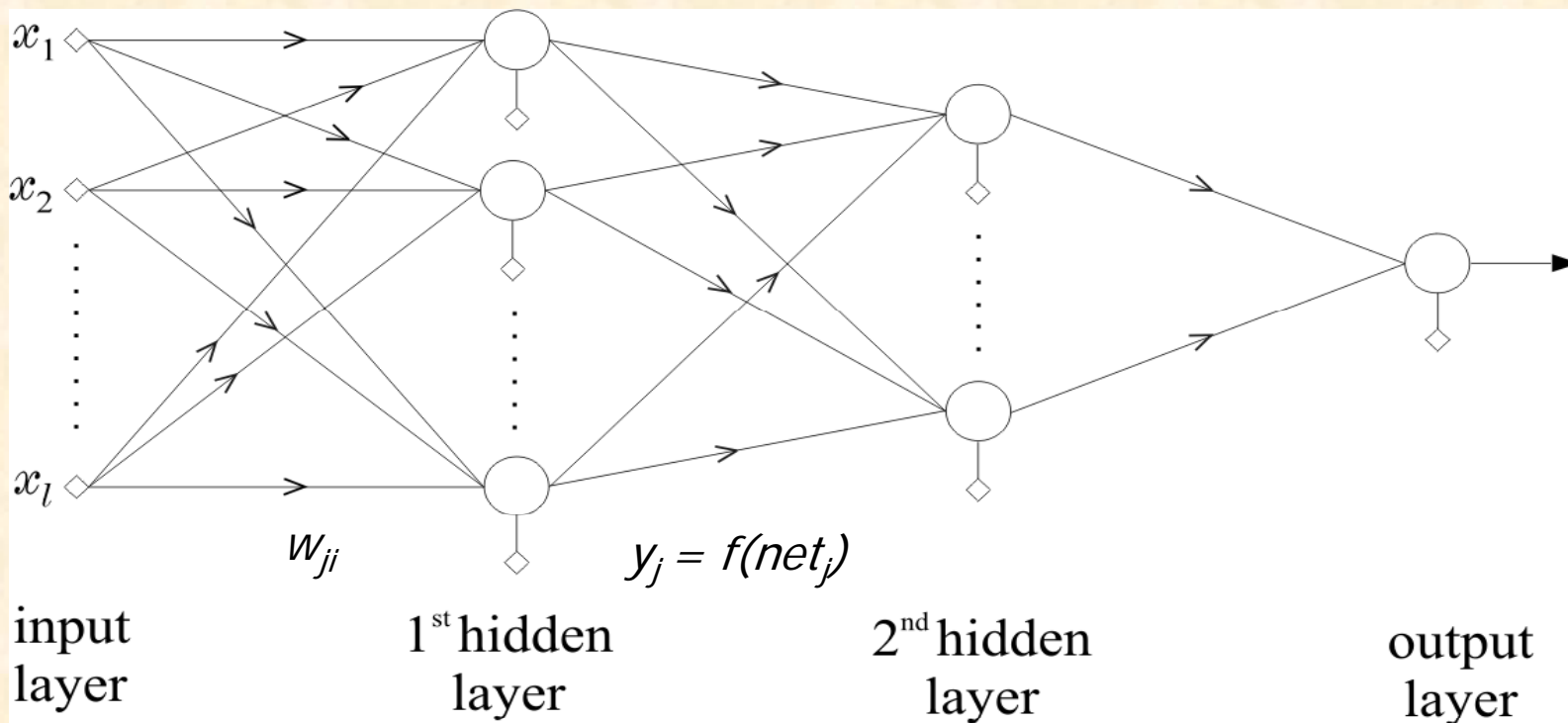


- The output neuron realizes a hyperplane in the transformed y space, that separates some of the vertices from the others. Thus, the two layer perceptron has the capability to classify vectors into **classes that consist of unions of polyhedral regions**. But **NOT ANY** union. It depends on the relative position of the corresponding vertices.



❖ Three layer-perceptrons

➤ The architecture

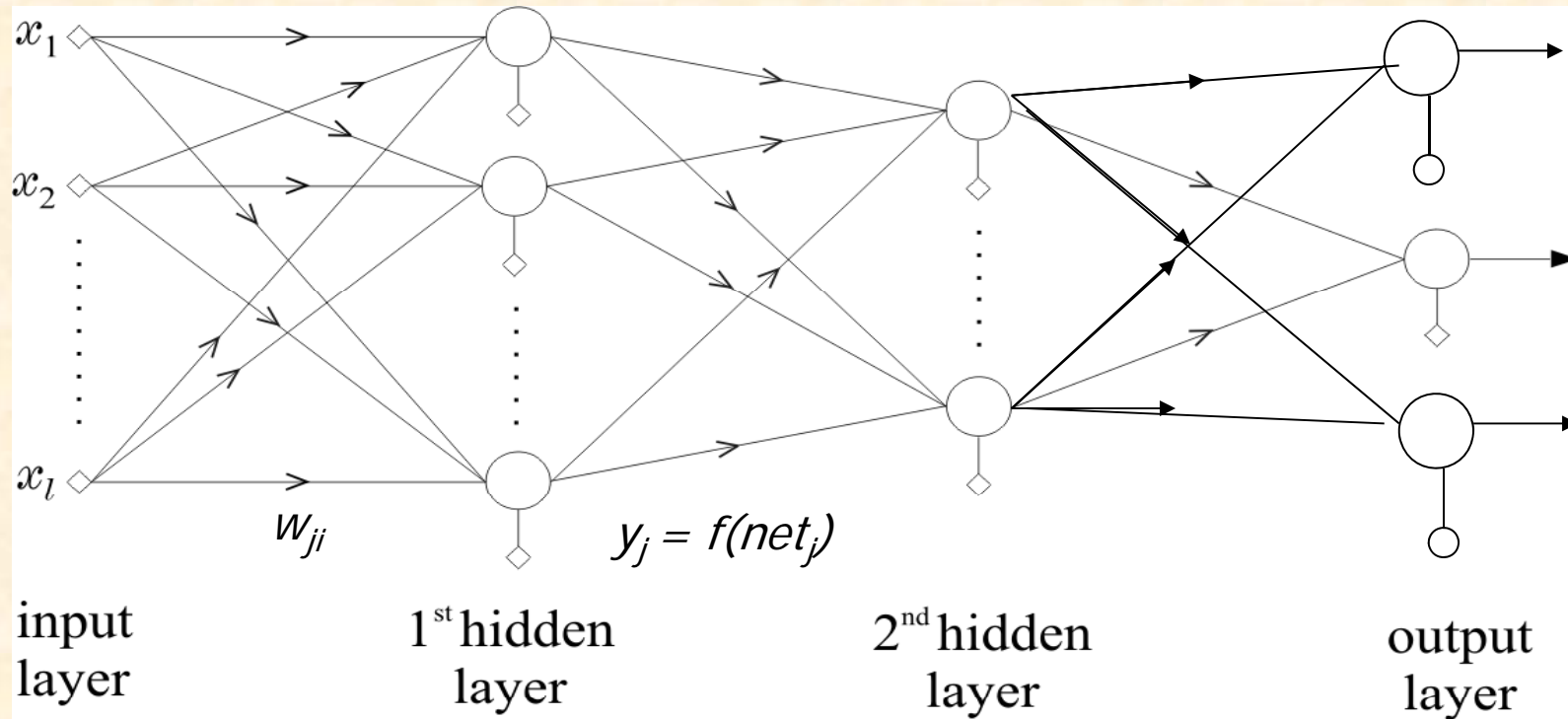


- This is capable to classify vectors into classes consisting of **ANY** union of polyhedral regions.
- The idea is similar to the XOR problem. It realizes more than one planes in the $\underline{y} \in R^p$ space.



❖ Three layer-perceptrons with C classes

➤ The architecture for more than 2 classes





- ❖ A single “bias unit” is connected to each unit other than the input units

- ❖ Net activation:
$$net_j = \sum_{i=1}^d x_i w_{ji} + w_{j0} = \sum_{i=0}^d x_i w_{ji} \equiv w_j^t \cdot x,$$

where the subscript i indexes units in the input layer, j in the hidden; w_{ji} denotes the input-to-hidden layer weights at the hidden unit j . (In neurobiology, such weights or connections are called “synapses”)

- ❖ Each hidden unit emits an output that is a nonlinear function of its activation, that is: $y_j = f(net_j)$



- The reasoning
 - For each vertex, corresponding to class, say A, construct a hyperplane which leaves **THIS vertex** on one side (+) and **ALL** the others to the other side (-).
 - The output neuron realizes an OR gate

- Overall:

The first layer of the network forms the **hyperplanes**, the second layer forms the **regions** and the output neuron forms the **classes**.

❖ Designing Multilayer Perceptrons

- One direction is to adopt the above rationale and develop a structure that classifies **correctly all** the training patterns.
- The other direction is to choose a structure and compute the synaptic weights to **optimize a cost function**.



❖ Expressive Power of multi-layer Networks

Question: Can every decision be implemented by a three-layer network described by equation (1) ?

Answer: Yes (due to A. Kolmogorov)

“Any continuous function from input to output can be implemented in a three-layer net, given sufficient number of hidden units n_H , proper nonlinearities, and weights.”

$$g(\mathbf{x}) = \sum_{j=1}^{2n+1} \delta_j \left(\sum \beta_{ij} (x_i) \right) \quad \forall \mathbf{x} \in I^n (I = [0,1]; n \geq 2)$$

for properly chosen functions δ_j and β_{ij}



- ❖ Each of the $2n+1$ hidden units δ_j takes as input a sum of d nonlinear functions, one for each input feature x_i
- ❖ Each hidden unit emits a nonlinear function δ_j of its total input
- ❖ The output unit emits the sum of the contributions of the hidden units

Unfortunately: Kolmogorov's theorem tells us very little about how to find the nonlinear functions based on data; this is the central problem in network-based pattern recognition



Backpropagation Algorithm

- ❖ Any function from input to output can be implemented as a three-layer neural network
- ❖ These results are of greater theoretical interest than practical, since the construction of such a network requires the nonlinear functions and the weight values which are unknown!

