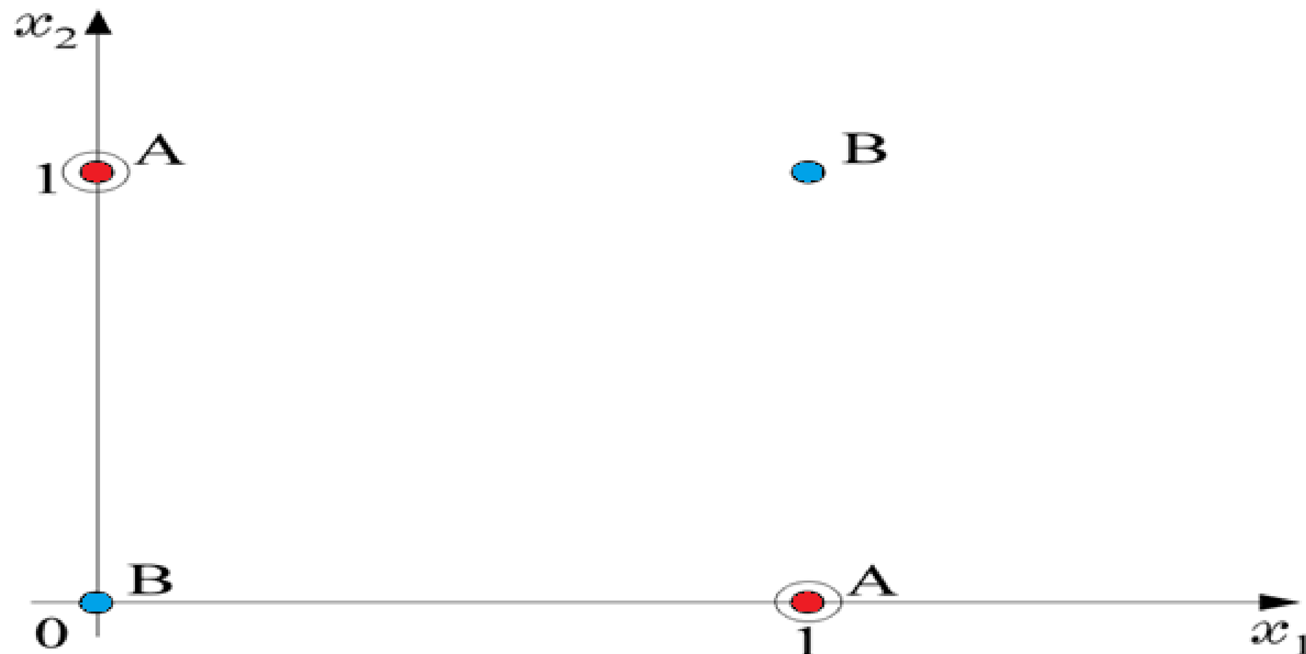
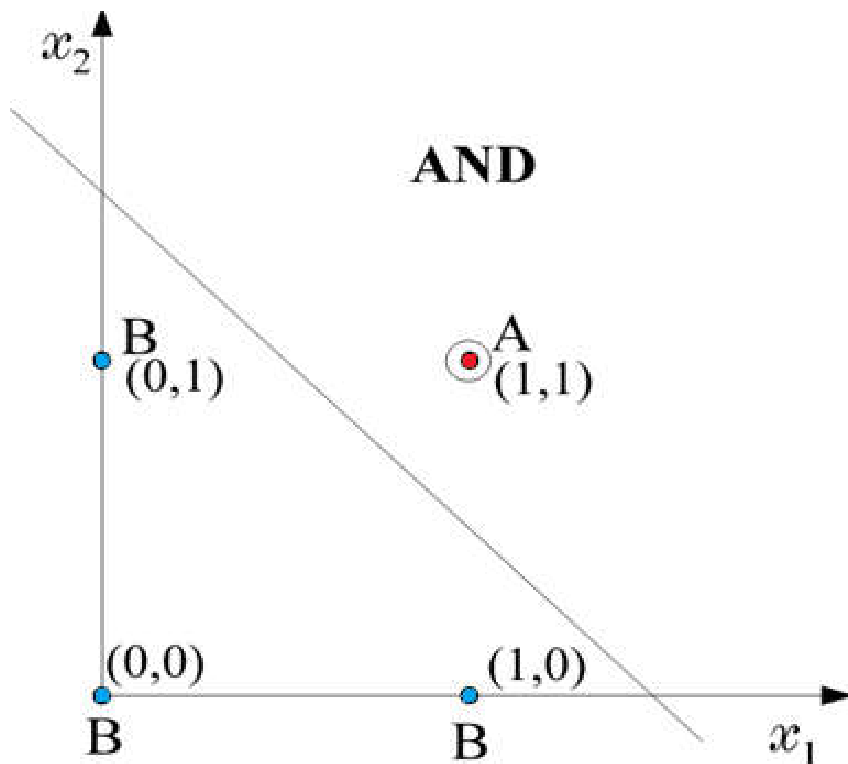
Ch 4: Non Linear Classifiers

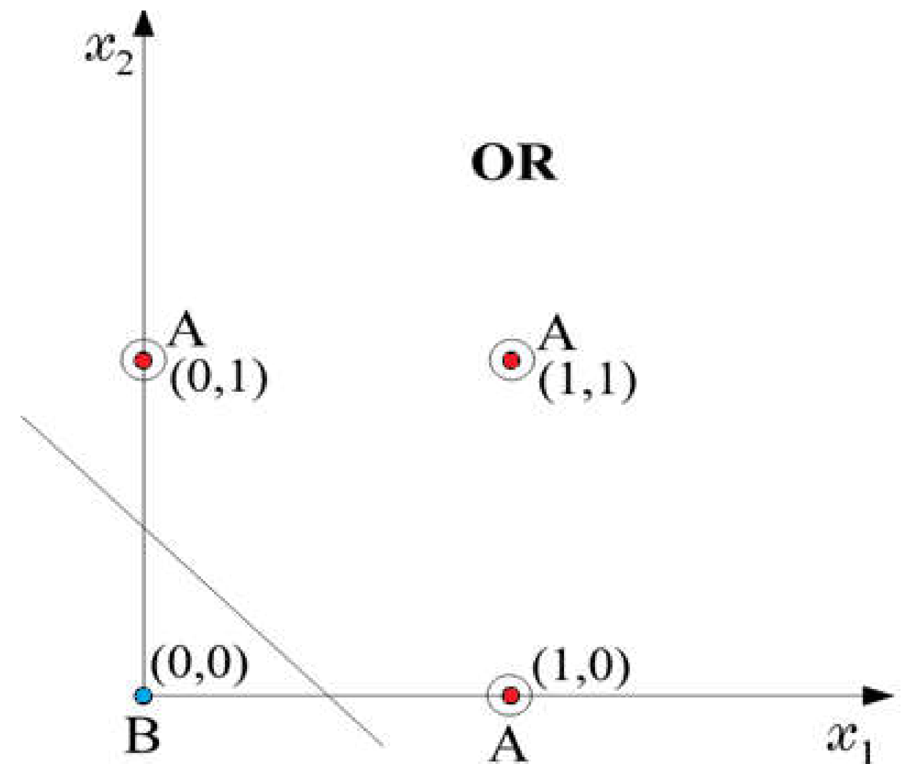
❖ The XOR problem

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



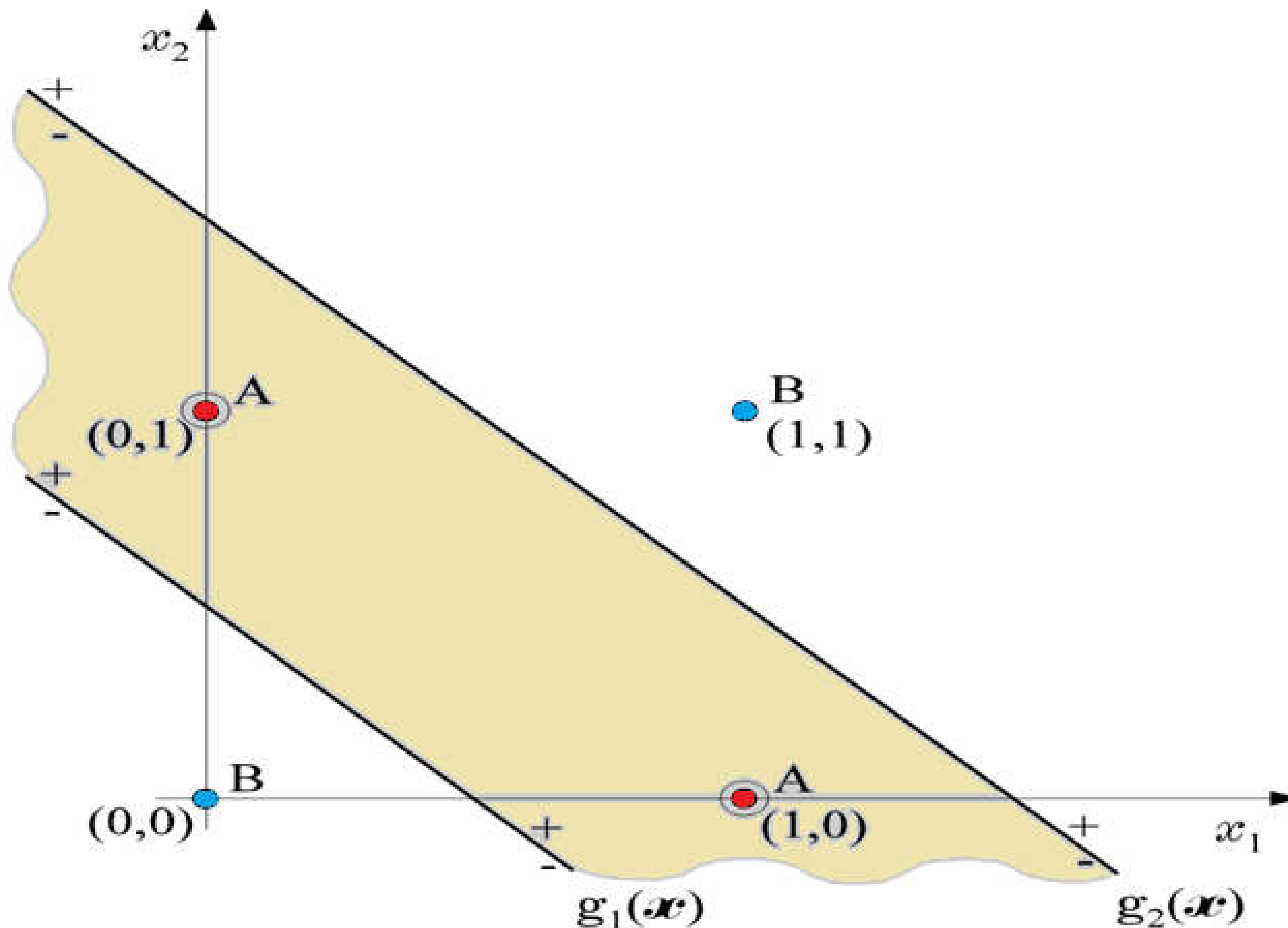
❖ There is no single line (hyperplane) that separates class A from class B. On the contrary, AND and OR operations are linearly separable problems





❖ 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

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

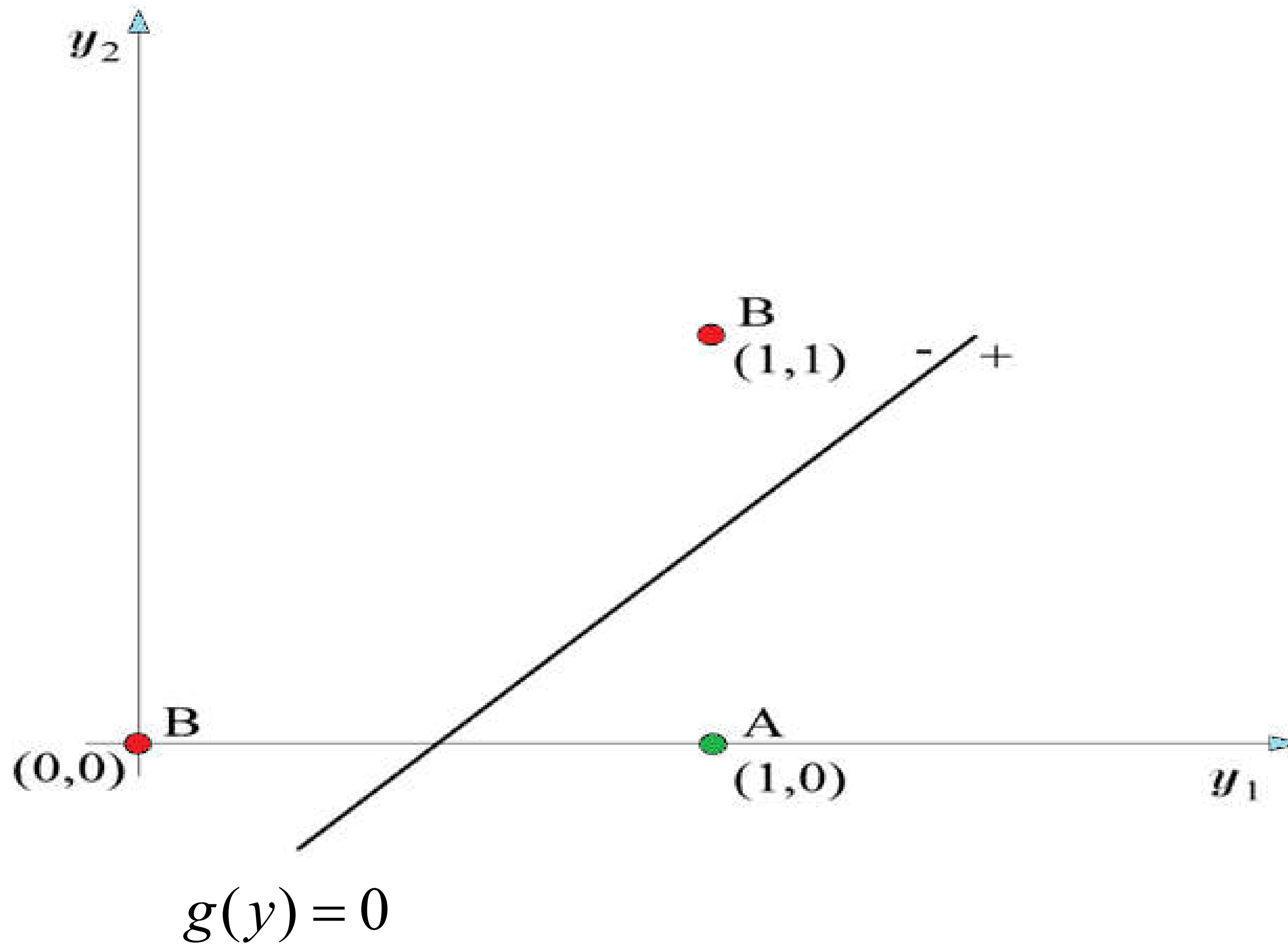
    depending on the position of $\underline{x}$.