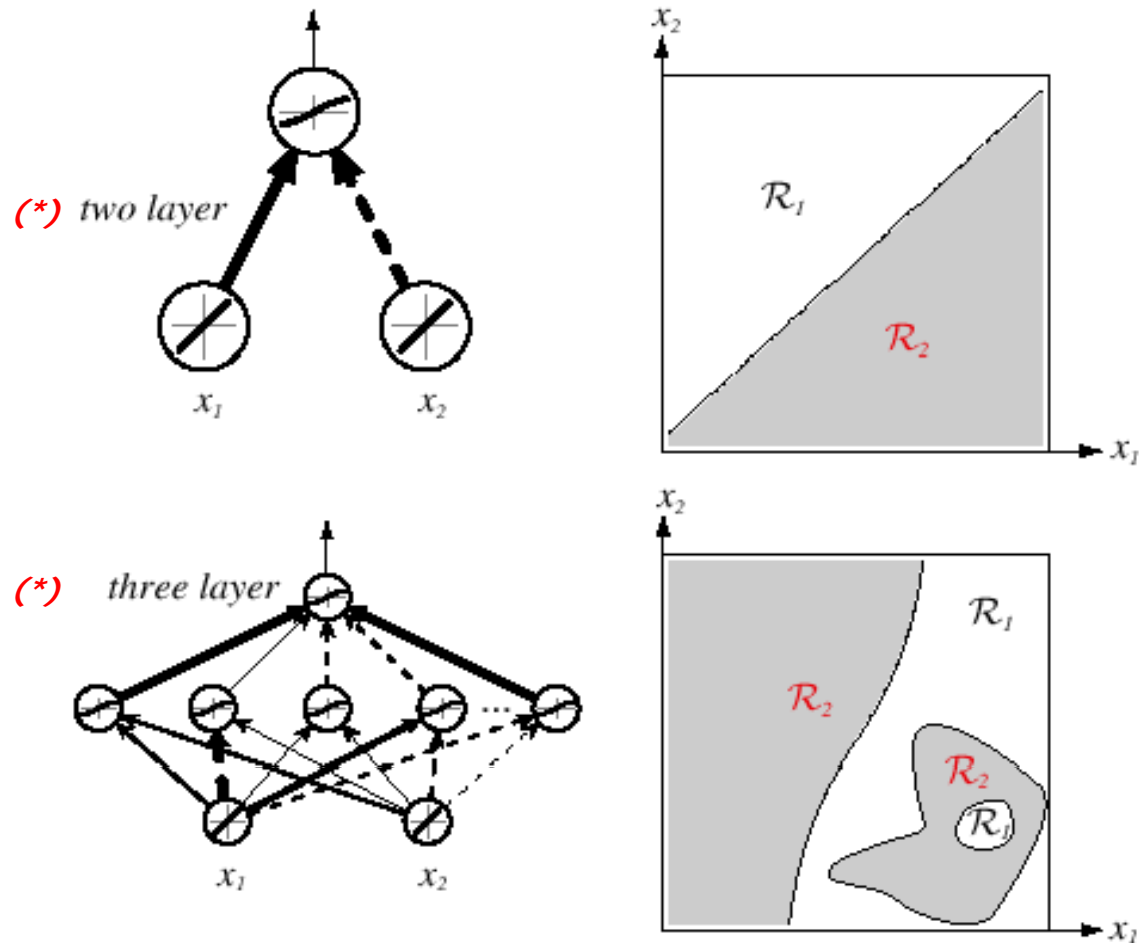


FIGURE 6.3. Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex or simply connected. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

(*) Στο σχήμα αυτό, αντικατέστησε two layer με one layer και three layer με two layer



- ❖ Our goal now is to set the interconnexion weights based on the training patterns and the desired outputs
- ❖ In a three-layer network, it is a straightforward matter to understand how the output, and thus the error, depend on the hidden-to-output layer weights
- ❖ The power of backpropagation is that it enables us to compute an effective error for each hidden unit, and thus derive a learning rule for the input-to-hidden weights, this is known as:



❖ The Backpropagation Algorithm

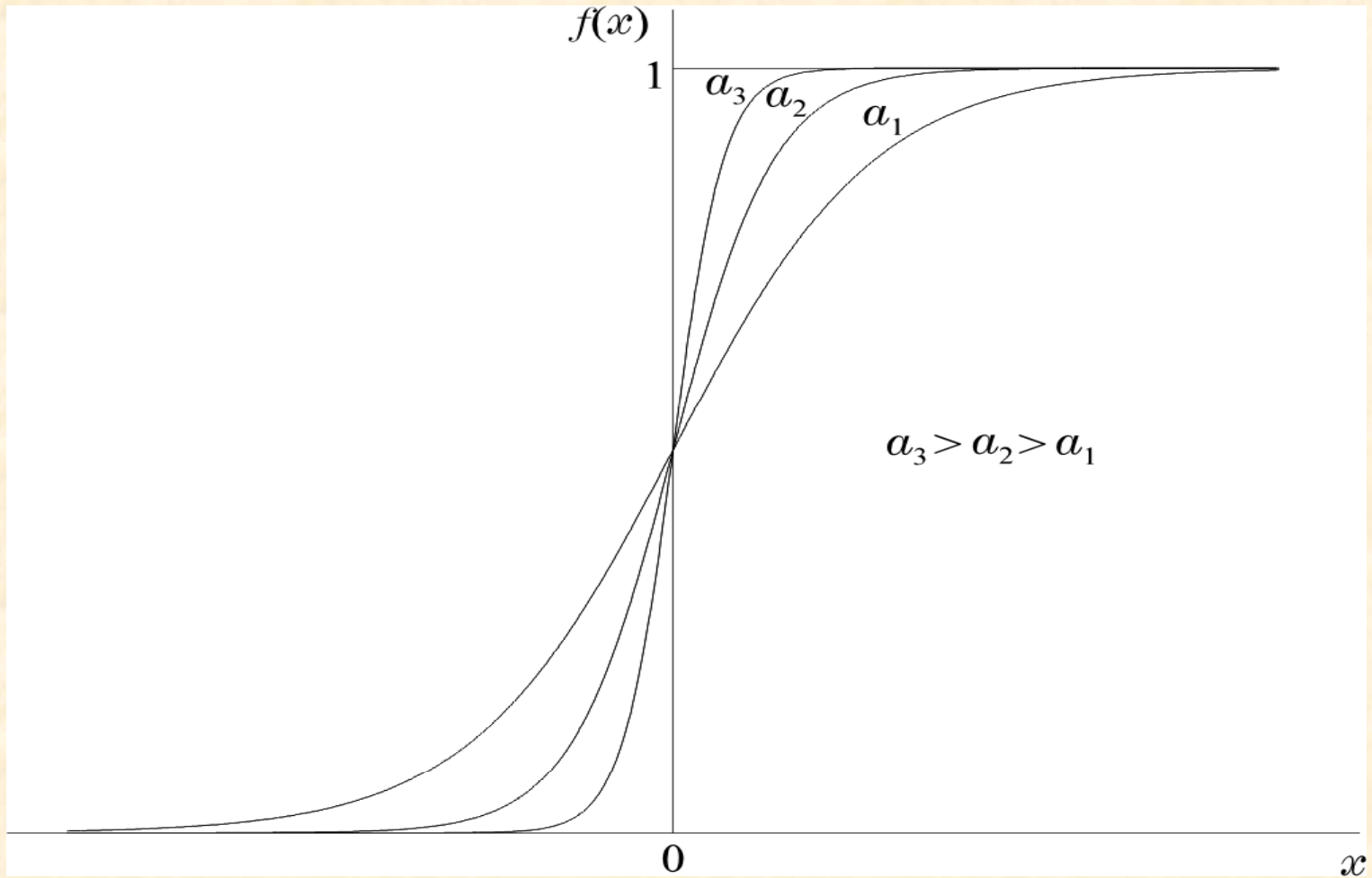
- This is an algorithmic procedure that computes the synaptic weights *iteratively*, so that an adopted *cost function is minimized (optimized)*
- In a large number of optimizing procedures, computation of derivatives are involved. Hence, discontinuous activation functions pose a problem, i.e.,

$$\cancel{f(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}}$$

- There is always an escape path!!! The logistic function

$$f(x) = \frac{1}{1 + \exp(-ax)}$$

is an example. Other functions are also possible and in some cases more desirable.





➤ The steps:

- Adopt an optimizing cost function, e.g.,
 - Least Squares Error
 - Relative Entropy

between desired responses and actual responses of the network for the available training patterns. That is, from now on we have to live with errors. We only try to minimize them, using certain criteria.

- Adopt an algorithmic procedure for the optimization of the cost function with respect to the synaptic weights
e.g.,
 - Gradient descent
 - Newton's algorithm
 - Conjugate gradient



- The task is a **nonlinear** optimization one. For the gradient descent method

$$\underline{w}_1^r(\text{new}) = \underline{w}_1^r(\text{old}) + \Delta \underline{w}_1^r$$

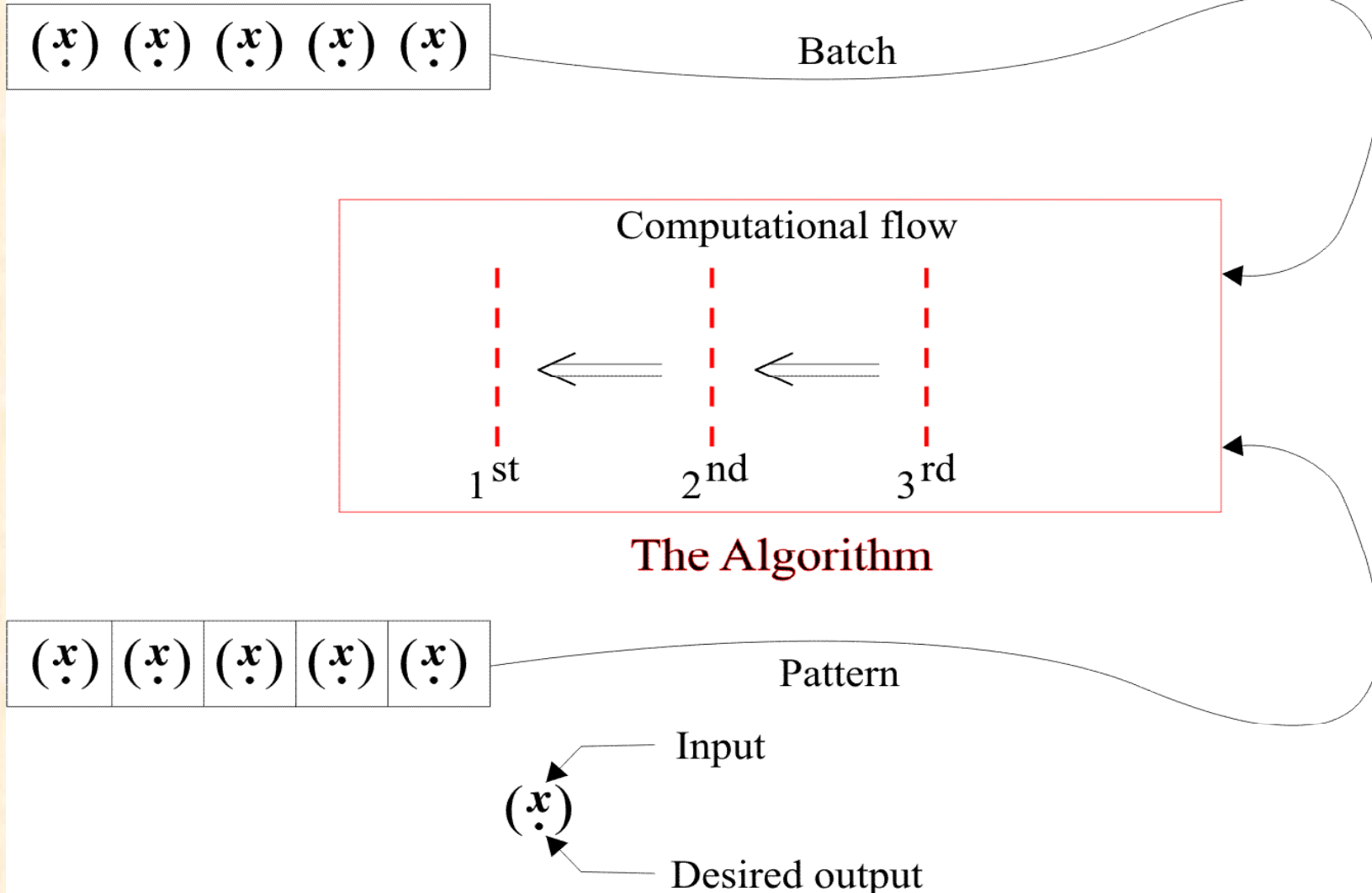
$$\Delta \underline{w}_1^r = -\eta \frac{\partial J}{\partial w_1^r}$$

- η is the **learning rate**



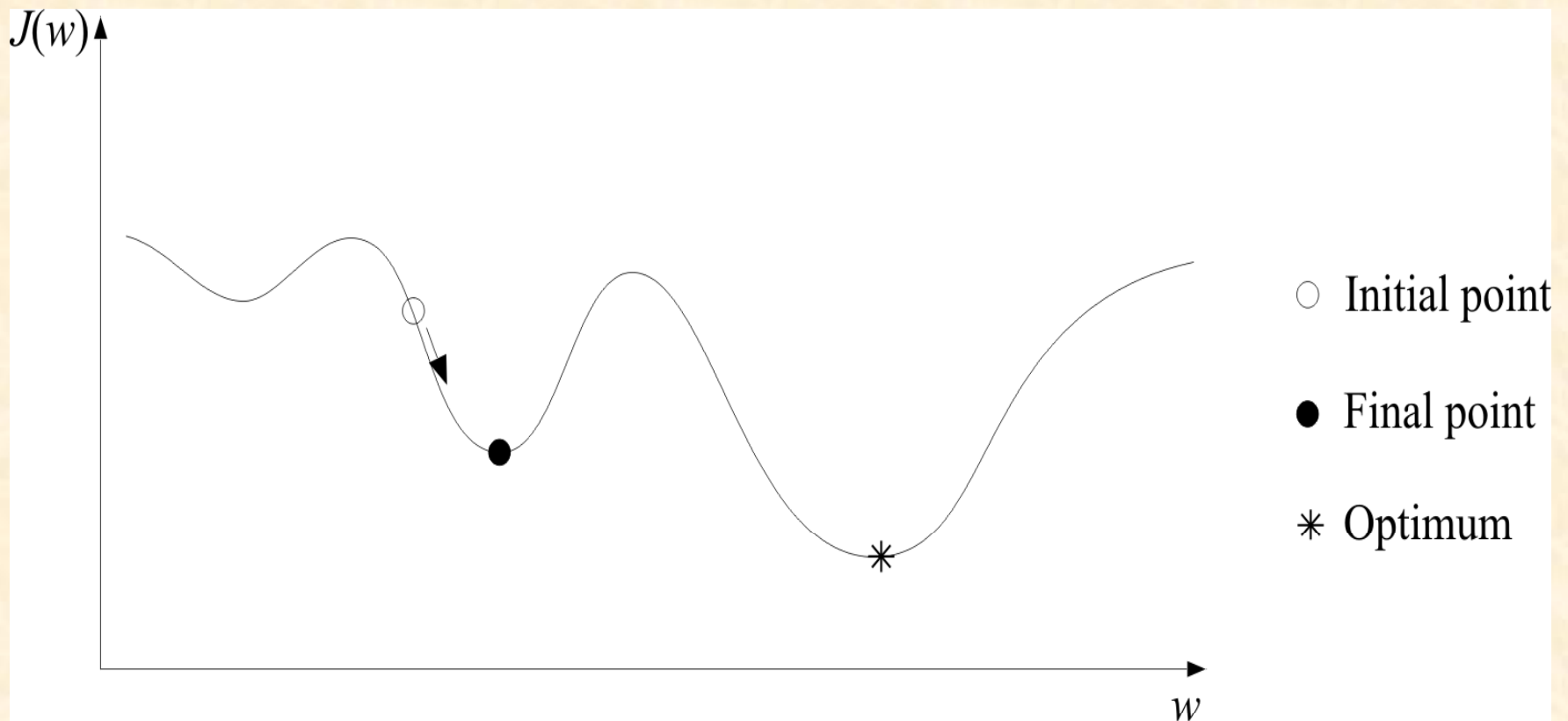
- The Procedure:
 - Initialize unknown weights randomly with small values.
 - Compute the gradient terms **backwards**, starting with the weights of the last (3rd) layer and then moving towards the first
 - Update the weights
 - Repeat the procedure until a termination procedure is met

- Two major philosophies:
 - **Batch mode**: The gradients of the last layer are computed once **ALL training data** have appeared to the algorithm, i.e., by summing up all error terms.
 - **Pattern mode**: The gradients are computed every time **a new training data pair appears**. Thus gradients are based on successive individual errors.





- A major problem: The algorithm may converge to a local minimum





➤ The Cost function choice

Examples:

- The Least Squares

$$J = \sum_{i=1}^N E(i)$$

$$E(i) = \sum_{m=1}^k e_m^2(i) = \sum_{m=1}^k (y_m(i) - \hat{y}_m(i))^2$$

$$i = 1, 2, \dots, N$$

$y_m(i) \rightarrow$ Desired response of the m^{th} output neuron (1 or 0) for $\underline{x}(i)$

$\hat{y}_m(i) \rightarrow$ Actual response of the m^{th} output neuron, in the interval $[0, 1]$, for input $\underline{x}(i)$



➤ The cross-entropy

$$J = \sum_{i=1}^N E(i)$$

$$E(i) = \sum_{m=1}^k \{y_m(i) \ln \hat{y}_m(i) + (1 - y_m(i)) \ln(1 - \hat{y}_m(i))\}$$

This presupposes an interpretation of y and \hat{y} as **probabilities**

- Classification error rate. This is also known as discriminative learning. Most of these techniques use a smoothed version of the classification error.



- **Remark 1:** A common feature of all the above is the danger of local minimum convergence. **“Well formed”** cost functions guarantee convergence to a **“good”** solution, that is one that classifies correctly **ALL** training patterns, provided such a solution exists. The **cross-entropy** cost function **is a well formed one**. The **Least Squares is not**.



- **Remark 2:** Both, the Least Squares and the cross entropy lead to output values $\hat{y}_m(i)$ that approximate **optimally class a-posteriori probabilities!!!**

$$\hat{y}_m(i) \cong P(\omega_m | \underline{x}(i))$$

That is, the probability of class ω_m given $\underline{x}(i)$.

This is a very interesting result. It **does not** depend on the underlying distributions. It is a characteristic of **certain** cost functions. How good or bad is the approximation, depends on the underlying model. Furthermore, it is **only** valid at the **global minimum**.



Network Learning



$$net_k = \sum_{i=1}^d x_i w_{ki} + w_{k0} = \sum_{i=0}^d x_i w_{ki} \equiv \mathbf{w}_k^t \cdot \mathbf{x},$$

- Let t_k be the k -th target (or desired) output and z_k be the k -th computed output with $k = 1, \dots, c$ and \mathbf{w} represents all the weights of the network

- The training error: (Least Square)

$$J(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^c (t_k - z_k)^2 = \frac{1}{2} \|\mathbf{t} - \mathbf{z}\|^2$$

- The backpropagation learning rule is based on gradient descent
 - The weights are initialized with pseudo-random values and are changed in a direction that will reduce the error:

$$\Delta \mathbf{w} = -\eta \frac{\partial J}{\partial \mathbf{w}}$$



where η is **the learning rate** which indicates the relative size of the change in weights

$$w(m + 1) = w(m) + \Delta w(m)$$

where m is the m -th pattern presented

➤ Error on the hidden-to-output weights

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}$$

where the sensitivity of unit k is defined as: $\delta_k = -\frac{\partial J}{\partial net_k}$

and describes how the overall error changes with the activation of the unit's net

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k} \cdot \frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k)$$



Since $net_k = w_k^t \cdot y$ therefore: $\frac{\partial net_k}{\partial w_{kj}} = y_j$

Conclusion: the weight update (or learning rule) for the hidden-to-output weights is:

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j$$

➤ Error on the input-to-hidden units

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j} \cdot \frac{\partial y_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}}$$



However,

$$\begin{aligned}\frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j} \left[\frac{1}{2} \sum_{k=1}^c (t_k - z_k)^2 \right] = - \sum_{k=1}^c (t_k - z_k) \frac{\partial z_k}{\partial y_j} \\ &= - \sum_{k=1}^c (t_k - z_k) \frac{\partial z_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial y_j} = - \sum_{k=1}^c (t_k - z_k) f'(net_k) w_{kj}\end{aligned}$$

Similarly as in the preceding case, we define the sensitivity for a hidden unit:

$$\delta_j \equiv f'(net_j) \sum_{k=1}^c w_{kj} \delta_k$$

which means that: “The sensitivity at a hidden unit is simply the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{kj} ; all multiplied by $f'(net_j)$ ”

Conclusion: The learning rule for the input-to-hidden weights is:

$$\Delta w_{ji} = \eta x_i \delta_j = \eta \underbrace{\left[\sum w_{kj} \delta_k \right]}_{\delta_j} f'(net_j) x_i$$



STOCHASTIC BACKPROPAGATION

- Starting with a pseudo-random weight configuration, the stochastic backpropagation algorithm can be written as:

```

Begin initialize  $n_H$ ;  $w$ , criterion  $\theta$ ,  $\eta$ ,  $m \leftarrow 0$ 
  do  $m \leftarrow m + 1$ 
     $\mathbf{x}^m \leftarrow$  randomly chosen pattern
     $w_{ji} \leftarrow w_{ji} + \eta \delta_j \mathbf{x}_i$ ;  $w_{kj} \leftarrow w_{kj} + \eta \delta_k \mathbf{y}_j$ 
  until  $||\nabla J(\mathbf{w})|| < \theta$ 
  return  $w$ 
End

```



BATCH BACKPROPAGATION

- Starting with a pseudo-random weight configuration, the batch backpropagation algorithm can be written as:

```

Begin initialize  $n_H$ ;  $w$ , criterion  $\theta$ ,  $\eta$ ,  $r \leftarrow 0$ 
  do  $r \leftarrow r + 1$  (epoch)
     $m \leftarrow 0$ ;  $\Delta w_{ji} \leftarrow 0$ ;  $\Delta w_{kj} \leftarrow 0$ 
    do  $m \leftarrow m + 1$ 
       $x^m \leftarrow$  select pattern
       $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \delta_j x_i$ ;  $\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \delta_k y_j$ 
    until  $m=n$ 
     $w_{ji} \leftarrow w_{ji} + \eta \delta_j x_i$ ;  $w_{kj} \leftarrow w_{kj} + \eta \delta_k y_j$ 
  until  $||\nabla J(w)|| < \theta$ 
return  $w$ 
End

```




➤ Stopping criterion

- The algorithm terminates when the **change** in the criterion function $J(w)$ is smaller than some preset value θ
- There are other stopping criteria that lead to better performance than this one
- So far, we have considered the error on a single pattern, but we want to consider an error defined over the entirety of patterns in the training set
- The total training error is the sum over the errors of n individual patterns

$$J = \sum_{p=1}^n J_p \quad (1)$$



➤ Stopping criterion (cont.)

- A weight update may reduce the error on the single pattern being presented but can increase the error on the full training set
- However, given a large number of such individual updates, the total error of equation (1) decreases



Definitions

❖ Training set:

- A set of examples used for learning, that is to fit the parameters [i.e., weights] of the classifier.

❖ Validation set:

- A set of examples used to tune the parameters [i.e., architecture, not weights] of a classifier, for example to choose the number of hidden units in a neural network.

❖ Test set:

- A set of examples used only to assess the performance [generalization] of a fully-specified classifier.



HOLD OUT METHOD

Since our goal is to find the network having the best performance on new data, the simplest approach to the comparison of different networks is to evaluate the error function using data which is independent of that used for training. Various networks are trained by minimization of an appropriate error function defined with respect to a **training data** set. The performance of the networks is then compared by evaluating the error function using an independent **validation** set, and the network having the smallest error with respect to the validation set is selected. This approach is called the **hold out** method. Since this procedure can itself lead to some overfitting to the validation set, the performance of the selected network should be confirmed by measuring its performance on a third independent set of data called a **test set**.

The crucial point is that ***a test set is never used to choose among two or more networks***, so that the error on the test set provides an unbiased estimate of the generalization error. Any data set that is used to choose the best of two or more networks is, by definition, ***a validation set***, and the error of the chosen network on the validation set is optimistically biased.