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

1st phase				2nd 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 $\underline{x} \rightarrow \underline{y} = [y_1, y_2]^T$

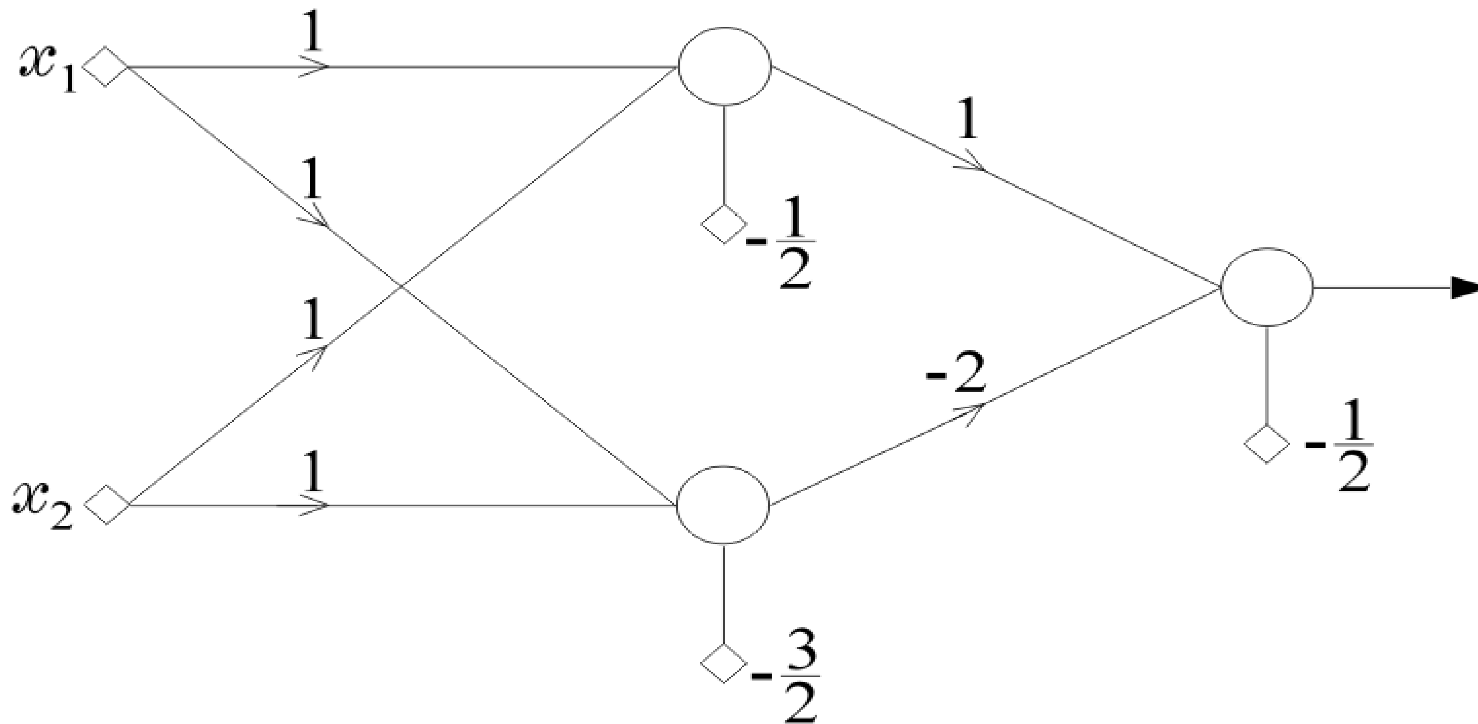
The decision is now performed on the transformed $\underline{y}$ data.



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)

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

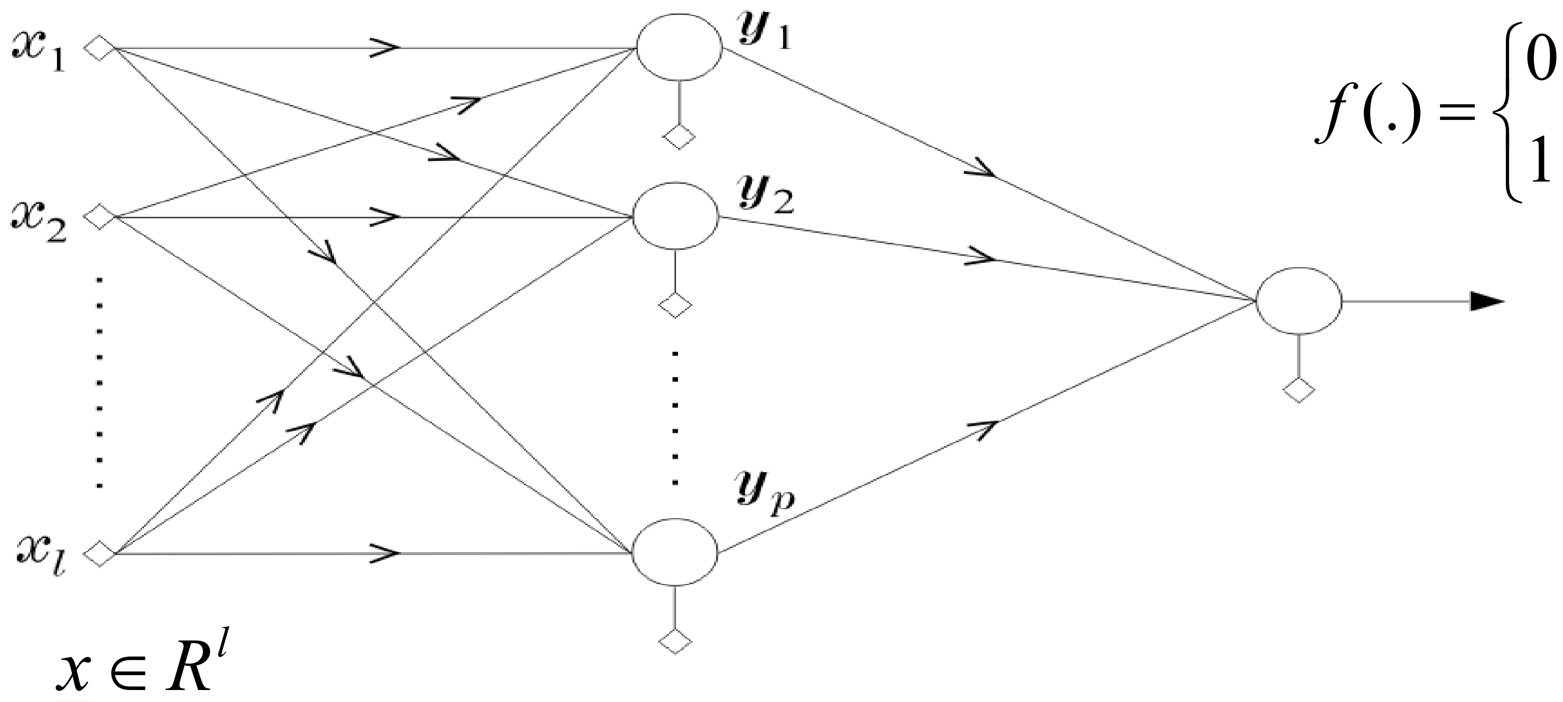
$$g_2(\underline{x}) = x_1 + x_2 - \frac{3}{2} = 0$$

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

❖ Classification capabilities of the two-layer perceptron

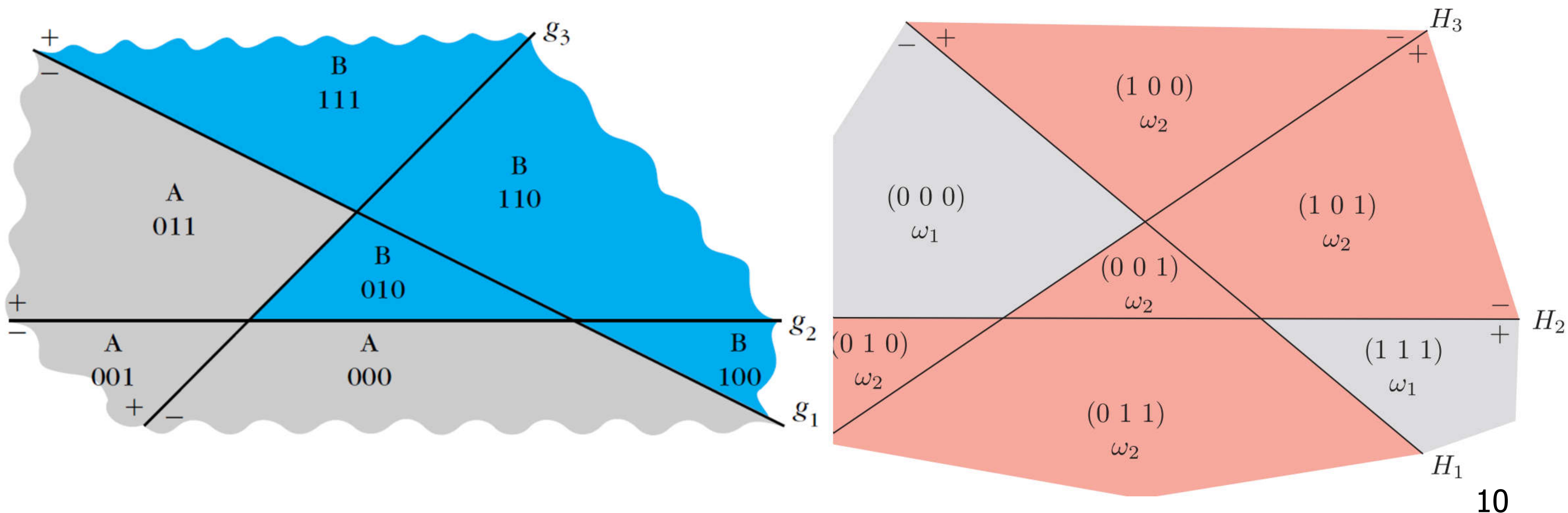
- ➢ The mapping performed by the first layer neurons is onto the vertices of the unit side square, e.g., (0, 0), (0, 1), (1, 0), (1, 1).

- ➢ The more general case,



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

- 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.

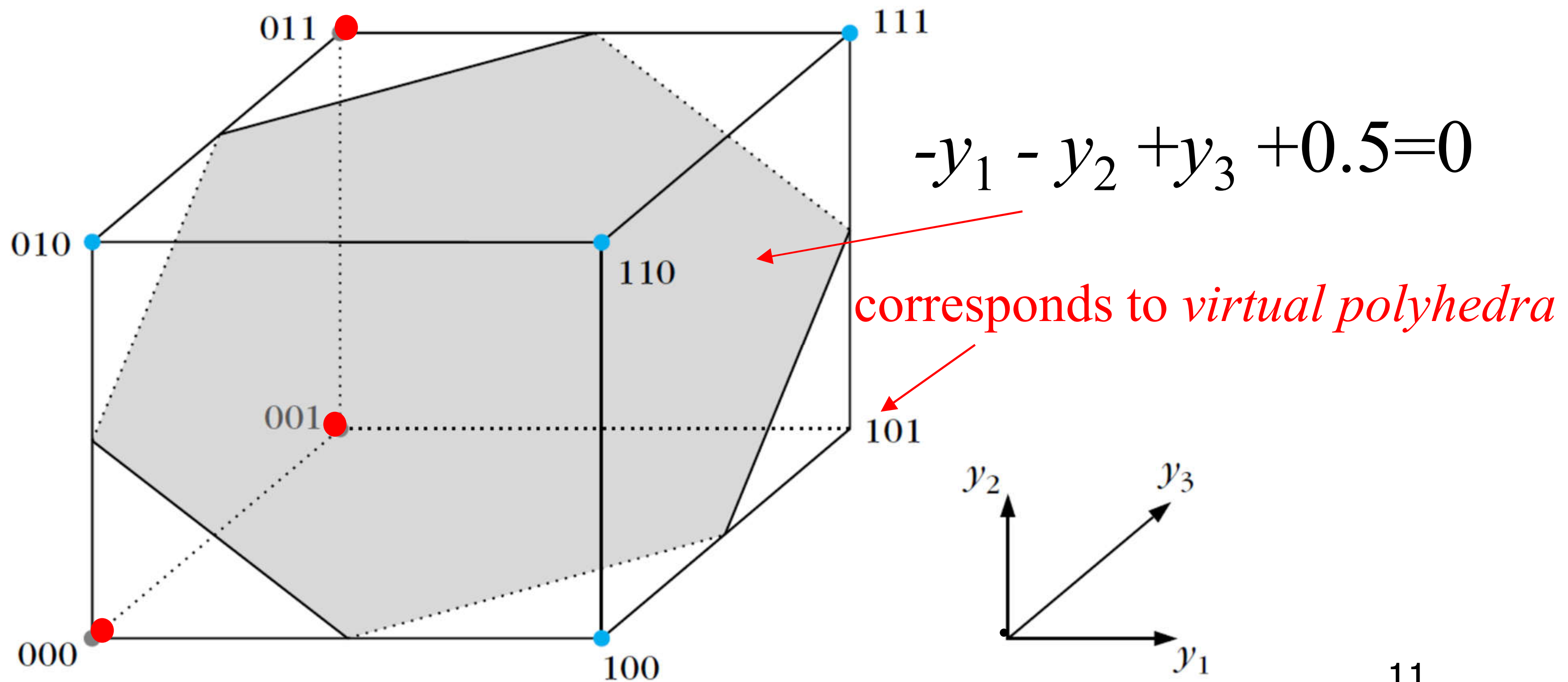


For example, the 001 vertex corresponds to the region which is located

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

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

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

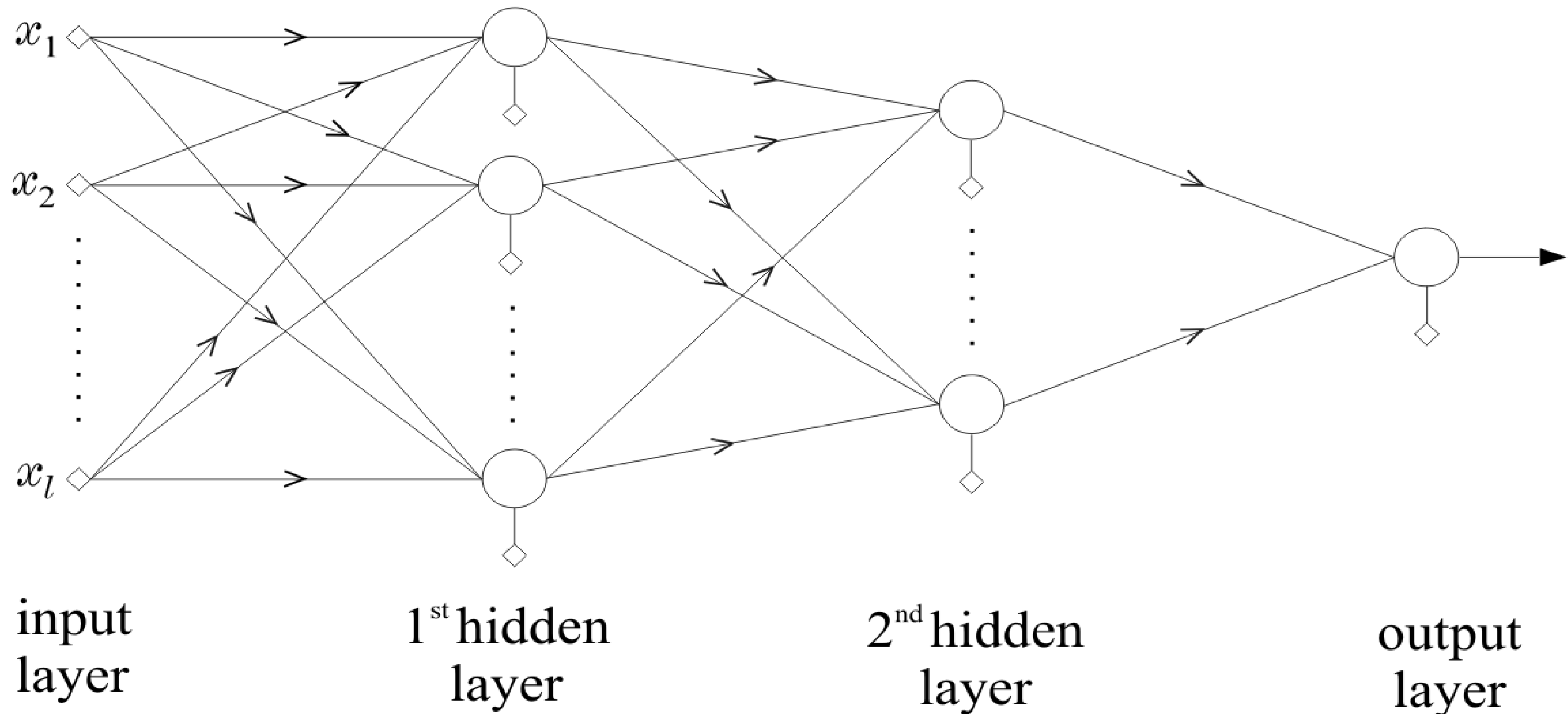


- The output neuron subsequently realizes another hyperplane, which separates the hypercube into two parts, having some of its vertices on one and some on the other side.