Read more: <http://www.faqs.org/faqs/ai-faq/neural-nets/part1/section-14.html#ixzz0pxWWWVHy>



❖ Learning Curves

- Before training starts, the error on the training set is high; through the learning process, the error becomes smaller
- The error per pattern depends on the amount of training data and the expressive power (such as the number of weights) in the network
- The average error on an independent test set is always higher than on the training set, and it can decrease as well as increase
- A validation set is used in order to decide when to stop training ; we do not want to overfit the network and decrease the power of the classifier generalization

“we stop training at a minimum of the error on the validation set”

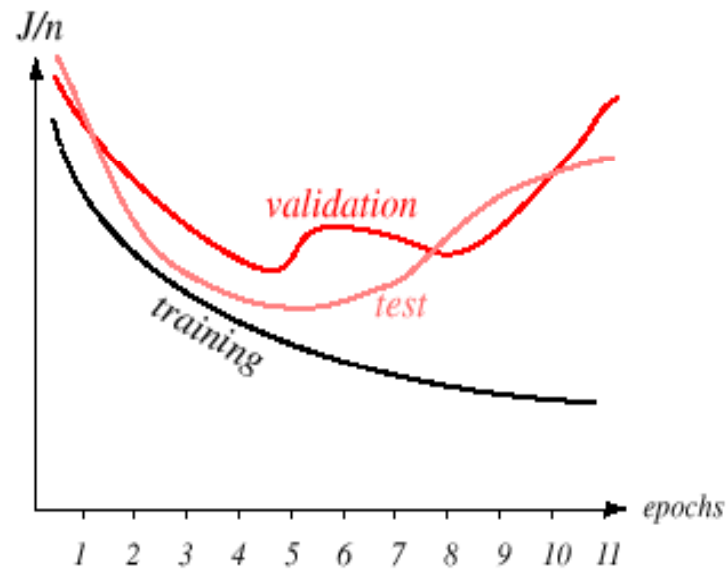


FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^n J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.



REGULARIZATION

➤ Choice of the network size.

How big a network can be. How many layers and how many neurons per layer?? There are two major directions



Pruning Techniques

- These techniques start from a large network and then weights and/or neurons are removed iteratively, according to a criterion.



—Methods based on parameter sensitivity

$$\delta J = \sum_i g_i \delta w_i + \frac{1}{2} \sum_i h_{ii} \delta w_i^2 + \frac{1}{2} \sum_i \sum_j h_{ij} \delta w_i \delta w_j$$

+ higher order terms where

$$g_i = \frac{\partial J}{\partial w_i}, \quad h_{ij} = \frac{\partial^2 J}{\partial w_i \partial w_j}$$

Near a minimum and assuming that

$$\delta J \cong \frac{1}{2} \sum_i h_{ii} \delta w_i^2$$



Pruning is now achieved in the following procedure:

- ✓ Train the network using Backpropagation for a number of steps
- ✓ Compute the **saliencies**

$$s_i = \frac{h_{ii} w_i^2}{2}$$

- ✓ Remove weights with small s_i .
- ✓ Repeat the process

— Methods based on function regularization

$$J = \sum_{i=1}^N E(i) + aE_p(\underline{w})$$



The second term favours small values for the weights, e.g.,

$$E_p(\underline{\omega}) = \sum_k h(w_k^2)$$

$$h(w_k^2) = \frac{w_k^2}{w_0^2 + w_k^2}$$

where $w_0 \cong 1$

After some training steps, weights with small values are removed.



Constructive techniques

They start with a small network and keep increasing it, according to a predetermined procedure and criterion.



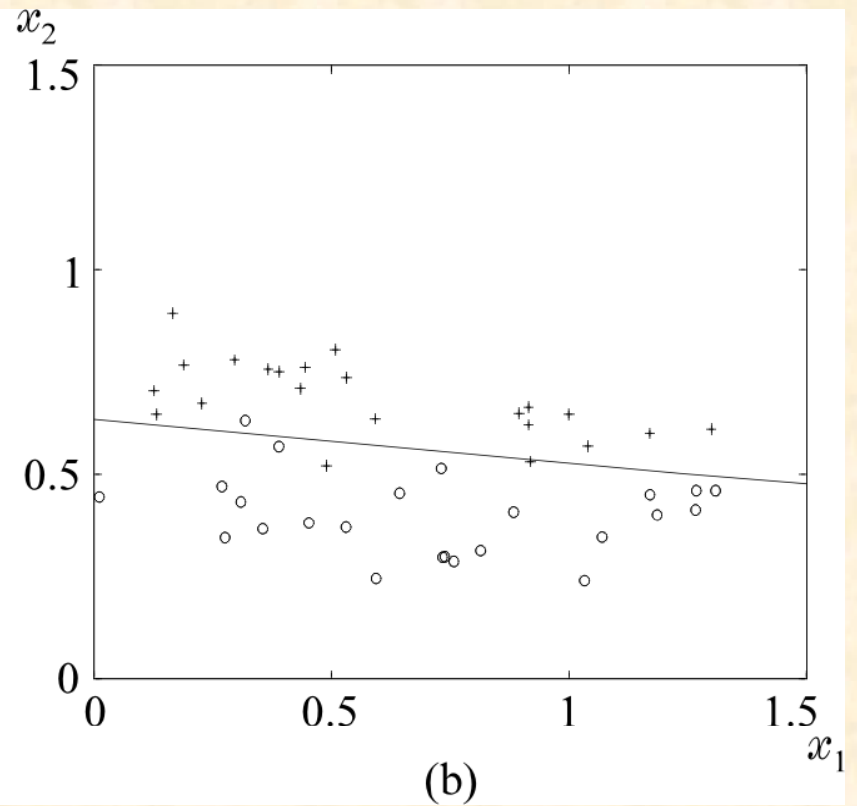
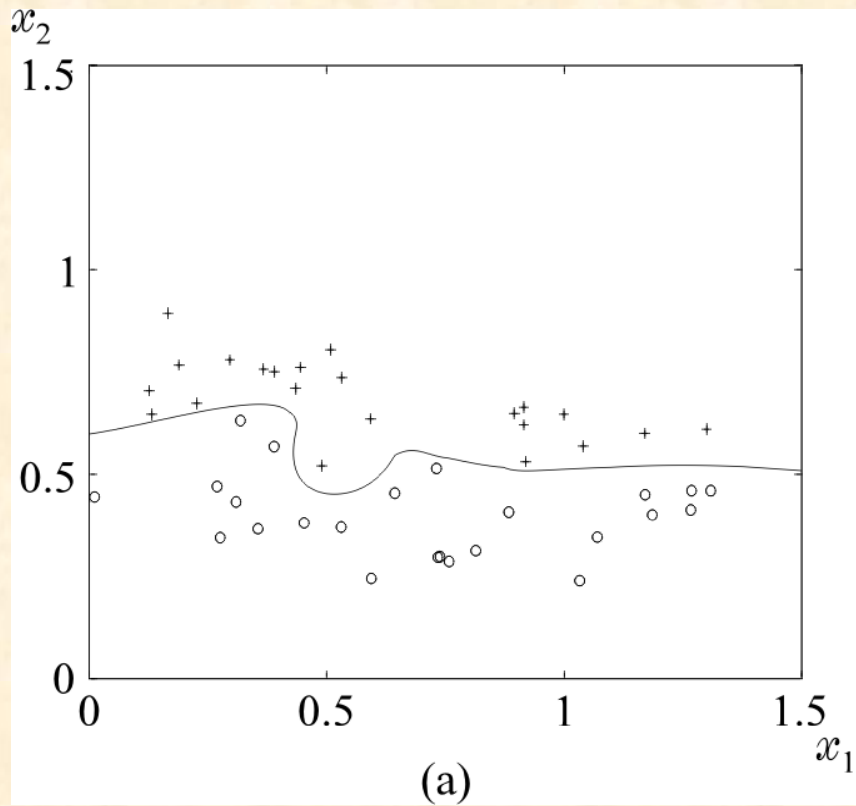
➤ **Remark:** Why not start with a large network and leave the algorithm to decide which weights are small??

This approach is just naïve. It overlooks that classifiers must have good **generalization** properties. A large network can result in small errors for the training set, since it can learn the particular details of the training set. On the other hand, it will not be able to perform well when presented with data unknown to it. The size of the network must be:

- **Large enough** to learn what makes data of the same class **similar** and data from different classes **dissimilar**
- **Small enough** not to be able to learn underlying differences between data of the same class. This leads to the so called **overfitting**.

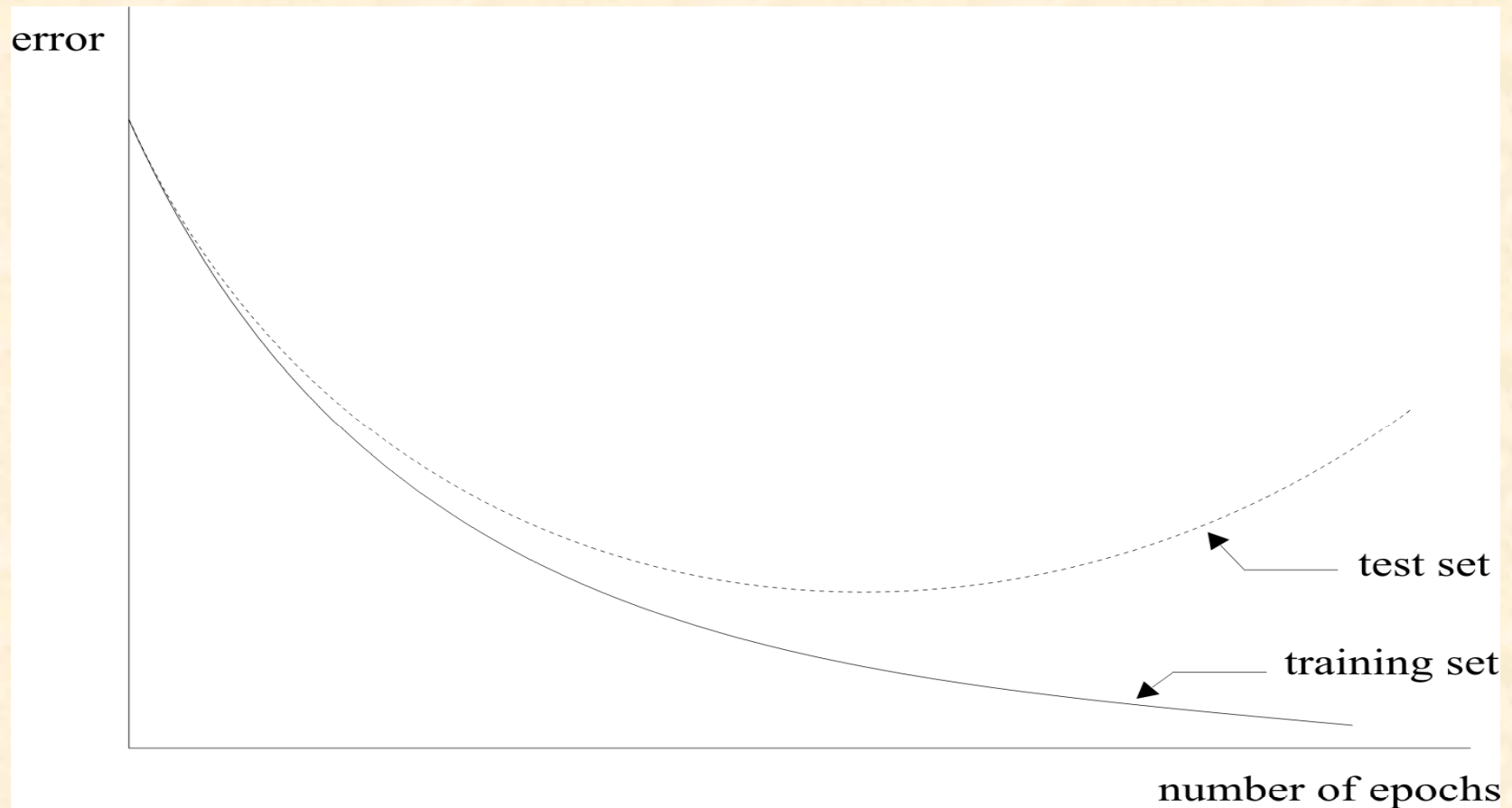


Example:





- **Overtraining** is another side of the same coin, i.e., the network adapts to the peculiarities of the training set.





❖ Generalized Linear Classifiers

- Remember the XOR problem. The mapping

$$\underline{x} \rightarrow \underline{y} = \begin{bmatrix} f(g_1(\underline{x})) \\ f(g_2(\underline{x})) \end{bmatrix}$$

$f(\cdot) \rightarrow$ The activation function transforms the nonlinear task into a linear one.

- In the more general case:
- Let $\underline{x} \in R^l$ and a nonlinear classification task.

$$f_i(\cdot), i = 1, 2, \dots, k$$



- Are there any functions and an appropriate k , so that the mapping

$$\underline{x} \rightarrow \underline{y} = \begin{bmatrix} f_1(\underline{x}) \\ \dots \\ f_k(\underline{x}) \end{bmatrix}$$

transforms the task into a **linear one**, in the $\underline{y} \in R^k$ space?

- If this is true, then there exists a hyperplane $\underline{w} \in R^k$ so that

$$\text{If } w_0 + \underline{w}^T \underline{y} > 0, \quad \underline{x} \in \omega_1$$

$$w_0 + \underline{w}^T \underline{y} < 0, \quad \underline{x} \in \omega_2$$



- In such a case this is equivalent with approximating the nonlinear discriminant function $g(\underline{x})$, in terms of $f_i(\underline{x})$, i.e.,

$$g(\underline{x}) \cong w_0 + \sum_{i=1}^k w_i f_i(\underline{x}) \quad (><) \quad 0$$

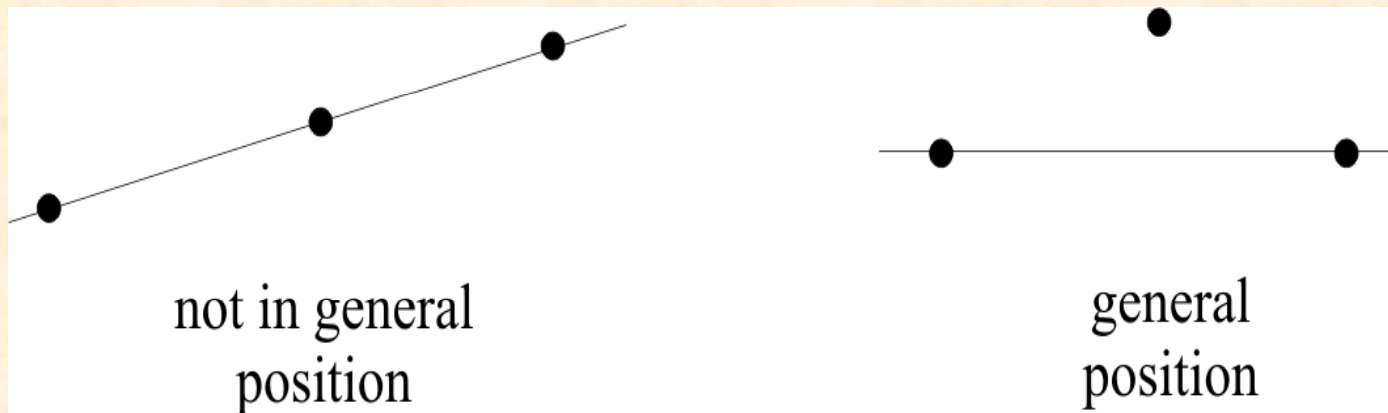
- Given $f_i(\underline{x})$, the task of computing the weights is a **linear** one.
- How sensible is this??
- From the numerical analysis point of view, this is justified if $f_i(\underline{x})$ are interpolation functions.
 - From the Pattern Recognition point of view, this is justified by Cover's theorem



❖ Capacity of the l -dimensional space in Linear Dichotomies

- Assume N points in R^l assumed to be in **general position**, that is:

Not $l + 1$ of these lie on a $l - 1$ dimensional space

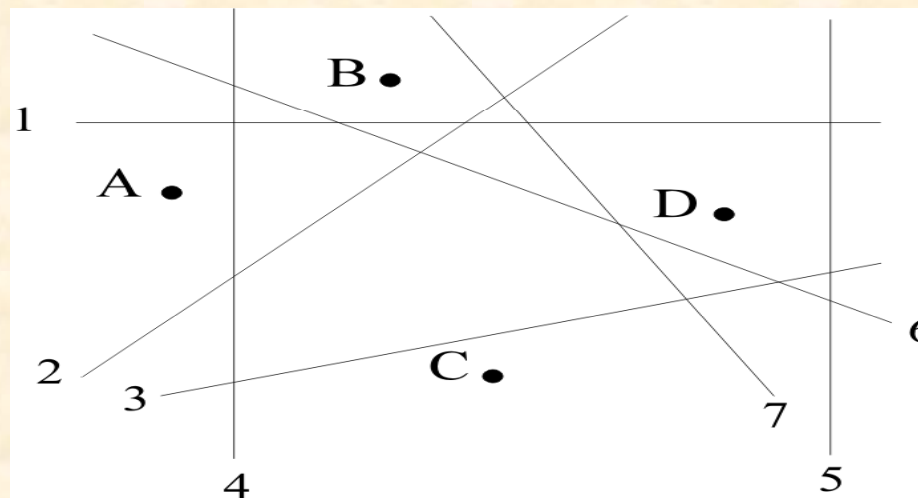




- **Cover's theorem** states: The number of groupings that can be formed by $(l-1)$ -dimensional **hyperplanes** to separate N points in two classes is

$$O(N, l) = 2 \sum_{i=0}^l \binom{N-1}{i}, \quad \binom{N-1}{i} = \frac{(N-1)!}{(N-1-i)!i!}$$

Example: $N=4, l=2, O(4,2)=14$



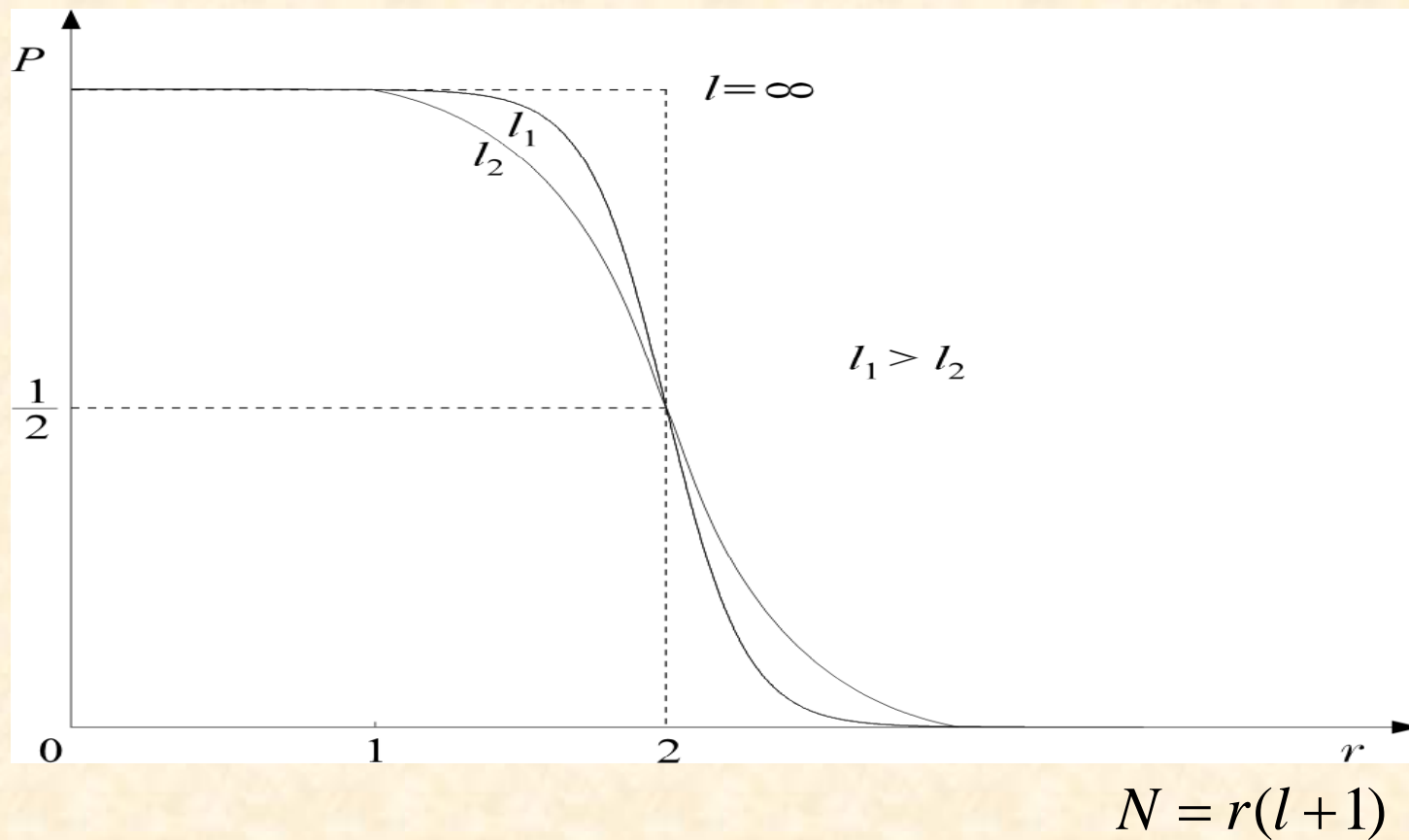
Notice: The total number of possible groupings is

$2^4=16$
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών



- Probability of grouping N points in two linearly separable classes is

$$\frac{O(N, l)}{2^N} = P_N^l$$





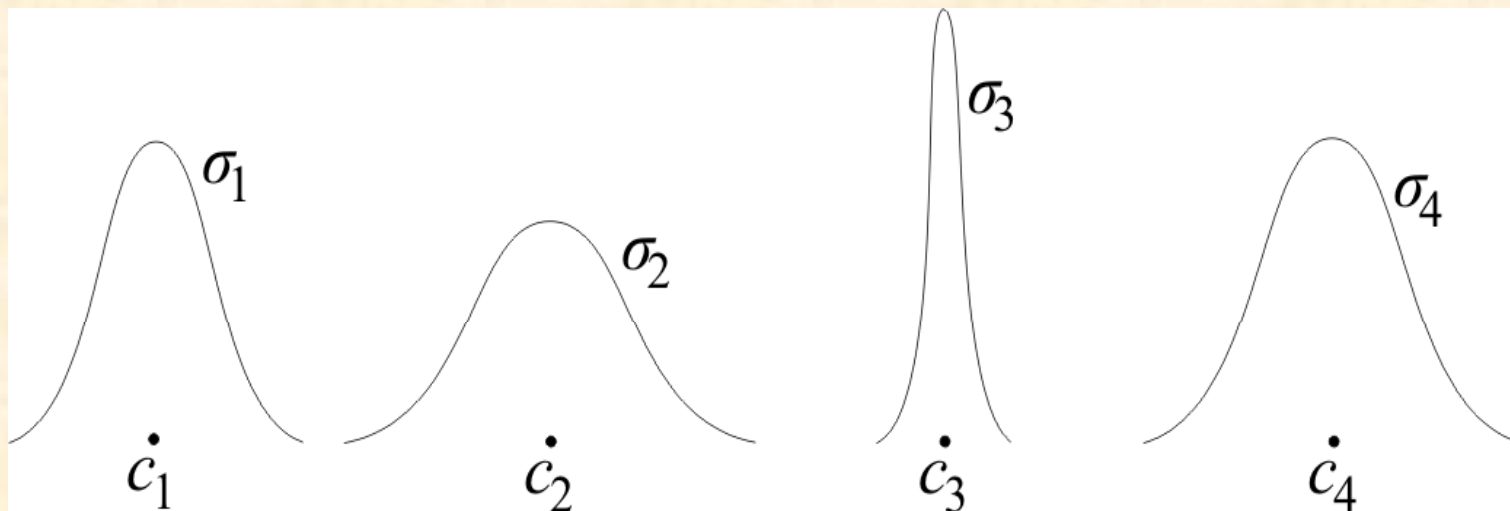
Thus, the probability of having N points in **linearly** separable classes tends to 1, for **large** l , **provided** $N < 2(l+1)$

Hence, by mapping to a higher dimensional space, we **increase the probability of linear separability**, provided the space is not too densely populated.