- The output $\underline{y}$ neuron realizes a hyperplane in the transformed 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.
- A three-layer perceptron architecture can separate classes resulting from **ANY** union of polyhedral regions.

❖ 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.

- 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.

❖ 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.,

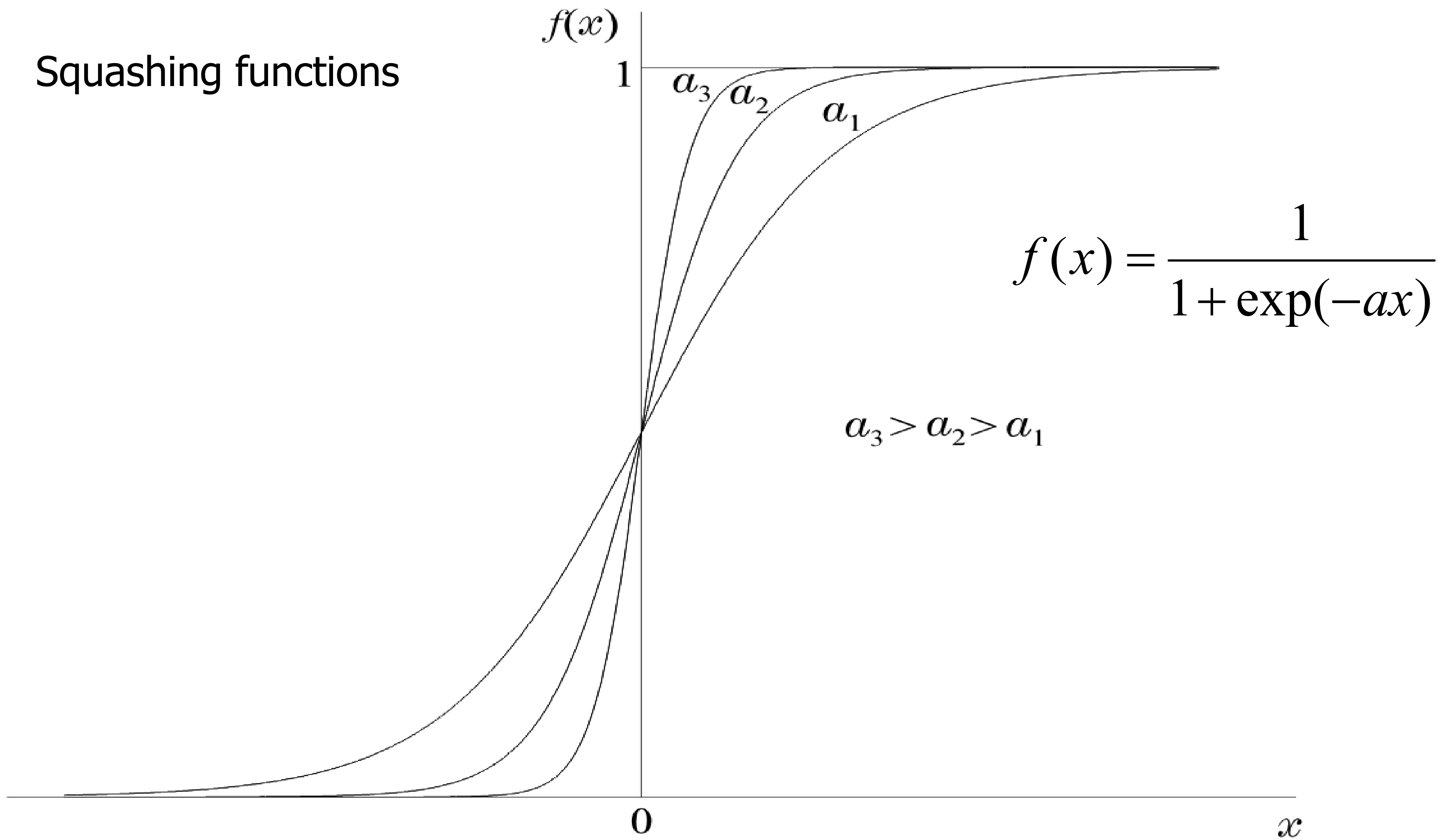
$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

- 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.

Squashing functions



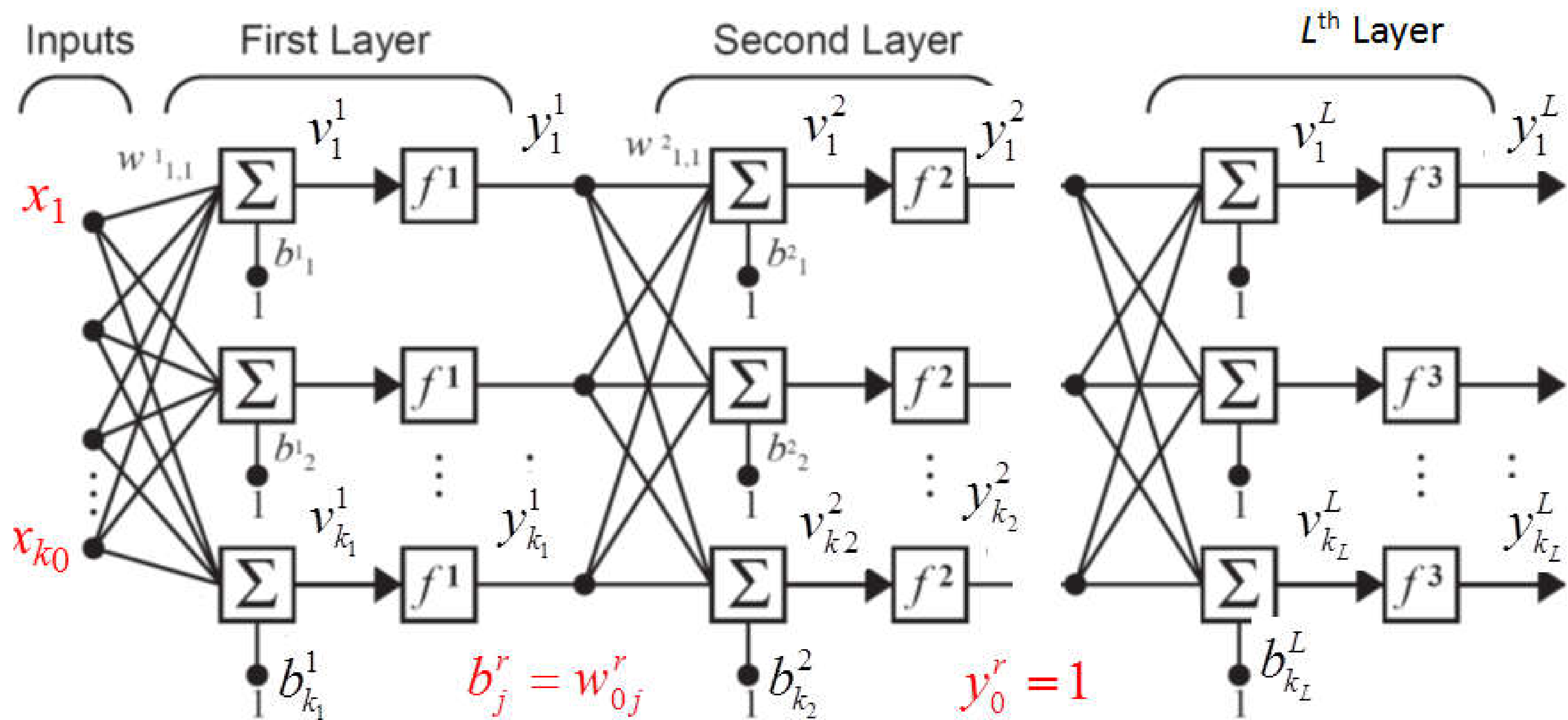
$$f(x) = \frac{1}{1+\exp(-ax)}$$

$$f(x) = \frac{2}{1+\exp(-ax)} - 1 \quad , \quad f(x) = c\,\frac{1-\exp(-ax)}{1+\exp(-ax)} = c.\tanh\left(\frac{ax}{2}\right)$$

- 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



Inputs
First Layer
Second Layer
L^{th} Layer
x_1
x_{k_0}
$w^1_{1,1}$
$w^2_{1,1}$
v^1_1
y^1_1
v^2_1
y^2_1
v^L_1
y^L_1
b^1_1
b^1_2
b^2_1
b^2_2
$v^1_{k_1}$
$y^1_{k_1}$
v^2_{k2}
$y^2_{k_2}$
$v^L_{k_L}$
$y^L_{k_L}$
f^1
f^2
f^3
$b^1_{k_1}$
$b^2_{k_2}$
$b^L_{k_L}$
$b^r_j = w^r_{0j}$
$y^r_0 = 1$

- L = # of layers , k_r = # of nodes in the rth layer, $k_0 = l$
- The input (feature) vectors $\underline{x}(i)=[x_1(i), . . . , x_{k_0}(i)]^T$
- The output vectors $\underline{y}(i)=[y_1(i), . . . , y_{k_L}(i)]^T$

- The weight vector (including the threshold) of the jth neuron in the rth layer, which is a vector of dimension $k_{r-1} + 1$ is defined as:

$$\underline{w}_j^r = \left[w_{j0}^r, w_{j1}^r, \ldots, w_{jk_{r-1}}^r \right]^T$$

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

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

$$\Delta \underline{w}_j^r = -\mu \frac{\partial J}{\partial \underline{w}_j^r}$$

Where J is a Cost function

- The Procedure:
  - Initialize unknown weights randomly with small values.
  - For each of the training feature vectors compute outputs. Compute the cost function for the current estimate of weights.
  - 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

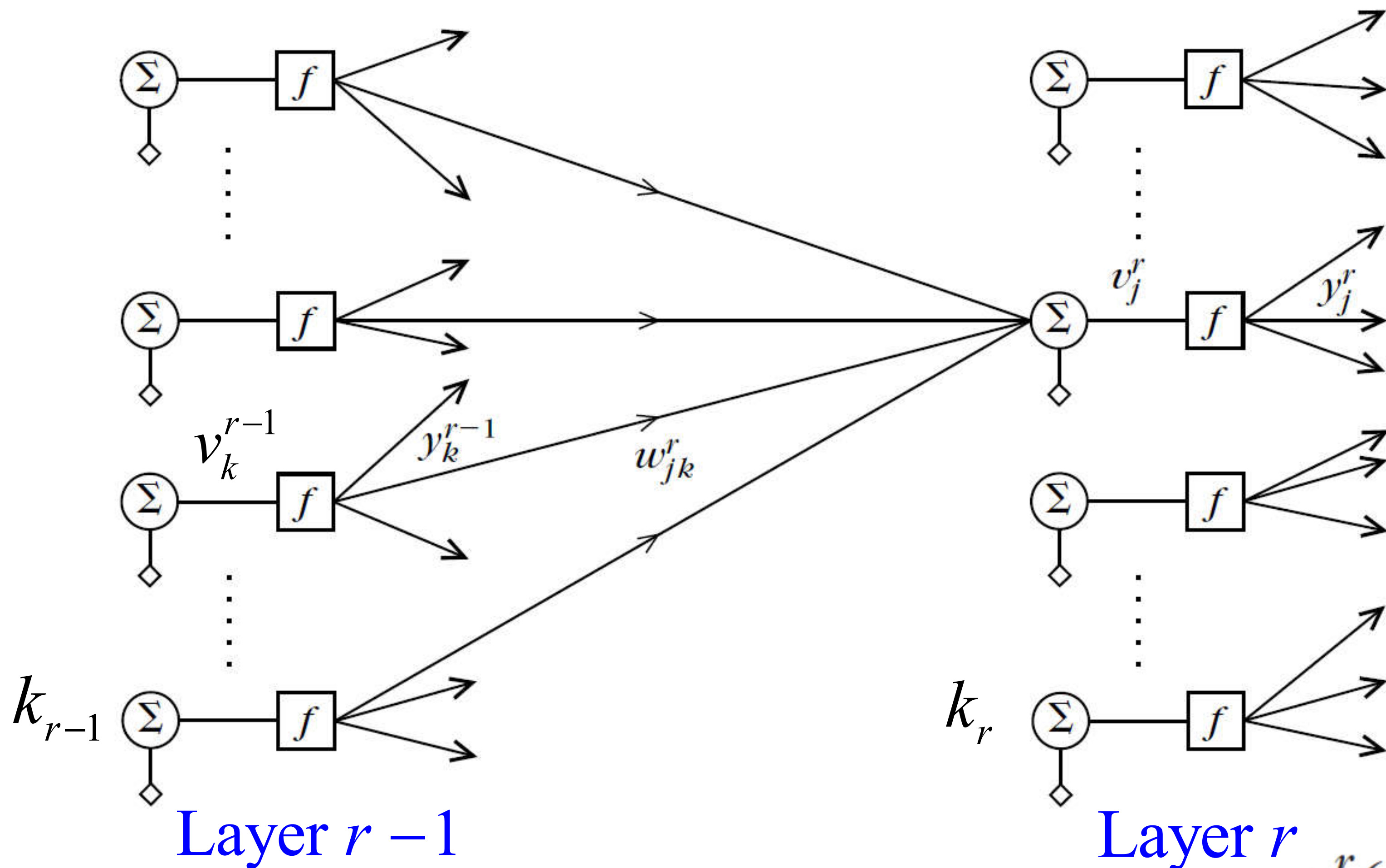
$$J = \sum_{i=1}^{N} \mathcal{E}(i)$$

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

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

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

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



$y_0^r(i) \equiv +1, \forall r$

$$v_j^r(i) = \sum_{k=1}^{k_{r-1}} w_{jk}^r \, y_k^{r-1}(i) + w_{jo}^r \equiv \sum_{k=0}^{k_{r-1}} w_{jk}^r \, y_k^{r-1}(i)$$

$$\frac{\partial \mathcal{E}(i)}{\partial \boldsymbol{w}_j^r} = \frac{\partial \mathcal{E}(i)}{\partial v_j^r(i)} \frac{\partial v_j^r(i)}{\partial \boldsymbol{w}_j^r}$$

we define $\dfrac{\partial \mathcal{E}(i)}{\partial v_j^r(i)} \equiv \delta_j^r(i)$

$$\frac{\partial}{\partial \boldsymbol{w}_j^r} v_j^r(i) \equiv \begin{bmatrix} \frac{\partial}{\partial w_{j0}^r} v_j^r(i) \\ \vdots \\ \frac{\partial}{\partial w_{jk_{r-1}}^r} v_j^r(i) \end{bmatrix} = \boldsymbol{y}^{r-1}(i) \quad \text{where} \quad \boldsymbol{y}^{r-1}(i) = \begin{bmatrix} +1 \\ y_1^{r-1}(i) \\ \vdots \\ y_{k_{r-1}}^{r-1}(i) \end{bmatrix}$$

$$\Rightarrow \Delta \boldsymbol{w}_j^r = -\mu \sum_{i=1}^{N} \delta_j^r(i) \boldsymbol{y}^{r-1}(i)$$

Computation of $\delta_j^r(i)$ for the Cost Function in (4.6)

The computations start from $r=L$ and propagate backward for $r=L-1, L-2, \ldots, 1$.