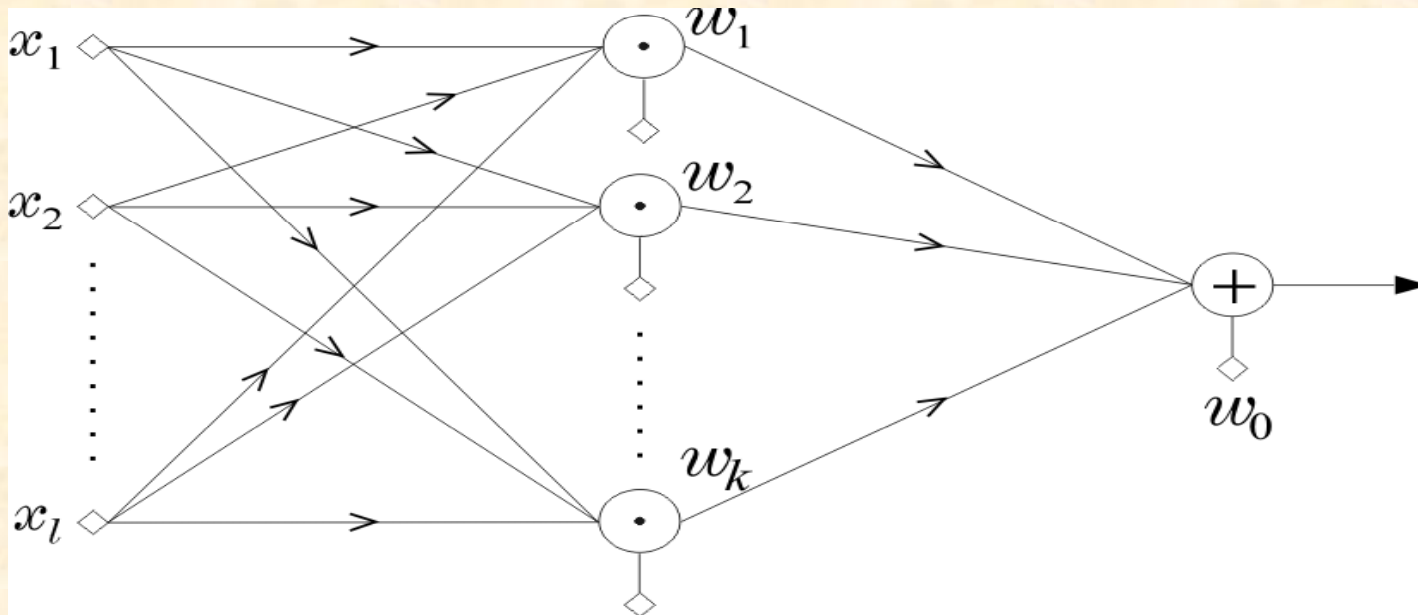
❖ Radial Basis Function Networks (RBF)

➤ Choose





$$f_i(\underline{x}) = \exp\left(-\frac{\|\underline{x} - \underline{c}_i\|^2}{2\sigma_i^2}\right)$$



Equivalent to a single layer network, with RBF activations and linear output node.



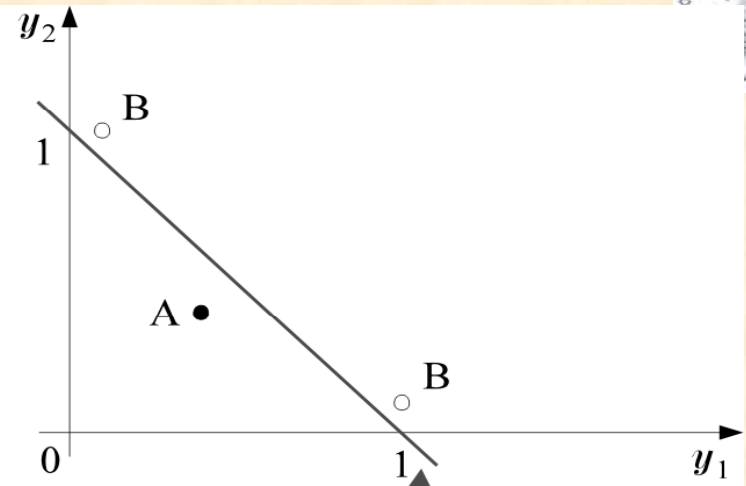
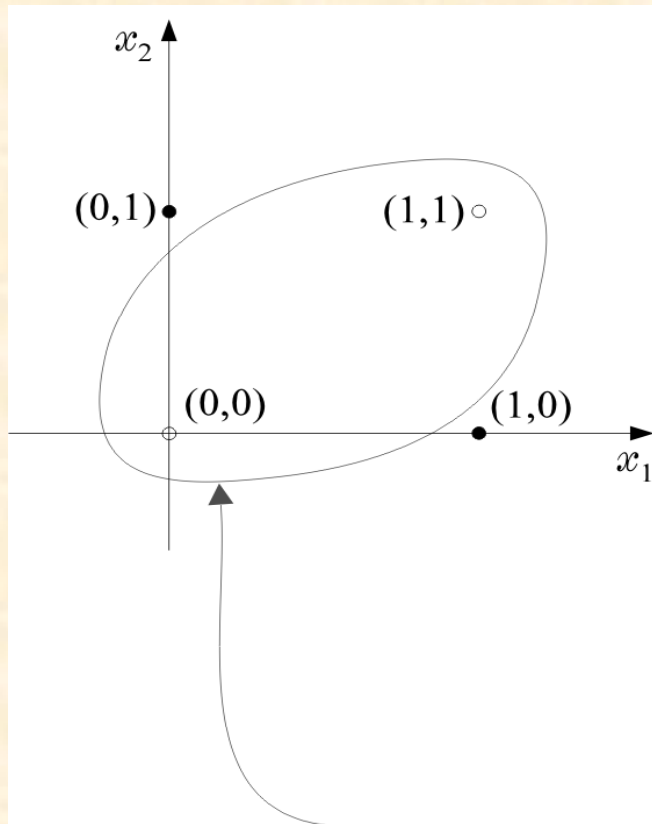
➤ Example: The XOR problem

- Define:

$$\underline{c}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \underline{c}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \sigma_1 = \sigma_2 = \frac{1}{\sqrt{2}}$$

$$\underline{y} = \begin{bmatrix} \exp(-\|\underline{x} - \underline{c}_1\|^2) \\ \exp(-\|\underline{x} - \underline{c}_2\|^2) \end{bmatrix}$$

- $\begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0.135 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 0.135 \end{bmatrix}$
 $\begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0.368 \\ 0.368 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.368 \\ 0.368 \end{bmatrix}$



$$g(\underline{y}) = y_1 + y_2 - 1 = 0$$

$$g(\underline{x}) = \exp(-\|\underline{x} - \underline{c}_1\|^2) + \exp(-\|\underline{x} - \underline{c}_2\|^2) - 1 = 0$$



➤ Training of the RBF networks

- Fixed centers: Choose centers randomly among the data points. Also fix σ_i 's. Then

$$g(\underline{x}) = w_0 + \underline{w}^T \underline{y}$$

is a typical linear classifier design.

- Training of the centers: This is a **nonlinear** optimization task
- Combine supervised and unsupervised learning procedures.
- The unsupervised part reveals clustering tendencies of the data and assigns the centers at the cluster representatives.



❖ Universal Approximators

It has been shown that any nonlinear continuous function can be approximated **arbitrarily close**, both, by a two layer perceptron, with sigmoid activations, and an RBF network, provided a **large enough** number of nodes is used.

❖ Multilayer Perceptrons vs. RBF networks

- MLP's involve activations of global nature. All points on a plane $w^T \underline{x} = c$ give the same response.
- RBF networks have activations of a local nature, due to the exponential decrease as one moves away from the centers.
- MLP's learn slower but have better generalization properties



❖ Support Vector Machines: The non-linear case

- Recall that the probability of having **linearly separable classes increases** as the **dimensionality** of the feature vectors **increases**. Assume the mapping:

$$\underline{x} \in R^l \rightarrow \underline{y} \in R^k, k > l$$

Then use SVM in R^k

- Recall that in this case the dual problem formulation will be

$$\underset{\underline{\lambda}}{\text{maximize}} \left(\sum_{i=1}^N \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j \underline{y}_i^T \underline{y}_j \right)$$

$$\text{where } \underline{y}_i \in R^k$$



Also, the classifier will be

$$\begin{aligned}g(\underline{y}) &= \underline{w}^T \underline{y} + w_0 \\ &= \sum_{i=1}^{N_s} \lambda_i y_i \underline{y}_i \underline{y}\end{aligned}$$

where $\underline{x} \rightarrow \underline{y} \in R^k$

Thus, inner products in a high dimensional space are involved, hence

- High complexity



- Something clever: Compute the inner products in the **high** dimensional space as functions of inner products performed in the **low** dimensional space!!!
- Is this POSSIBLE?? Yes. Here is an example

$$\text{Let } \underline{x} = [x_1, x_2]^T \in R^2$$

$$\text{Let } \underline{x} \rightarrow \underline{y} = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \in R^3$$

Then, it is easy to show that

$$\underline{y}_i^T \underline{y}_j = (\underline{x}_i^T \underline{x}_j)^2$$



➤ Mercer's Theorem

Let $\underline{x} \rightarrow \underline{\Phi}(\underline{x}) \in H$

Then, the inner product in H

$$\sum_r \Phi_r(\underline{x})\Phi_r(\underline{y}) = K(\underline{x}, \underline{y})$$

where

$$\int K(\underline{x}, \underline{y}) g(\underline{x}) g(\underline{y}) d\underline{x} d\underline{y} \geq 0$$

for **any** $g(\underline{x})$, \underline{x} :

$$\int g^2(\underline{x}) d\underline{x} < +\infty$$

$K(\underline{x}, \underline{y})$ **symmetric** function known as **kernel**.



- The opposite is also true. Any kernel, with the above properties, corresponds to an inner product in **SOME** space!!!

- Examples of kernels

- Radial basis Functions:

$$K(\underline{x}, \underline{z}) = \exp\left(-\frac{\|\underline{x} - \underline{z}\|^2}{\sigma^2}\right)$$

- Polynomial:

$$K(\underline{x}, \underline{z}) = (\underline{x}^T \underline{z} + 1)^q, \quad q > 0$$

- Hyperbolic Tangent:

$$K(\underline{x}, \underline{z}) = \tanh(\beta \underline{x}^T \underline{z} + \gamma)$$

for appropriate values of β, γ .



➤ SVM Formulation

- Step 1: Choose appropriate kernel. This implicitly assumes a mapping to a higher dimensional (yet, not known) space.

- Step 2:
$$\max_{\underline{\lambda}} \left(\sum_i \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j K(\underline{x}_i, \underline{x}_j) \right)$$

subject to: $0 \leq \lambda_i \leq C, i = 1, 2, \dots, N$

$$\sum_i \lambda_i y_i = 0$$

This results to an **implicit** combination

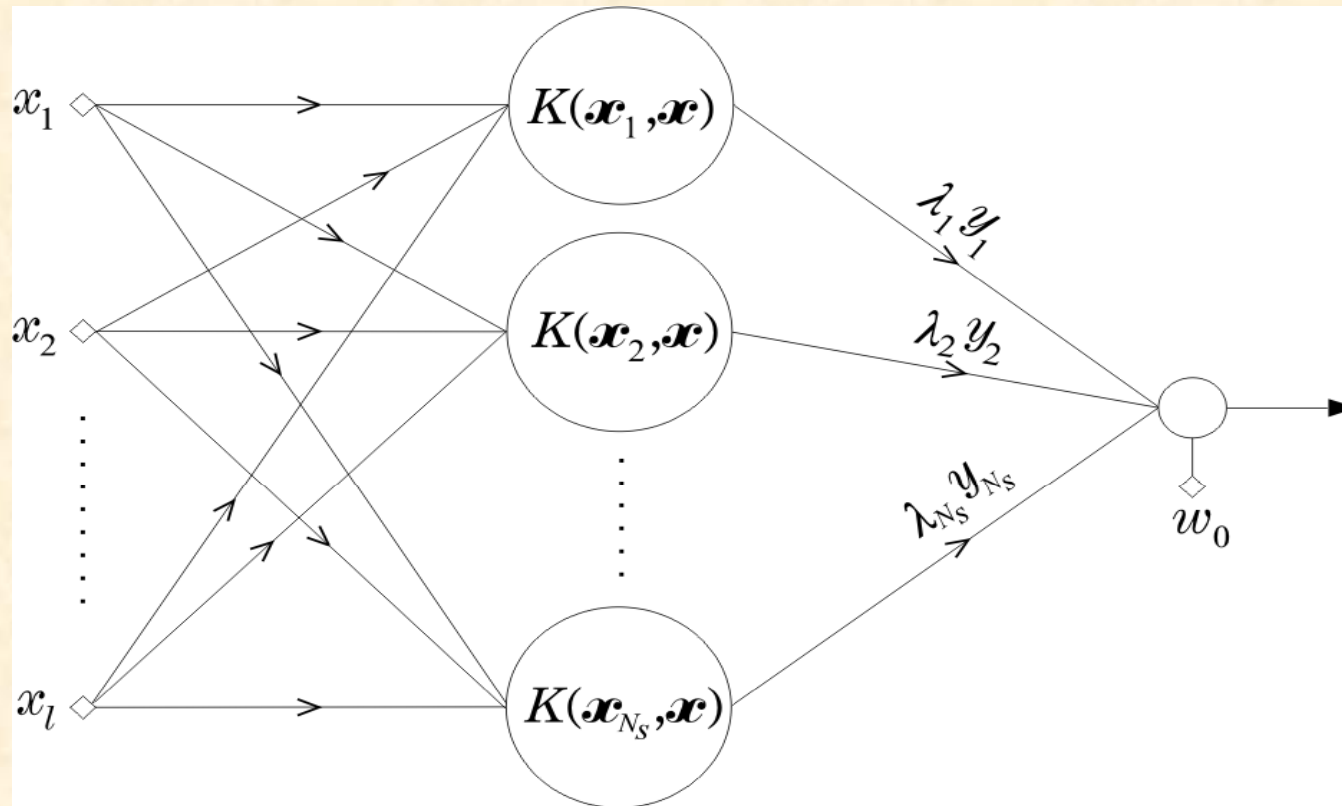
$$\underline{w} = \sum_{i=1}^{N_s} \lambda_i y_i \underline{\varphi}(\underline{x}_i)$$

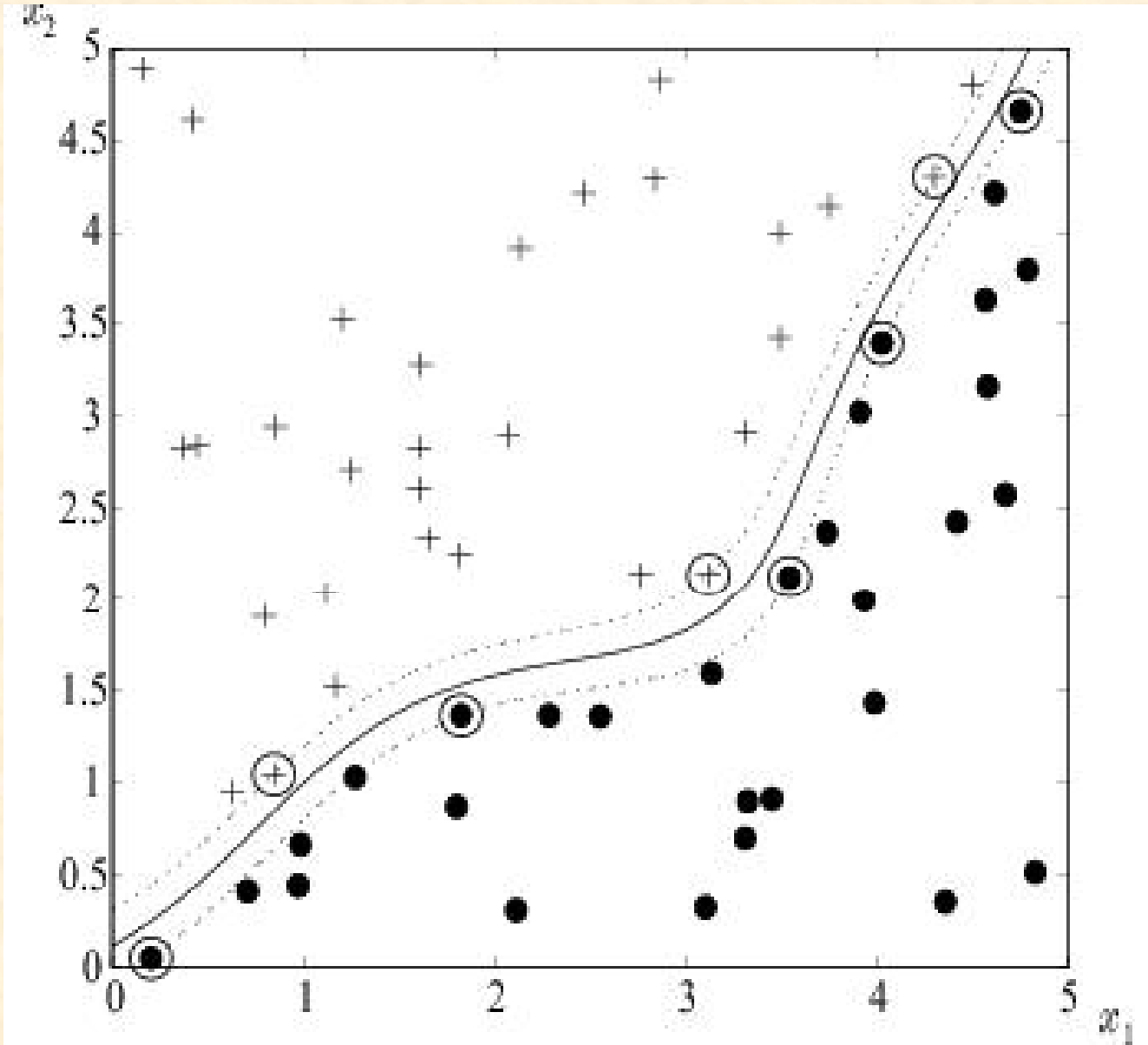


- Step 3: Assign \underline{x} to

$$\omega_1(\omega_2) \text{ if } g(\underline{x}) = \sum_{i=1}^{N_s} \lambda_i y_i K(\underline{x}_i, \underline{x}) + w_0 > (<) 0$$

- The SVM Architecture







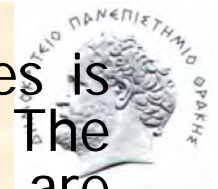
❖ Decision Trees

This is a family of non-linear classifiers. They are **multistage** decision systems, in which classes are **sequentially** rejected, until a finally accepted class is reached. To this end:

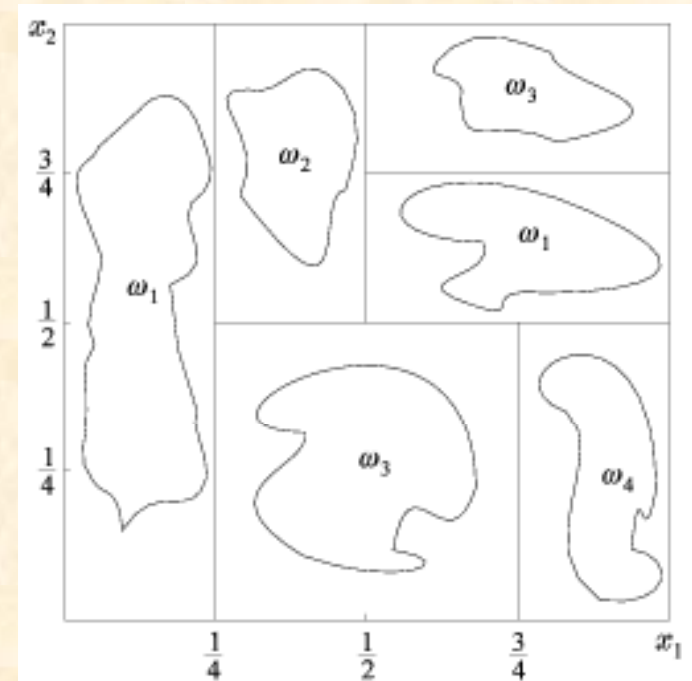
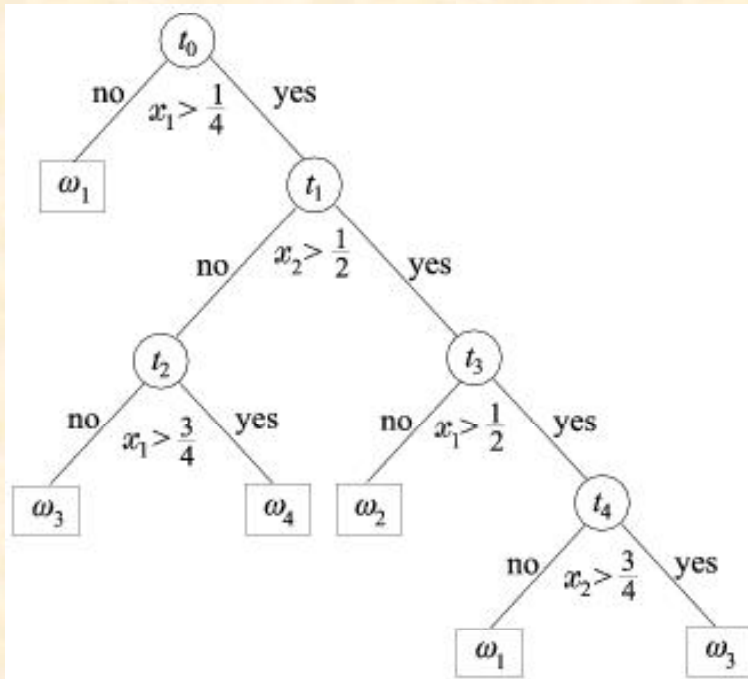
- The feature space is split into **unique** regions in a sequential manner.
- Upon the arrival of a feature vector, sequential decisions, assigning features to specific regions, are performed along a path of **nodes** of an appropriately constructed **tree**.
- The sequence of decisions is applied to **individual** features, and the queries performed in each node are of the **type**:

$$\text{is feature } x_i \leq a$$

where a is a pre-chosen (during training) threshold.