1. $r = L$ $\quad \delta_j^L(i) = \frac{\partial \mathcal{E}(i)}{\partial v_j^L(i)}$; $\quad \longrightarrow \quad \delta_j^L(i) = e_j(i) f'(v_j^L(i))$

$$\mathcal{E}(i) \equiv \frac{1}{2} \sum_{m=1}^{k_L} e_m^2(i) \equiv \frac{1}{2} \sum_{m=1}^{k_L} (f(v_m^L(i)) - y_m(i))^2$$

2. $r < L$

$$\frac{\partial \mathcal{E}(i)}{\partial v_j^{r-1}(i)} = \sum_{k=1}^{k_r} \frac{\partial \mathcal{E}(i)}{\partial v_k^r(i)} \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)}$$

$$\delta_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)}$$

But
$$\frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = \frac{\partial \left[\sum_{m=0}^{k_{r-1}} w_{km}^r \, y_m^{r-1}(i) \right]}{\partial v_j^{r-1}(i)}$$

with
$$y_m^{r-1}(i) = f(v_m^{r-1}(i))$$

Hence,
$$\frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = w_{kj}^r \, f'(v_j^{r-1}(i))$$

$$\Rightarrow \quad \delta_j^{r-1}(i) = \left[\sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r\right] f'(v_j^{r-1}(i))$$

$$\delta_j^{r-1}(i) = e_j^{r-1}(i) f'\left(v_j^{r-1}(i)\right)$$

$$e_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r$$

where $f'(x) = af(x)(1 - f(x))$

$$\delta_j^L(i) = e_j(i) f'\left(v_j^L(i)\right)$$

The Backpropagation Algorithm

1- Initialization

2- Forward computations:

3- Backward computations:

4- Update the weights

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

$$\Delta \underline{w}_j^r = -\mu \sum_{i=1}^{N} \delta_j^r(i) \underline{y}^{r-1}(i)$$

Demo: nnd11bc nnd11fa nnd11gn

VARIATIONS ON THE BACKPROPAGATION THEME

❖ Use Momentum term

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

$$\Delta\underline{w}_j^r(new) = \alpha\Delta\underline{w}_j^r(old) - \mu\sum_{i=1}^{N}\delta_j^r(i)\underline{y}^{r-1}(i)$$

the *momentum factor* $0.1<\alpha<0.8$

Adaptive learning factor μ

$$\frac{J(t)}{J(t-1)} < 1, \quad \mu(t) = r_i\mu(t-1)$$

$$\frac{J(t)}{J(t-1)} > c, \quad \mu(t) = r_d\mu(t-1)$$

$$1 \leq \frac{J(t)}{J(t-1)} \leq c, \quad \mu(t) = \mu(t-1)$$

$\mu=0.01$, $\alpha=0.85$,
$r_i=1.05$, $c=1.05$, $r_d=0.7$.

- Use an adaptive value for the learning factor
- *delta-delta* rule
- Conjugate gradient algorithm
- Newton family approaches
- Algorithms based on the Kalman filtering approach
- Levenberg–Marquardt algorithm
- *Quickprop & Rprop* schemes

The cross-entropy

- At the least squares cost function all errors in the output nodes are first squared and summed up, large error values influence the learning process much more than the small errors.
- If the dynamic ranges of the desired outputs are not *all* of the same order, the least squares criterion will result in weights that have "learned" via a process of unfair provision of information.
- In Ch 3 we have seen that, if we adopt the least squares cost function and the desired outputs y_k are binary (belong to or not in class ω_k), then for the optimal values of the weights ***w**** the corresponding output of the network, $\hat{y}_k$, is *the least squares optimal estimate of the posterior probability* $P(\omega_k | x)$

- Assume the desired output values, y_k, are independent binary random variables and that $\hat{y}_k$ are the respective posterior probabilities that these random variables are 1.

$$p(\mathbf{y}) = \prod_{k=1}^{k_l} (\hat{y}_k)^{y_k} (1-\hat{y}_k)^{1-y_k}$$

- The cross-entropy cost function is then defined by $J = -\sum_{i=1}^{N} \log(p(\mathbf{y}))$

$$J = -\sum_{i=1}^{N} \sum_{k=1}^{k_L} \{ y_k(i) \ln \hat{y}_k(i) + (1 - y_k(i)) \ln(1 - \hat{y}_k(i)) \}$$

- J takes its minimum value when $y_k(i) = \hat{y}_k(i)$.
- If $y_k(i)$ were true probabilities in (0, 1) then subtracting the minimum value from J becomes

$$J = -\sum_{i=1}^{N} \sum_{k=1}^{k_L} \left(y_k(i) \ln \frac{\hat{y}_k(i)}{y_k(i)} + (1 - y_k(i)) \ln \frac{1 - \hat{y}_k(i)}{1 - y_k(i)} \right)$$

- For binary valued y_ks the above is still valid if we use the limiting value $0 \ln 0 = 0$.

- The cross-entropy cost function depends on the relative errors and not on the absolute errors, as its least squares counterpart; thus it gives the same weight to small and large values.
- It can be shown that adopting the cross-entropy cost function and binary values for the desired responses, the outputs $\hat{y}_k$ *corresponding to the* optimal weights ***w*** are indeed estimates of* $P(\omega_k|x)$, *as in the least squares case.* This presupposes an interpretation of y *and* $\hat{y}$ *as* **probabilities**.
- An alternative cost function is the relative entropy or KL divergence, (rarely used)

$$J = -\sum_{i=1}^{N}\sum_{k=1}^{k_L} y_k(i)\ln\frac{\hat{y}_k(i)}{y_k(i)}$$

- 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 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.

❖ Choice of the network size.

How big a network can be. How many layers and how many neurons per layer?

❖ The number of free parameters (synaptic weights) to be estimated should be

- ➢ (a) large enough to learn what makes "similar" the feature vectors within each class and at the same time what makes one class different from the other.
- ➢ (b) small enough, with respect to number N of training pairs, so as not to be able to learn the underlying differences among the data of the same class.

There are 3 major directions:

- 1) Analytical methods. This category employs algebraic or statistical techniques to determine the number of its free parameters. It is static and does not take into consideration the cost function used as well as the training procedure

- 2) 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: Taylor series Exp.

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

$$+ \quad \text{higher order terms}$$

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

Near a minimum and assuming that the Hessian matrix is diagonal

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

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 first term is the performance cost function, and it is chosen according to what we have already discussed (e.g., least squares, cross entropy).

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

$$E_p(\underline{w}) = \sum_{k=1}^{K} h(w_k^2)$$

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

$h(\cdot)$ is an appropriately chosen differentiable function.

where $w_0 \cong 1$.
After some training steps, weights with small values are removed.

- 3) Constructive techniques:
  They start with a small network and keep increasing it, according to a predetermined procedure and criterion.

- Remark: Why do 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. Too many parameters leads to the so called overfitting.

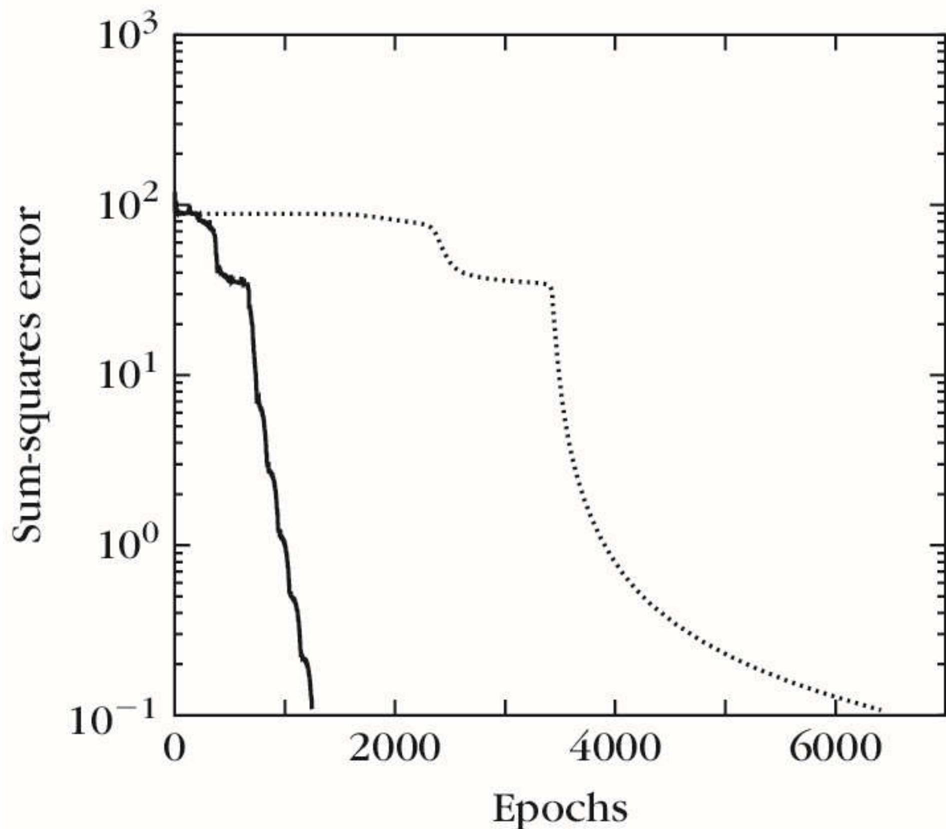
Example: *NN*: (2-3-2-1) ; Logistic Function with a=1.

(a) The momentum μ=0.05, α=0.85 and

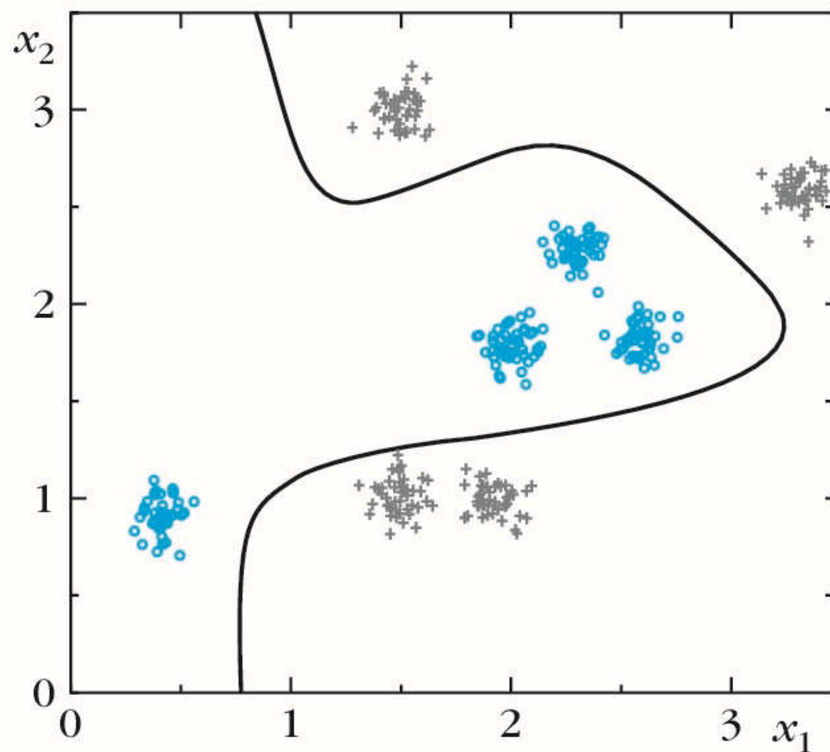
(b) The adaptive momentum μ=0.01, α=0.85, r_i=1.05, c=1.05, r_d =0.7.

$[0.4, 0.9]^T$, $[2, 1.8]^T$, $[2.3, 2.3]^T$, $[2.6, 1.8]^T$
$[1.5, 1.0]^T$, $[1.9, 1.0]^T$, $[1.5, 3.0]^T$, $[3.3, 2.6]^T$

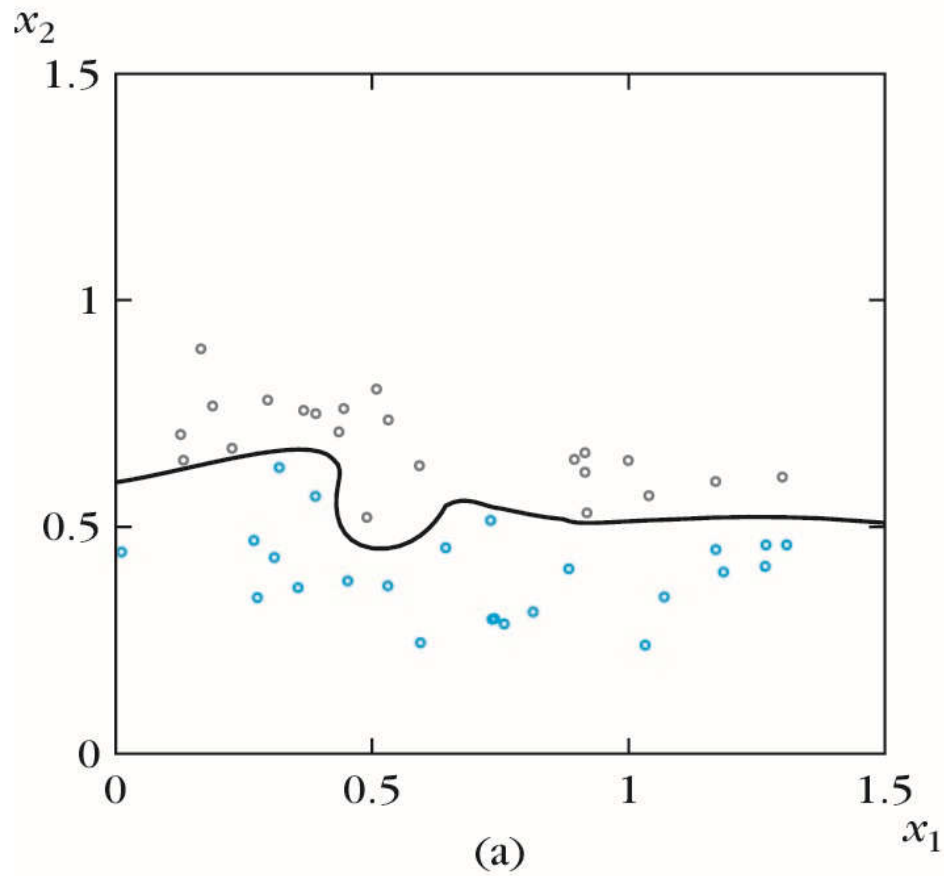
400 training Samples
The mean values
Variances=0.08

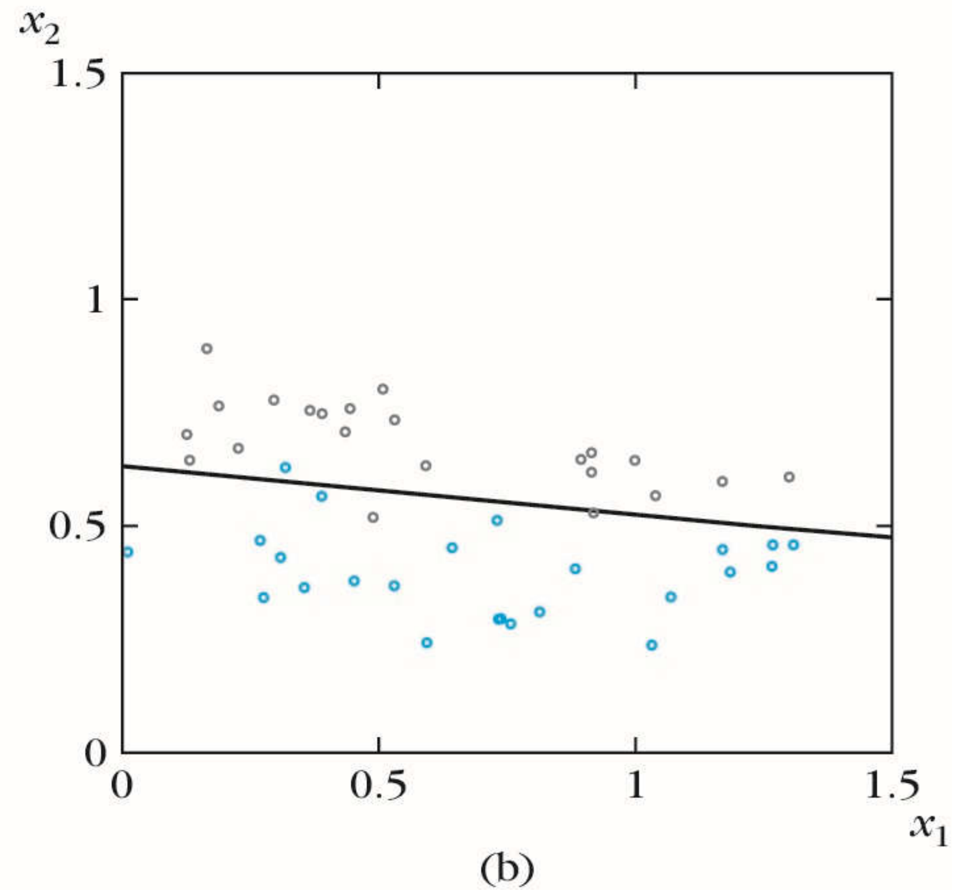


(a)