- The figures below are such examples. This type of trees is known as **Ordinary Binary Classification Trees (OBCT)**. The decision hyperplanes, splitting the space into regions, are parallel to the axis of the spaces. Other types of partition are also possible, yet less popular.





➤ Design Elements that define a decision tree.

- Each node, t , is associated with a subset $X_t \subseteq X$, where X is the training set. At each node, X_t is split into **two** (binary splits) **disjoint descendant** subsets $X_{t,Y}$ and $X_{t,N}$, where

$$X_{t,Y} \cap X_{t,N} = \emptyset$$

$$X_{t,Y} \cup X_{t,N} = X_t$$

$X_{t,Y}$ is the subset of X_t for which the answer to the query at node t is **YES**. $X_{t,N}$ is the subset corresponding to **NO**. The split is decided according to an **adopted question** (query).



- A **splitting** criterion must be adopted for the **best** split of X_t into $X_{t,Y}$ and $X_{t,N}$.
- A **stop-splitting** criterion must be adopted that controls the growth of the tree and a node is declared as **terminal (leaf)**.
- A rule is required that assigns each (terminal) leaf to a class.



- **Set of Questions:** In OBCT trees the set of questions is of the type

$$\text{is } x_i \leq a \text{ ?}$$

The choice of the specific x_i and the value of the threshold a , for each node t , are the results of searching, during training, among the features and a set of possible threshold values. The final combination is the one that results to the **best value** of a criterion.



- **Splitting Criterion:** The main idea behind splitting at each node is the resulting descendant subsets $X_{t,Y}$ and $X_{t,N}$ to be more **class homogeneous** compared to X_t . Thus the criterion must be in harmony with such a goal. A commonly used criterion is the **node impurity**:

$$I(t) = - \sum_{i=1}^M P(\omega_i | t) \log_2 P(\omega_i | t)$$

and

$$P(\omega_i | t) \approx \frac{N_t^i}{N_t}$$

where N_t^i is the number of data points in X_t that belong to class ω_i . The **decrease in node impurity** is defined as:

$$\Delta I(t) = I(t) - \frac{N_{t,Y}}{N_t} I(t_Y) - \frac{N_{t,N}}{N_t} I(t_N)$$



- The goal is to choose the parameters in each node (feature and threshold) that result in **a split with the highest decrease in impurity**.
- Why highest decrease? Observe that the highest value of $I(t)$ is achieved if all classes are **equiprobable**, i.e., X_t is the **least** homogenous.
- Stop - splitting rule. Adopt a threshold T and stop splitting a node (i.e., assign it as a **leaf**), if the impurity decrease is less than T . That is, node t is **"pure enough"**.
- Class Assignment Rule: Assign a leaf to a class ω_j , where:

$$j = \arg \max_i P(\omega_i | t)$$



➤ Summary of an OBCT algorithmic scheme:

- Begin with the root node, i.e., $X_t = X$
- For each new node t
 - * For every feature $x_k, k = 1, 2, \dots, l$
 - For every value $\alpha_{kn}, n = 1, 2, \dots, N_{tk}$
 - Generate X_{tY} and X_{tN} according to the answer in the question: is $x_k(i) \leq \alpha_{kn}, i = 1, 2, \dots, N_t$
 - Compute the impurity decrease
 - End
 - Choose α_{kn_0} leading to the maximum decrease w.r. to x_k
 - * End
 - * Choose x_{k_0} and associated $\alpha_{k_0n_0}$ leading to the overall maximum decrease of impurity
 - * If stop-splitting rule is met declare node t as a leaf and designate it with a class label
 - * If not, generate two descendant nodes t_Y and t_N with associated subsets X_{tY} and X_{tN} , depending on the answer to the question: is $x_{k_0} \leq \alpha_{k_0n_0}$
- End



➤ Remarks:

- A critical factor in the design is the size of the tree. Usually one grows a tree to a large size and then applies various **pruning** techniques.
- Decision trees belong to the class of **unstable** classifiers. This can be overcome by a number of “averaging” techniques. **Bagging** is a popular technique. Using **bootstrap** techniques in X , various trees are constructed, T_i , $i=1, 2, \dots, B$. The decision is taken according to a **majority voting** rule.



❖ Combining Classifiers

The basic philosophy behind the combination of different classifiers lies in the fact that even the “best” classifier fails in some patterns that other classifiers may classify correctly. Combining classifiers aims at exploiting this **complementary information** residing in the various classifiers.

Thus, one designs different optimal classifiers and then combines the results with a specific rule.

- Assume that each of the, say, L designed classifiers provides at its output the posterior probabilities:

$$P(\omega_i | \underline{x}), i = 1, 2, \dots, M$$



- **Product Rule:** Assign \underline{x} to the class ω_i :

$$i = \arg \max_k \prod_{j=1}^L P_j(\omega_k | \underline{x})$$

where $P_j(\omega_k | \underline{x})$ is the respective posterior probability of the j^{th} classifier.

- **Sum Rule:** Assign \underline{x} to the class : ω_i

$$i = \arg \max_k \sum_{j=1}^L P_j(\omega_k | \underline{x})$$



- **Majority Voting Rule:** Assign \underline{x} to the class for which there is a consensus or when at least ℓ_c of the classifiers agree on the class label of \underline{x} where:

$$\ell_c = \begin{cases} \frac{L}{2} + 1, & L \text{ even} \\ \frac{L+1}{2}, & L \text{ odd} \end{cases}$$

otherwise the decision is **rejection**, that is **no decision** is taken.

Thus, correct decision is made if the majority of the classifiers agree on the correct label, and wrong decision if the majority agrees in the wrong label.



➤ Dependent or not Dependent classifiers?

- Although there are not general theoretical results, experimental evidence has shown that the more independent in their decision the classifiers are, the higher the expectation should be for obtaining improved results after combination. However, there is **no guarantee** that combining classifiers results in **better** performance compared to the **"best" one among the classifiers**.

➤ Towards Independence: A number of Scenarios.

- Train the individual classifiers using different training data points. To this end, choose among a number of possibilities:
 - **Bootstrapping**: This is a popular technique to combine unstable classifiers such as decision trees (Bagging belongs to this category of combination).



- **Stacking**: Train the combiner with data points that have been **excluded** from the set used to train the individual classifiers.
- **Use different subspaces to train individual classifiers**: According to the method, each individual classifier operates in a different feature subspace. That is, use **different features** for each classifier.

➤ **Remarks:**

- The majority voting and the summation schemes rank among the most popular combination schemes.
- Training individual classifiers in different subspaces seems to lead to substantially better improvements compared to classifiers operating in the same subspace.
- Besides the above three rules, other alternatives are also possible, such as to use the median value of the outputs of individual classifiers.



❖ The Boosting Approach

- The origins: Is it possible a **weak** learning algorithm (one that performs slightly better than a random guessing) to be **boosted into a strong** algorithm? (Villiant 1984).

- The procedure to achieve it:
 - Adopt a weak classifier known as the **base** classifier.
 - Employing the base classifier, design a series of classifiers, in a **hierarchical fashion**, each time employing a different weighting of the training samples. Emphasis in the weighting is given on the **hardest** samples, i.e., the ones that keep “failing”.
 - Combine the hierarchically designed classifiers by a weighted average procedure.



➤ The AdaBoost Algorithm.

Construct an optimally designed classifier of the form:

$$f(\underline{x}) = \text{sign}\{F(\underline{x})\}$$

where:

$$F(\underline{x}) = \sum_{k=1}^K a_k \varphi(\underline{x}; \underline{\mathcal{G}}_k)$$

where $\varphi(\underline{x}; \underline{\mathcal{G}}_k)$ denotes the base classifier that returns a binary class label:

$$\varphi(\underline{x}; \underline{\mathcal{G}}_k) \in \{-1, 1\}$$

$\underline{\mathcal{G}}$ is a parameter vector.



- The essence of the method.

Design the series of classifiers:

$$\varphi(\underline{x}; \underline{\mathcal{G}}_1), \varphi(\underline{x}; \underline{\mathcal{G}}_2), \dots, \varphi(\underline{x}; \underline{\mathcal{G}}_k)$$

The parameter vectors

$$\underline{\mathcal{G}}_k, k = 1, 2, \dots, K$$

are optimally computed so as:

- To minimize the error rate on the **training** set.
- Each time, the training samples are re-weighted so that the weight of each sample depends on its history. **Hard** samples that **"insist" on failing** to be predicted correctly, by the previously designed classifiers, are **more heavily weighted**.



- Updating the weights for each sample $\underline{x}_i, i = 1, 2, \dots, N$

$$w_i^{(m+1)} = \frac{w_i^m \exp(-y_i a_m \varphi(\underline{x}_i; \underline{\mathcal{G}}_m))}{Z_m}$$

- Z_m is a normalizing factor common for all samples.

- $a_m = \frac{1}{2} \ln \frac{1-P_m}{P_m}$

where $P_m < 0.5$ (by assumption) is the error rate of the optimal classifier $\varphi(\underline{x}; \underline{\mathcal{G}}_m)$ at stage m . Thus $a_m > 0$.

- The term: $\exp(-y_i a_m \varphi(\underline{x}_i; \underline{\mathcal{G}}_m))$

takes a large value if $y_i \varphi(\underline{x}_i; \underline{\mathcal{G}}_m) < 0$ (wrong classification) and a small value in the case of correct classification $\{y_i \varphi(\underline{x}_i; \underline{\mathcal{G}}_m) > 0\}$

- The update equation is of a **multiplicative** nature. That is, successive large values of weights (hard samples) result in larger weight for the next iteration



- The algorithm

- Initialize: $w_i^{(1)} = \frac{1}{N}$, $i = 1, 2, \dots, N$
- Initialize: $m = 1$
- Repeat
 - Compute optimum θ_m in $\phi(\cdot; \theta_m)$ by minimizing P_m
 - Compute the optimum P_m
 - $\alpha_m = \frac{1}{2} \ln \frac{1-P_m}{P_m}$
 - $Z_m = 0.0$
 - For $i = 1$ to N
 - * $w_i^{(m+1)} = w_i^{(m)} \exp(-y_i \alpha_m \phi(x_i; \theta_m))$
 - * $Z_m = Z_m + w_i^{(m+1)}$
 - End{For}
 - For $i = 1$ to N
 - * $w_i^{(m+1)} = w_i^{(m+1)} / Z_m$
 - End {For}
 - $K = m$
 - $m = m + 1$
- Until a termination criterion is met.
- $f(\cdot) = \text{sign}(\sum_{k=1}^K \alpha_k \phi(\cdot, \theta_k))$



➤ Remarks:

- Training error rate tends to **zero** after a few iterations. The test error levels to some value.
- AdaBoost is **greedy** in reducing the **margin** that samples leave from the decision surface.