(b)

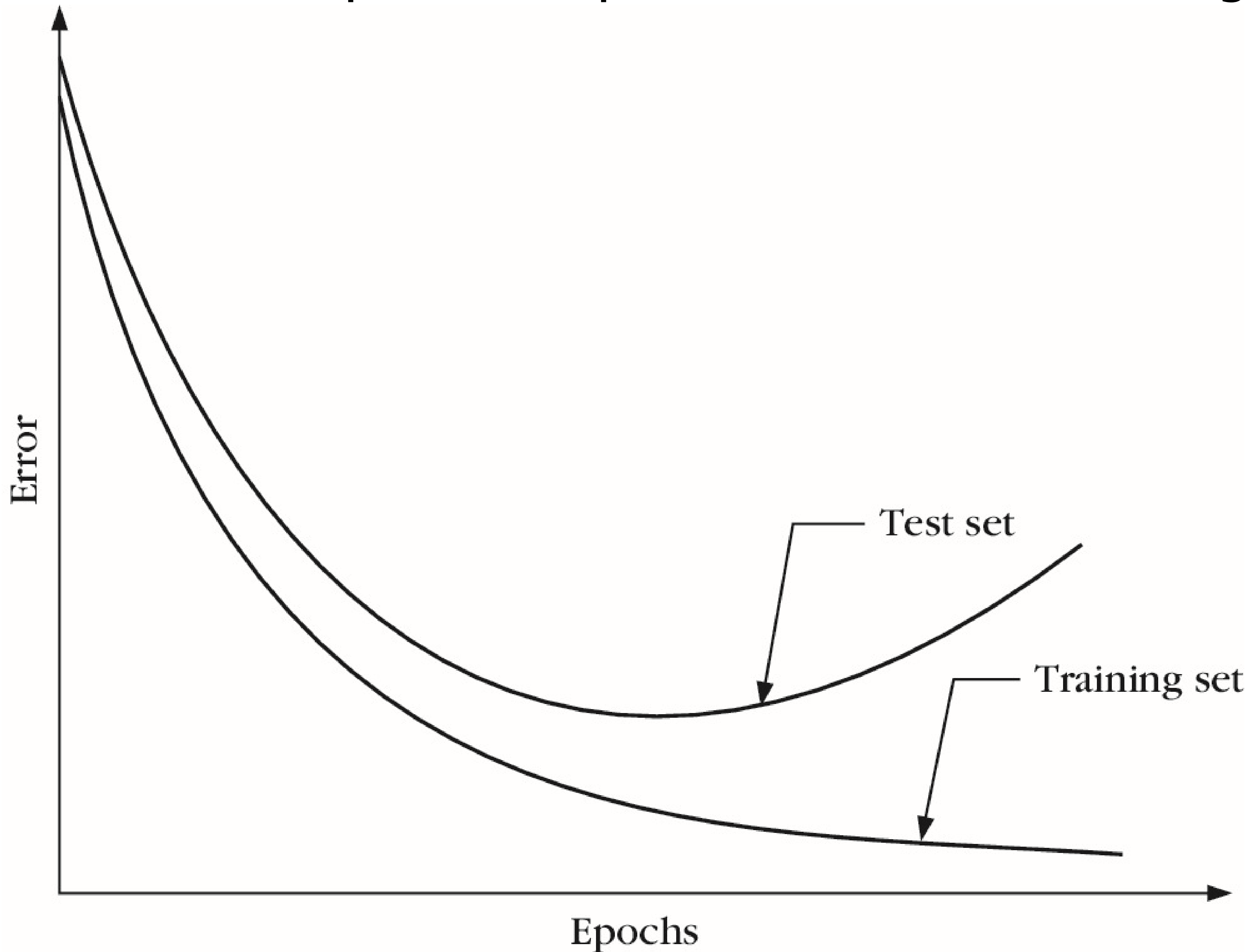




MLP: (2-20-20-1)

Decision curve (a) before pruning and (b) after pruning.

- 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

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

$f(.) \rightarrow$ 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.

$$f_i(.), i = 1,2,..., k$$

$$f_i : R^l \rightarrow R, \qquad i = 1, 2, \ldots, k$$

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

$$\underline{x} \rightarrow \underline{y} = \begin{bmatrix} f_1(\underline{x}) \\ \dots \\ f_k(\underline{x}) \end{bmatrix} \qquad \boldsymbol{x} \in \mathcal{R}^l \rightarrow \boldsymbol{y} \in \mathcal{R}^k$$

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

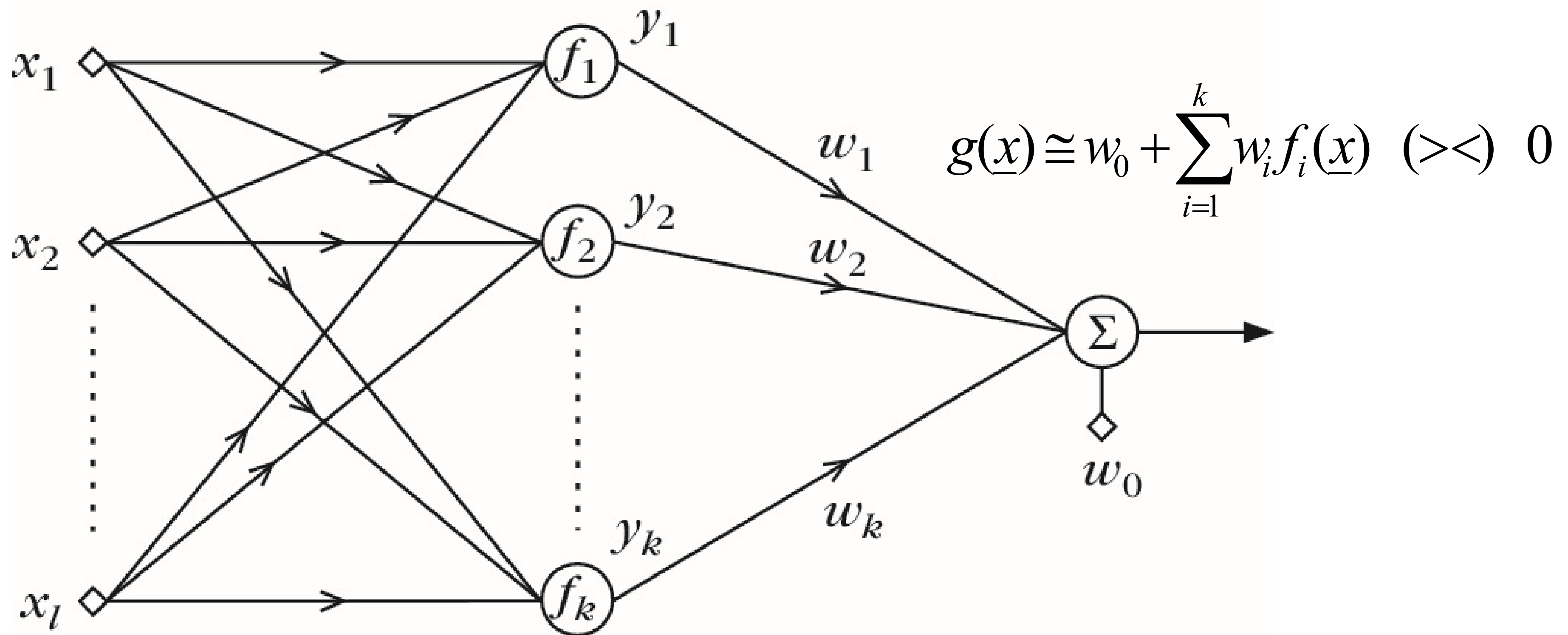
$$\text{If } \; w_0 + \underline{w}^T \underline{y} > 0, \quad \underline{x} \in \omega_1$$
$$w_0 + \underline{w}^T \underline{y} < 0, \quad \underline{x} \in \omega_2$$

- 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.



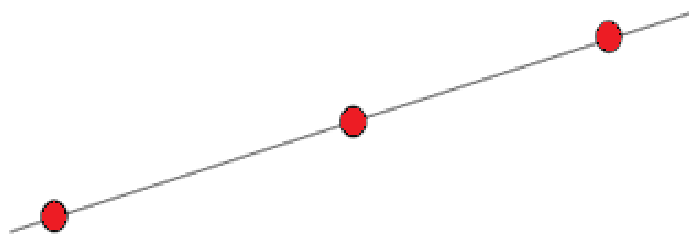
Generalized Linear Classification.

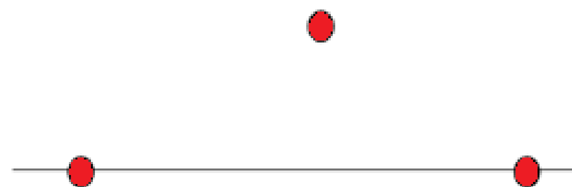
$g(\boldsymbol{x})$ corresponds to a two-layer network where the nodes of the hidden layer have different activation functions, $f_i(\cdot)$, i=1, 2, ..., k.

❖ Capacity of the l-dimensional space in Linear Dichotomies

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

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





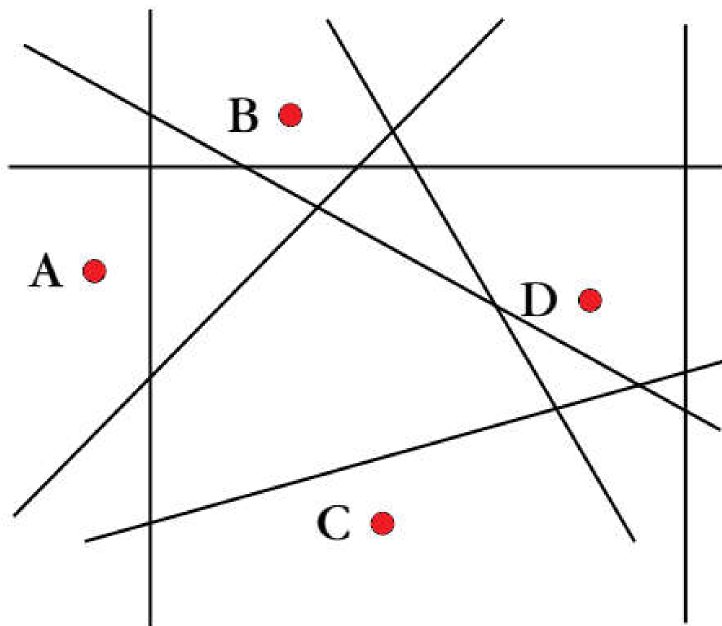
Not in general position

general position

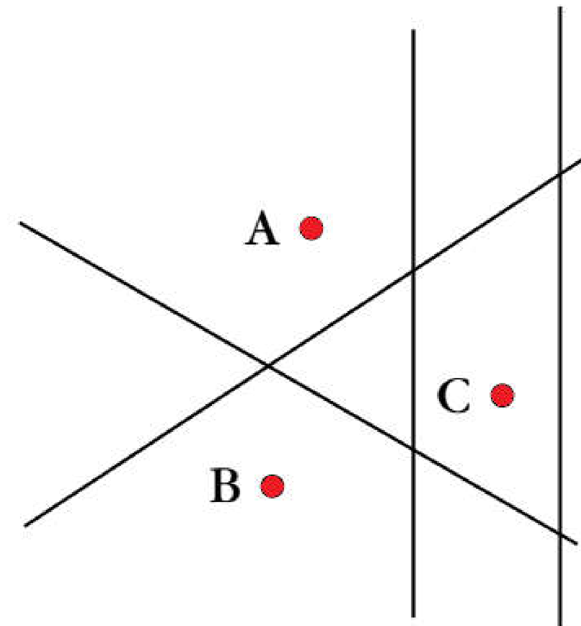
- *Cover's theorem states: The number of groupings that can be formed by (l-1)-dimensional hyperplanes to separate N points in two classes is

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

Example: N=4, l=2, $O(4,2)$=**14,** and $O(3, 2)$ = **8**



(a)

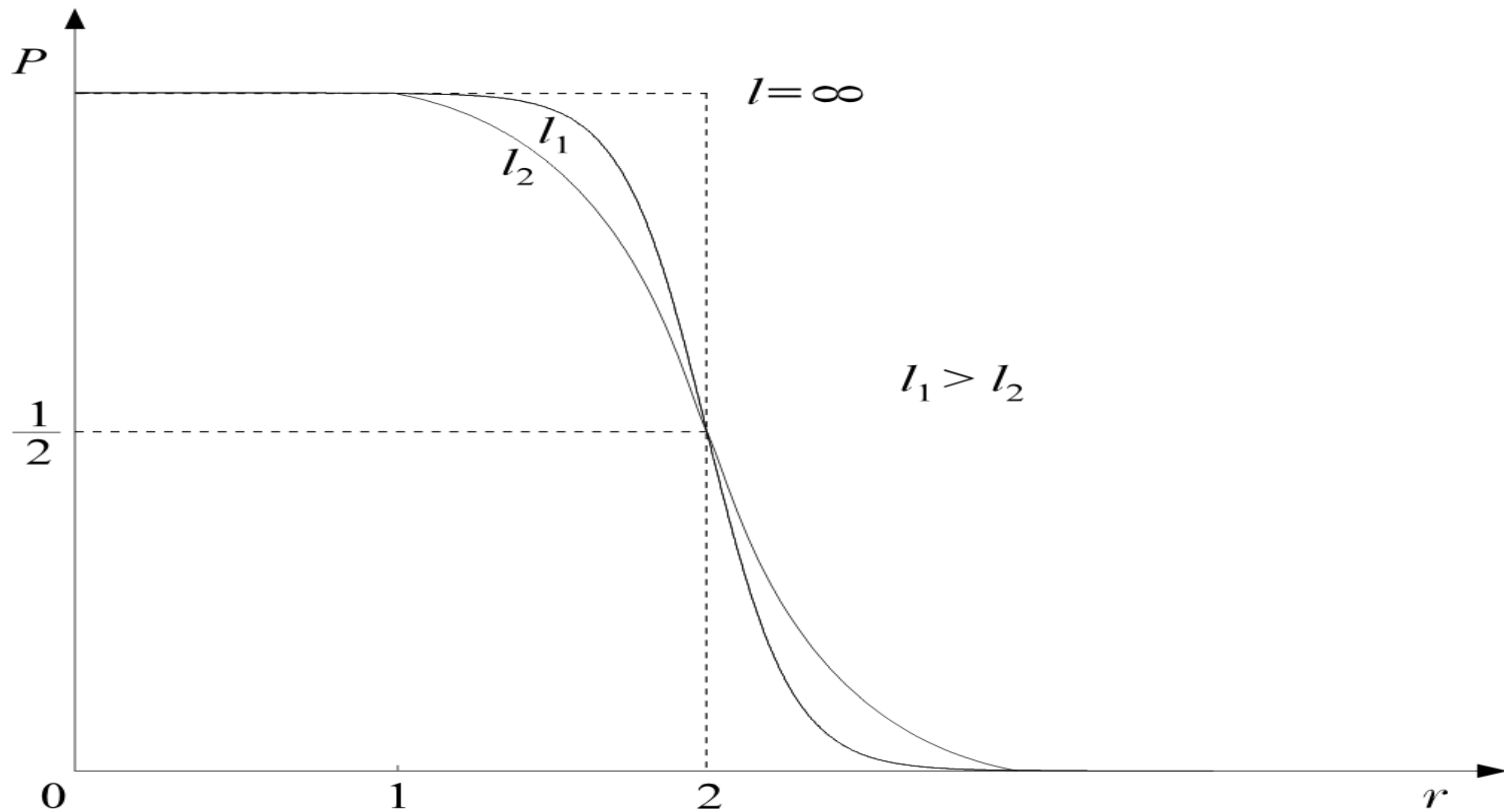


(b)

Notice: The total number of possible groupings is 2^4=16 (AD), (BC)

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

$$P_N^l = \frac{O(N,l)}{2^N} = \begin{cases} \frac{1}{2^{N-1}} \sum_{i=0}^{l} \binom{N-1}{i} & N > l+1 \\ 1 & N \le l+1 \end{cases}$$



$$N = r(l+1)$$

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.

❖POLYNOMIAL CLASSIFIERS

Function g(**x**) is approximated in terms of up to order r polynomials of the **x** components, for large enough r. For the special case of r =2 we have:

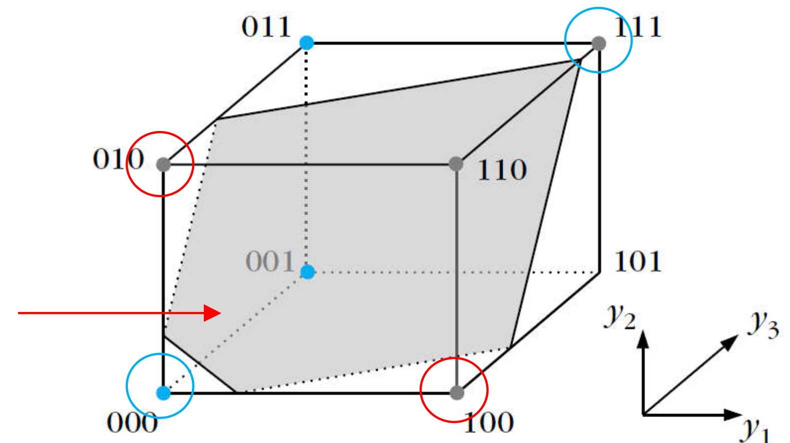
$$g(\boldsymbol{x}) = w_0 + \sum_{i=1}^{l} w_i x_i + \sum_{i=1}^{l-1} \sum_{m=i+1}^{l} w_{im} x_i x_m + \sum_{i=1}^{l} w_{ii} x_i^2$$

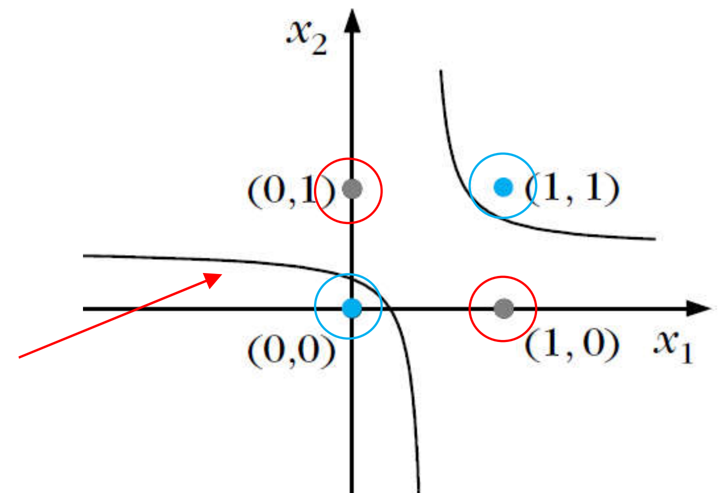
$$\boldsymbol{x} = [x_1, x_2]^T \qquad \boldsymbol{y} = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]^T$$

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{y} + w_0 \qquad \boldsymbol{w}^T = [w_1, w_2, w_{12}, w_{11}, w_{22}]$$

XOR problem:

$$\boldsymbol{y} = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix} \quad y_1 + y_2 - 2y_3 - \frac{1}{4} = 0$$

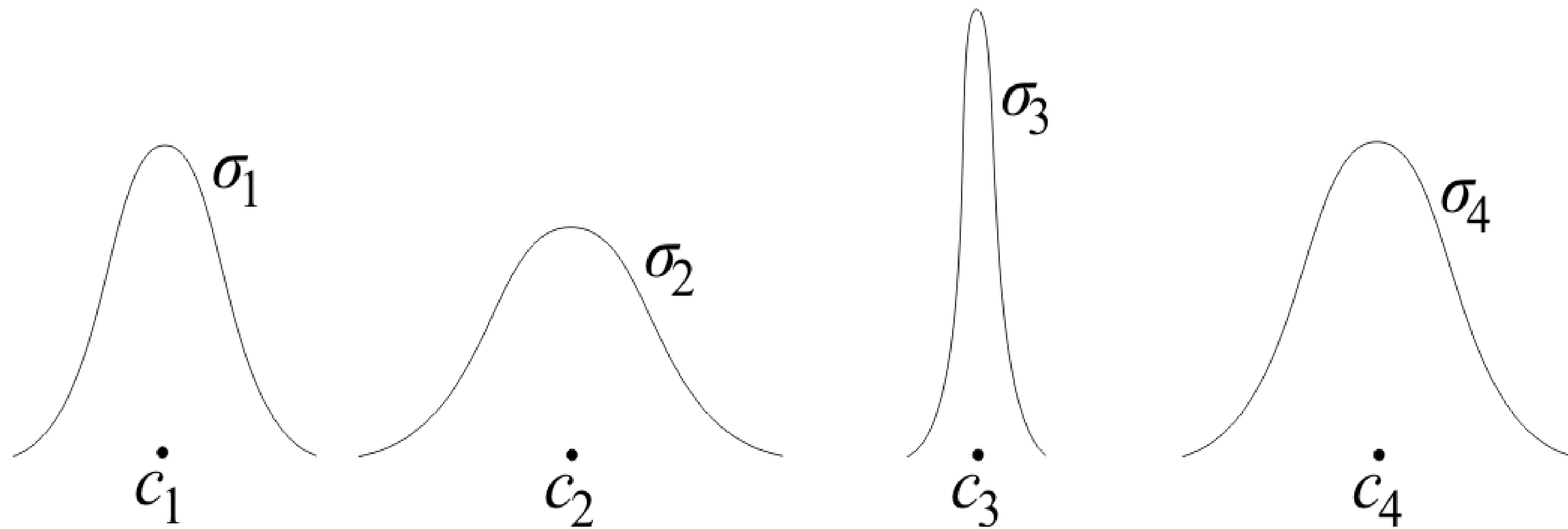




$$g(\boldsymbol{x}) = -\frac{1}{4} + x_1 + x_2 - 2x_1 x_2 \begin{array}{ll} >0 & \boldsymbol{x} \in A \\ <0 & \boldsymbol{x} \in B \end{array}$$

❖Radial Basis Function Networks (RBF)

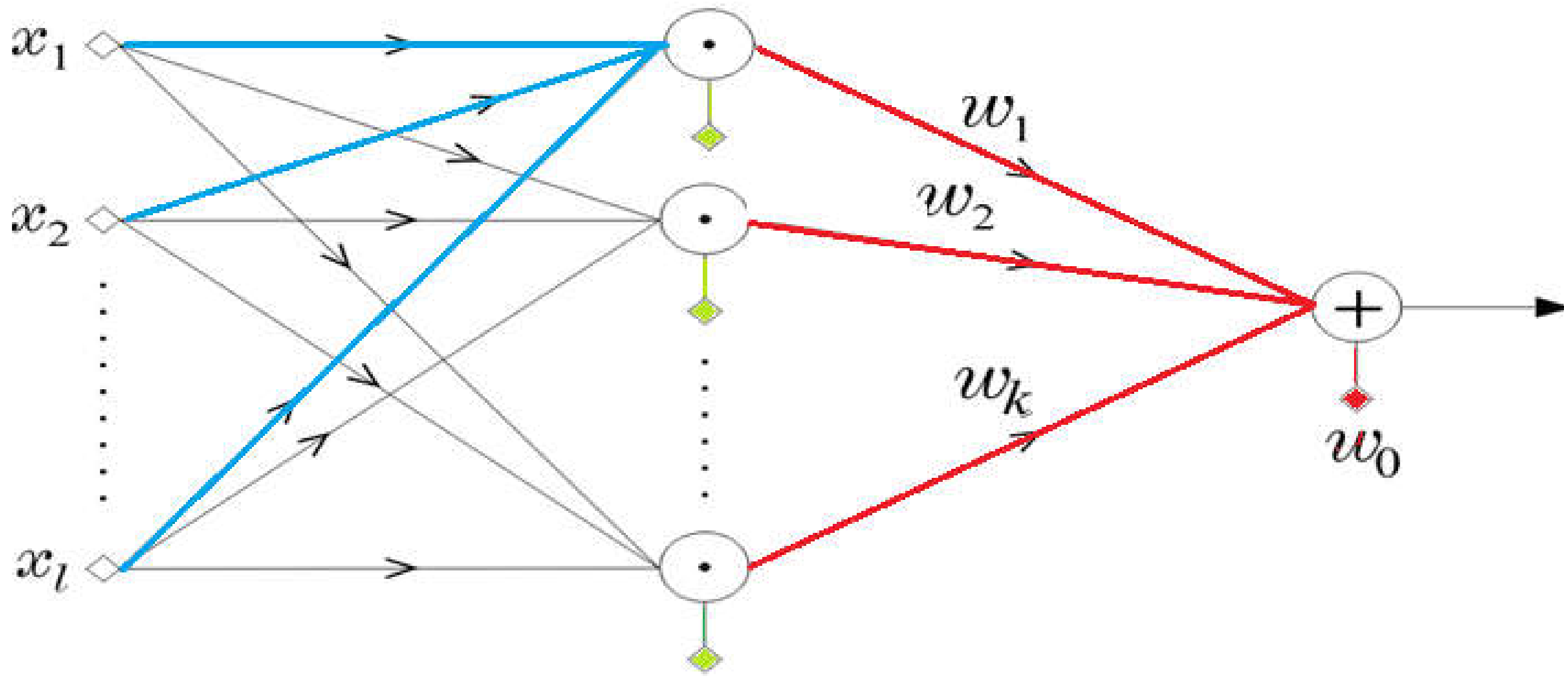
- Choose $f(\|\boldsymbol{x} - \boldsymbol{c}_i\|)$



$$f(\boldsymbol{x}) = \exp\left(-\frac{1}{2\sigma_i^2}\|\boldsymbol{x} - \boldsymbol{c}_i\|^2\right)$$

$$f(\boldsymbol{x}) = \frac{\sigma^2}{\sigma^2 + \|\boldsymbol{x} - \boldsymbol{c}_i\|^2}$$

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



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

$$g(\boldsymbol{x}) = w_0 + \sum_{i=1}^{k} w_i \exp\left(- \frac{(\boldsymbol{x} - \boldsymbol{c}_i)^T (\boldsymbol{x} - \boldsymbol{c}_i)}{2\sigma_i^2} \right)$$

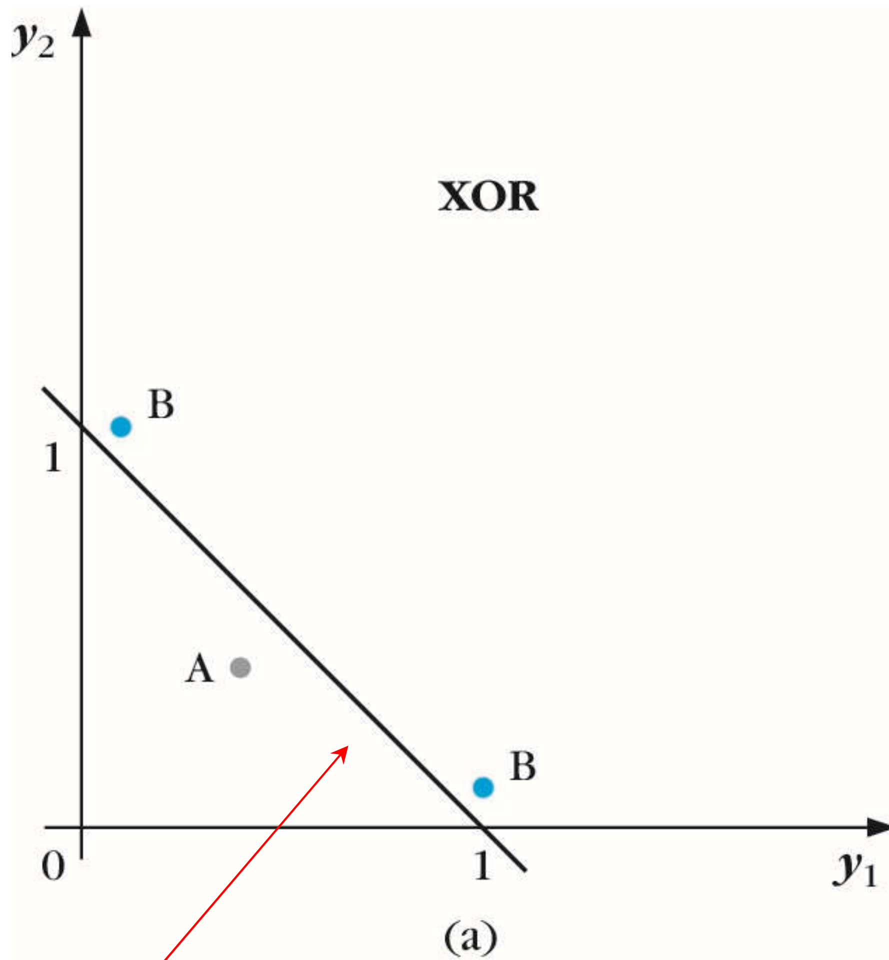
➢ Example: The XOR problem

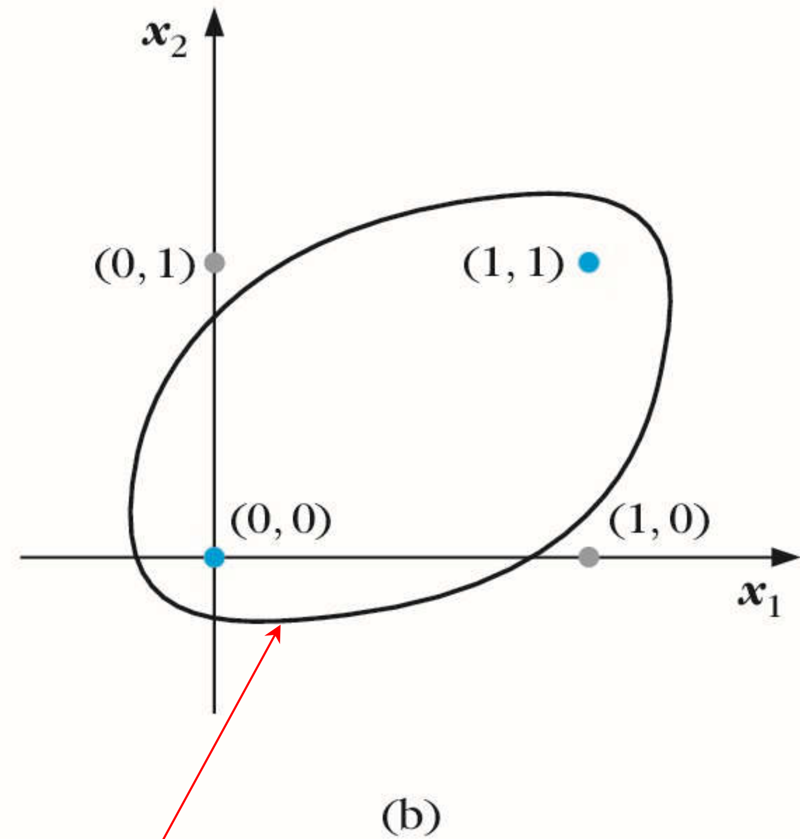
- Define:

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

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

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





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

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

❖ Training of the RBF networks

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

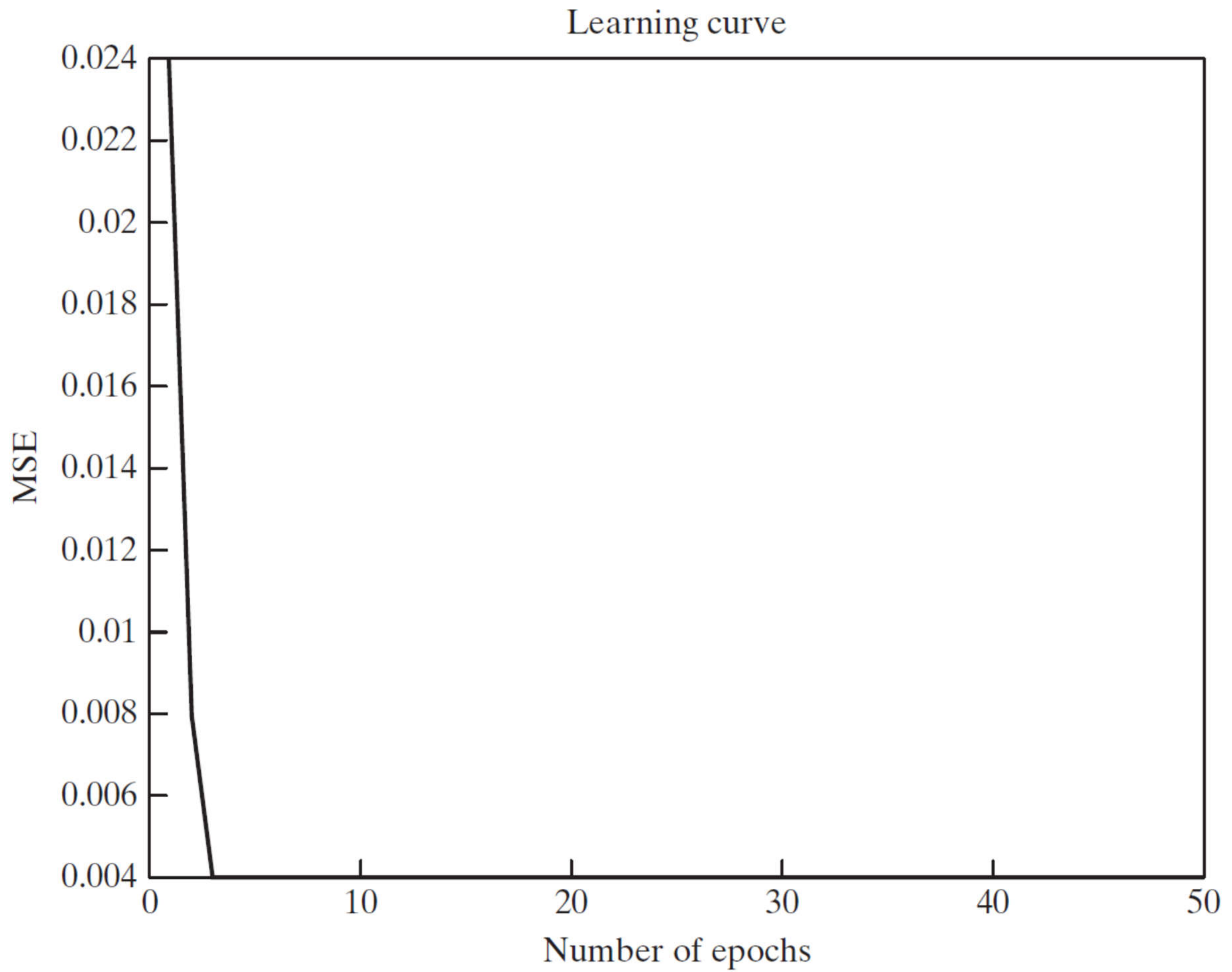
$$\underline{y} = \left[\exp\left(\frac{-\|\underline{x} - \underline{c}_1\|^2}{2\sigma_1^2} \right), \quad \cdots, \quad \exp\left(\frac{-\|\underline{x} - \underline{c}_k\|^2}{2\sigma_k^2} \right) \right]^T \qquad g(\underline{x}) = w_0 + \underline{w}^T \underline{y}$$

  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.



(a) Learning curve

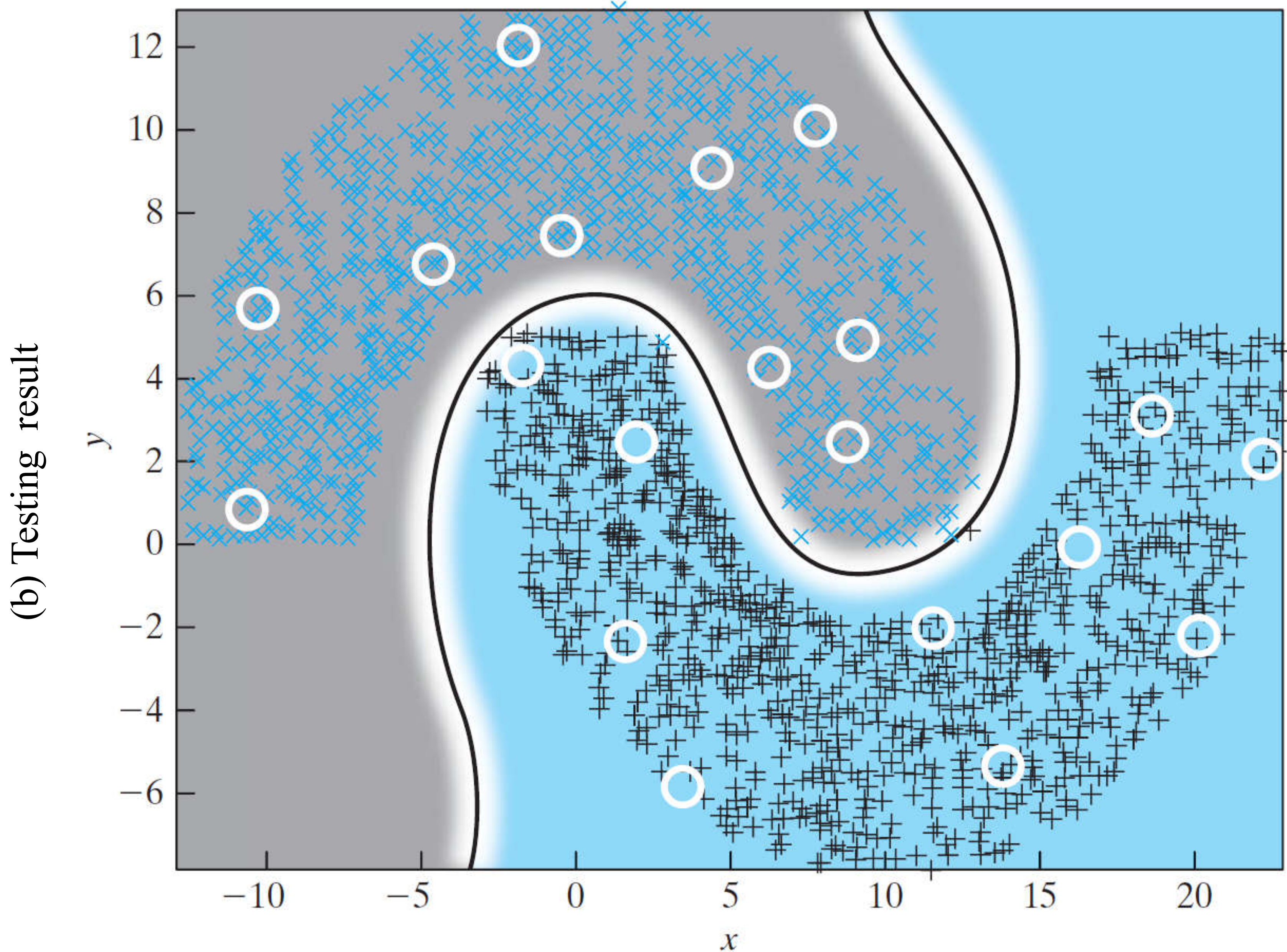
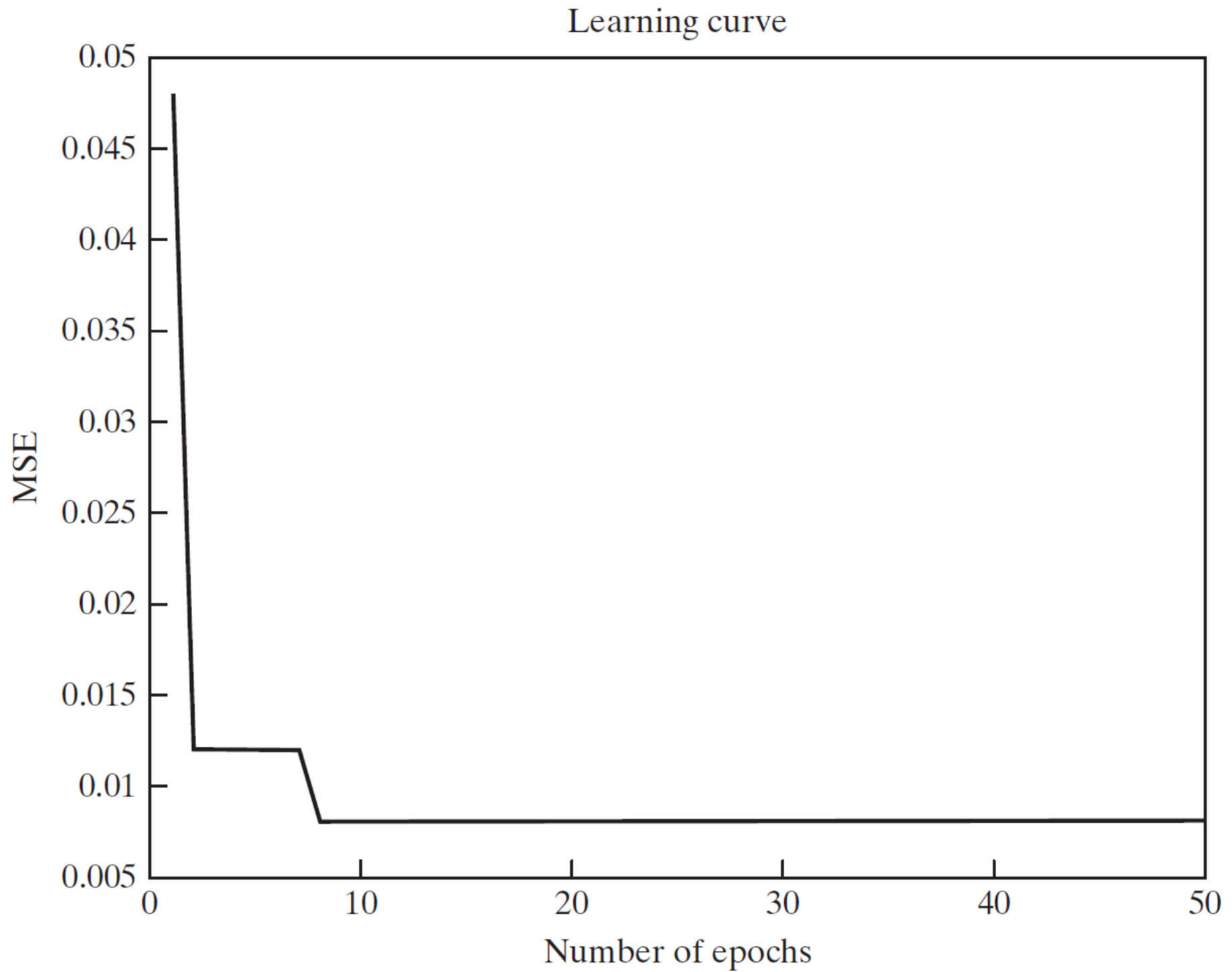


FIGURE 5.5 RBF network trained with *K*-means and RLS algorithms for distance d = -5. The MSE in part (a) of the figure stands for mean-square error.



(a) Learning curve

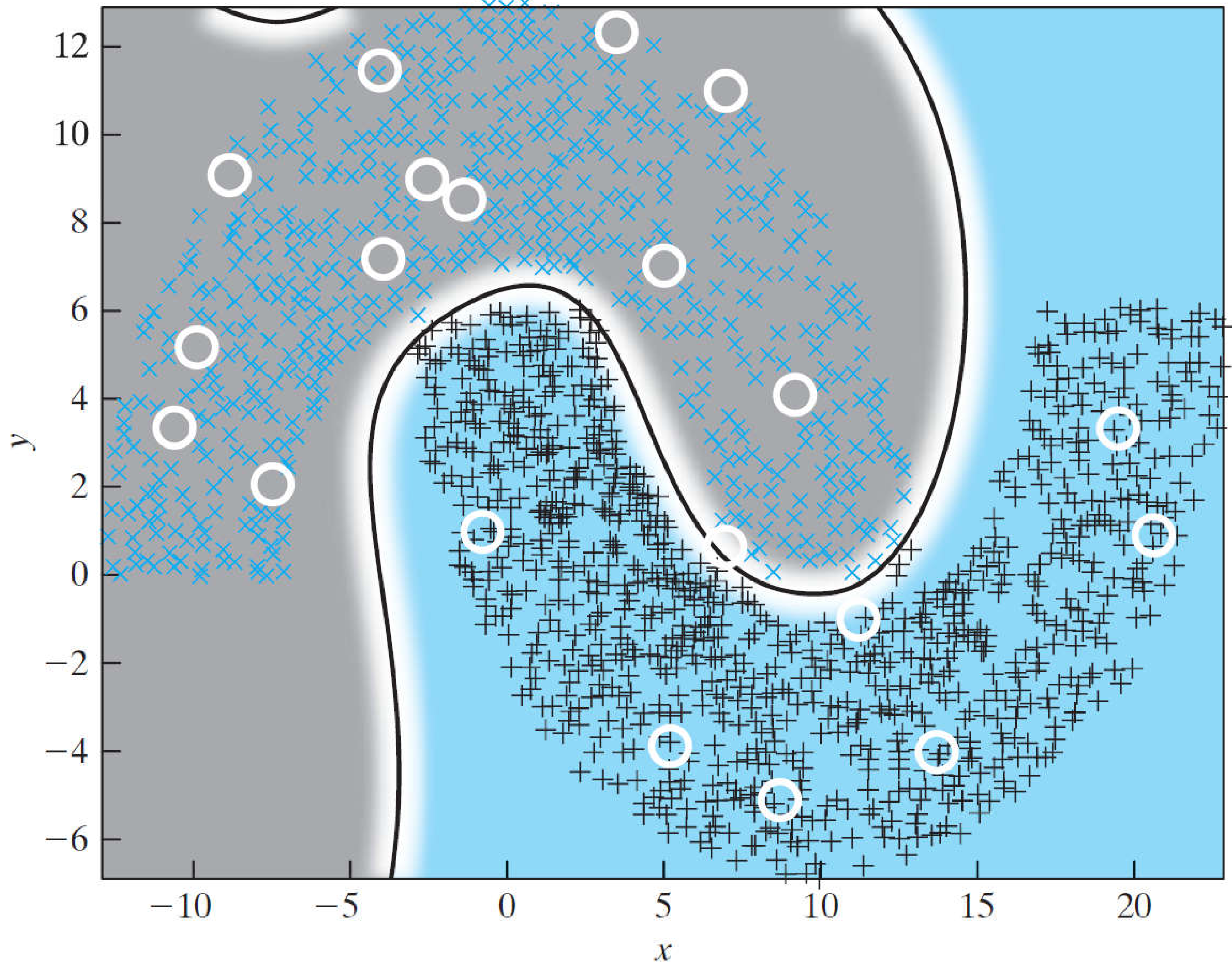


FIGURE 5.6 RBF network trained with *K*-means and RLS algorithms for distanced *d*=-6.

❖ 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

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

$$= \sum_{i=1}^{N_s} \lambda_i y_i \underline{y}_i \underline{y}$$

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

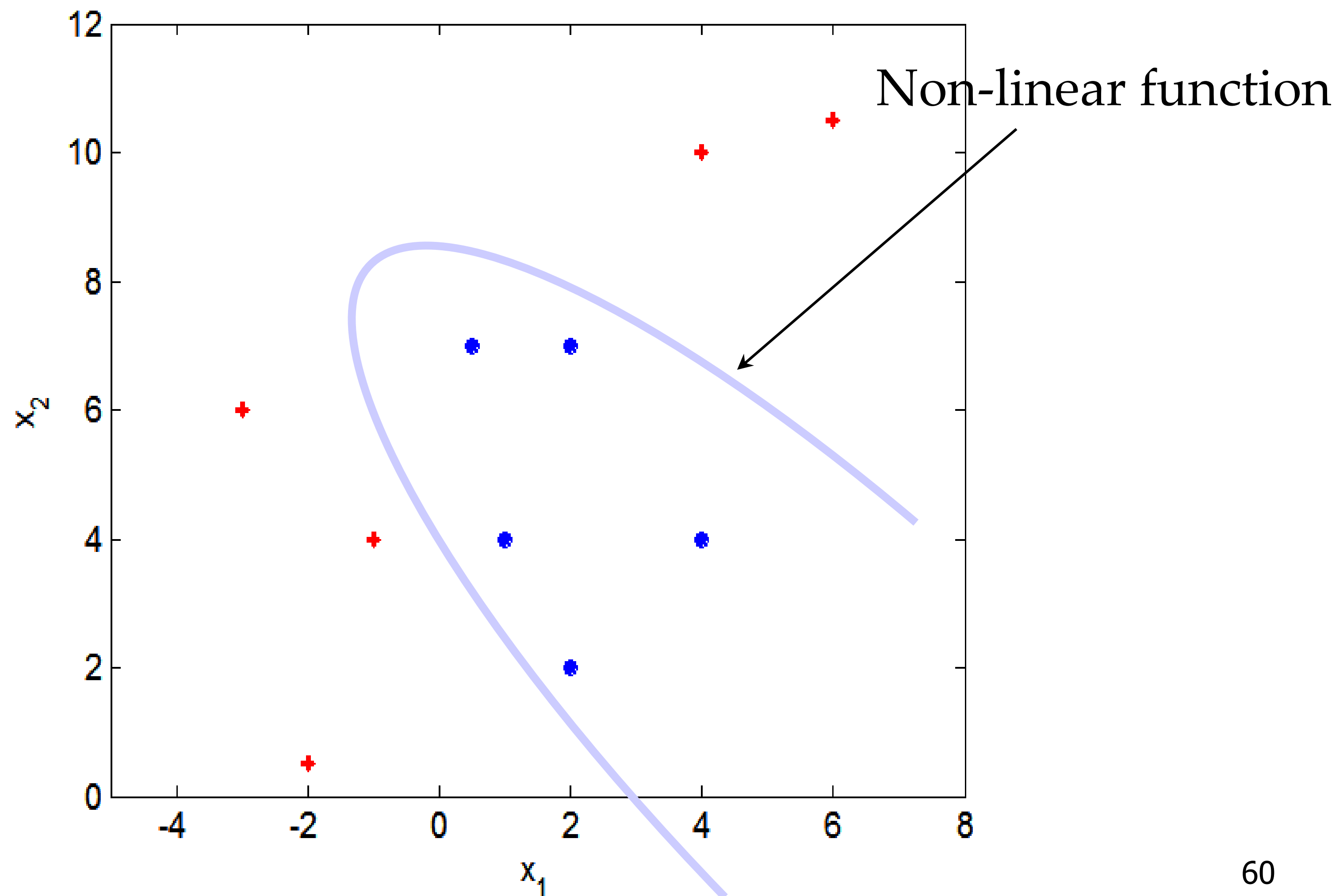
Thus, inner products in a high dimensional space are involved, hence

- High complexity

Nonlinear Support Vector Machines

- What if decision boundary is not linear?

Alternative 1:
Use technique that
Employs non-
linear decision
boundaries

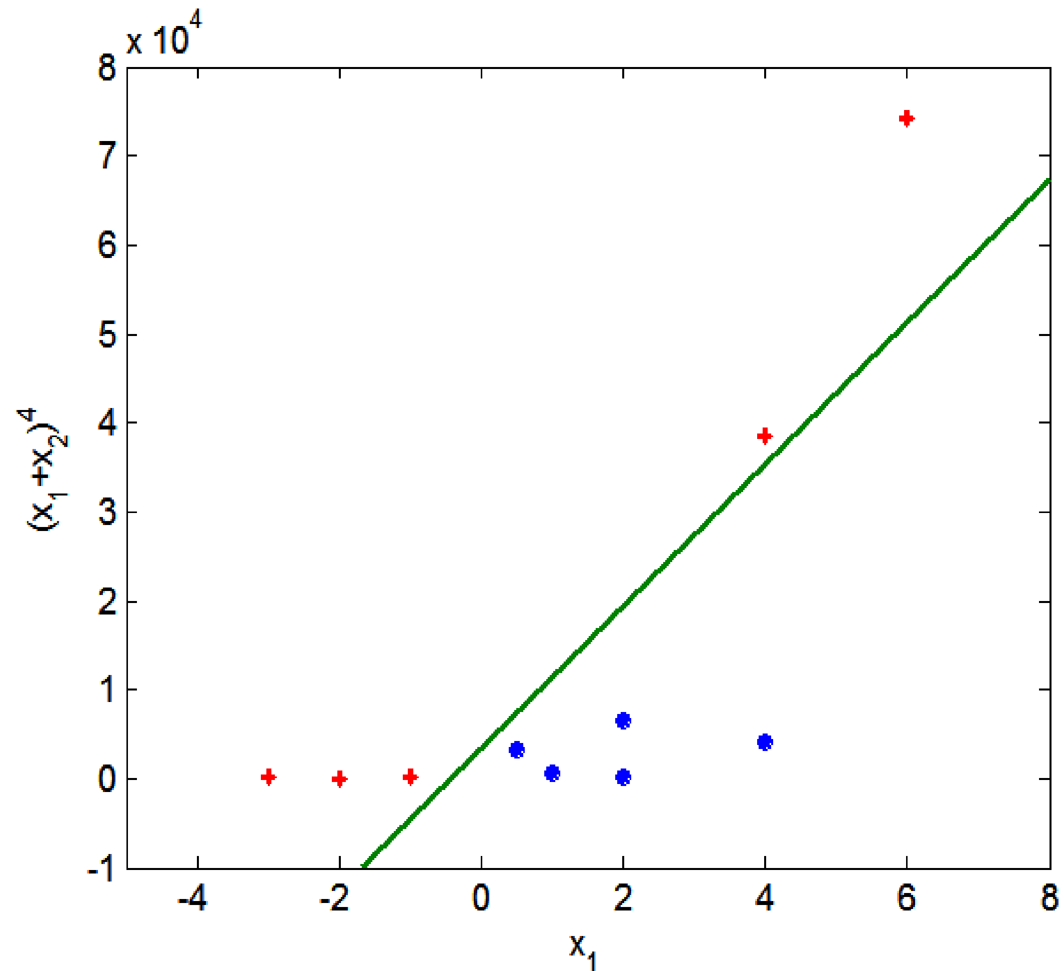


Nonlinear Support Vector Machines

1. Transform data into higher dimensional space
2. Find the best hyperplane using the methods introduced earlier

Alternative 2:

Transform into a higher dimensional attribute space and find linear decision boundaries in this space



- 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

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

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

  - 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 $\underline{x} \to \underline{\Phi}(\underline{x}) \in H$ H: Hilbert Space*

Then, the inner product in *H* is represented as:

$$\langle \Phi(\underline{x}), \quad \Phi(\underline{z}) \rangle = \sum_r \Phi_r(\underline{x}) \Phi_r(\underline{z}) = K(\underline{x}, \underline{z})$$

where $K(\underline{x},\underline{z})$ is a symmetric continuous function satisfying

$$\int_C \int_C K(\underline{x}, \underline{z}) g(\underline{x}) g(\underline{z}) d\underline{x} \, d\underline{z} \geq 0$$

for **any** $g(\underline{x}), \quad \underline{x} \in C \subset R^l$ such that:

$$\int_C g^2(\underline{x}) d\underline{x} < +\infty \qquad C\text{: Compact (finite) Subset of } R^l.$$

$K(\underline{x}, \underline{z})$ is a symmetric function known as kernel.

---

* A **Hilbert space** is a complete linear space equipped with an inner product operation. A finite dimensional Hilbert space is a Euclidean space.

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

- Examples of kernels
  - Polynomial:

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

  - Radial Basis Functions:

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

  - 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:

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

$$\text{subject to:} \quad 0 \le \lambda_i \le C, \quad i = 1, 2, \ldots, N$$

$$\sum_i \lambda_i y_i = 0$$

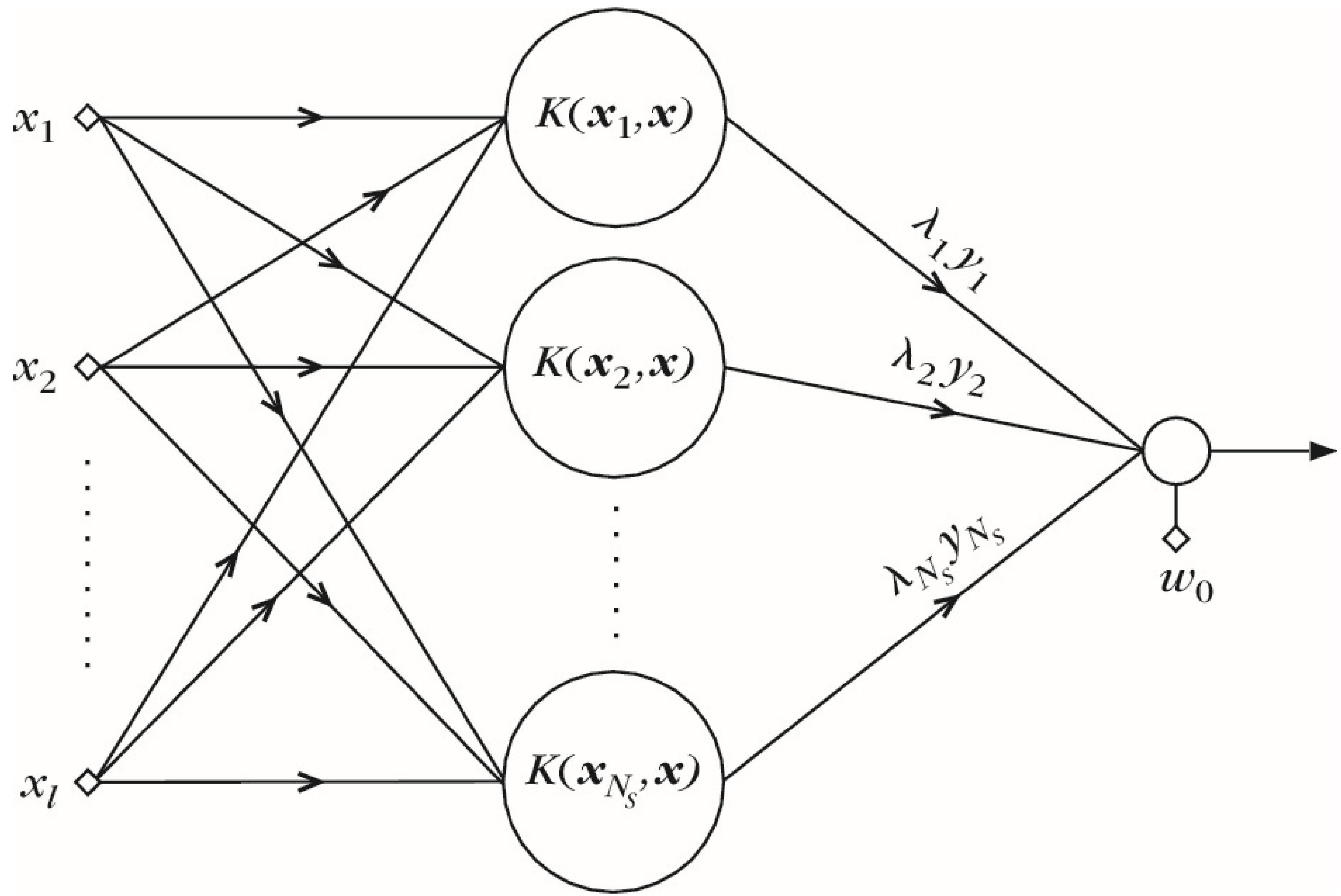
This results to an implicit combination

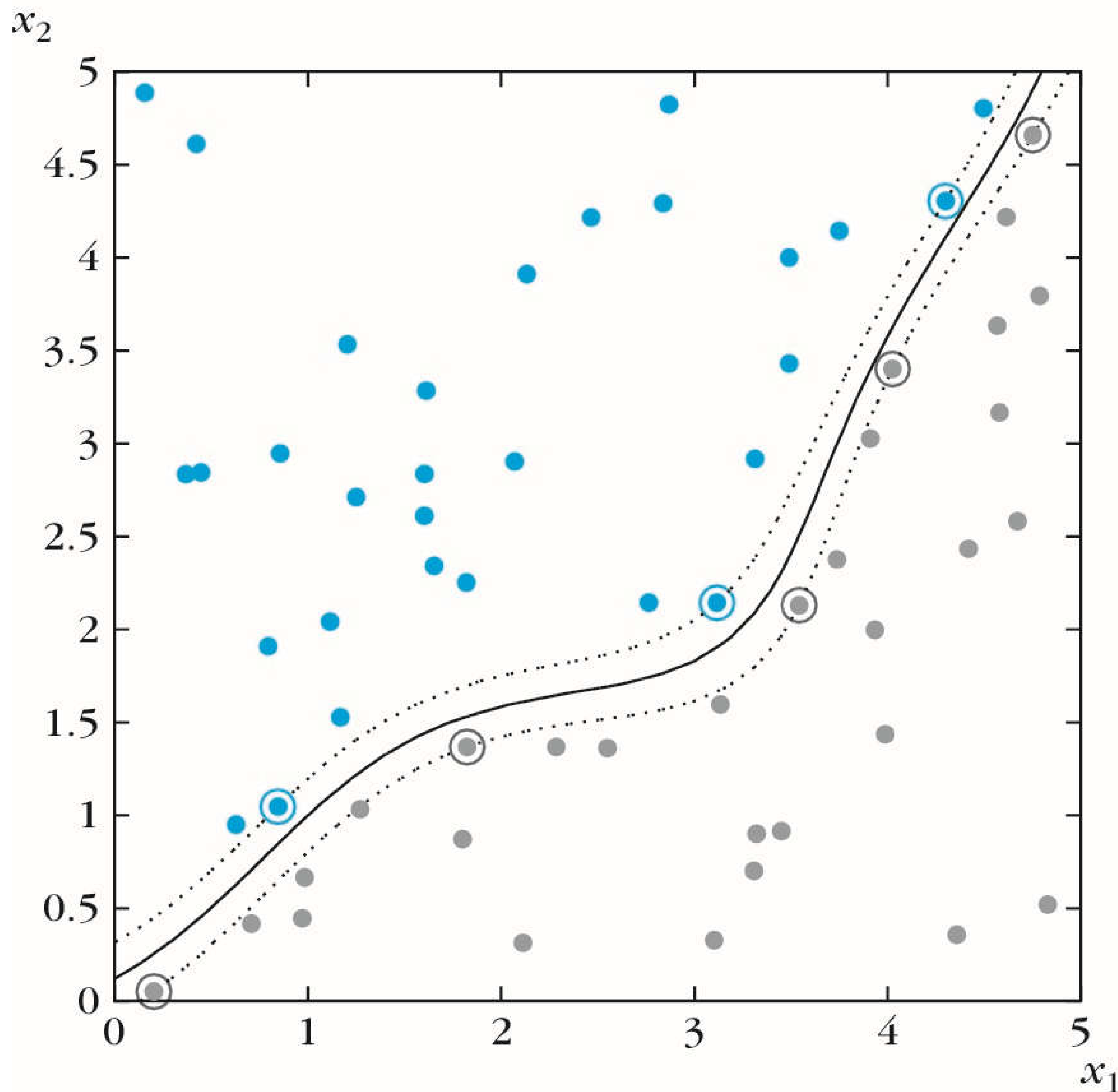
$$\underline{w} = \sum_{i=1}^{N_s} \lambda_i y_i \underline{\varphi}(\underline{x}_i)$$

- Step 3: Assign $\underline{x}$ to

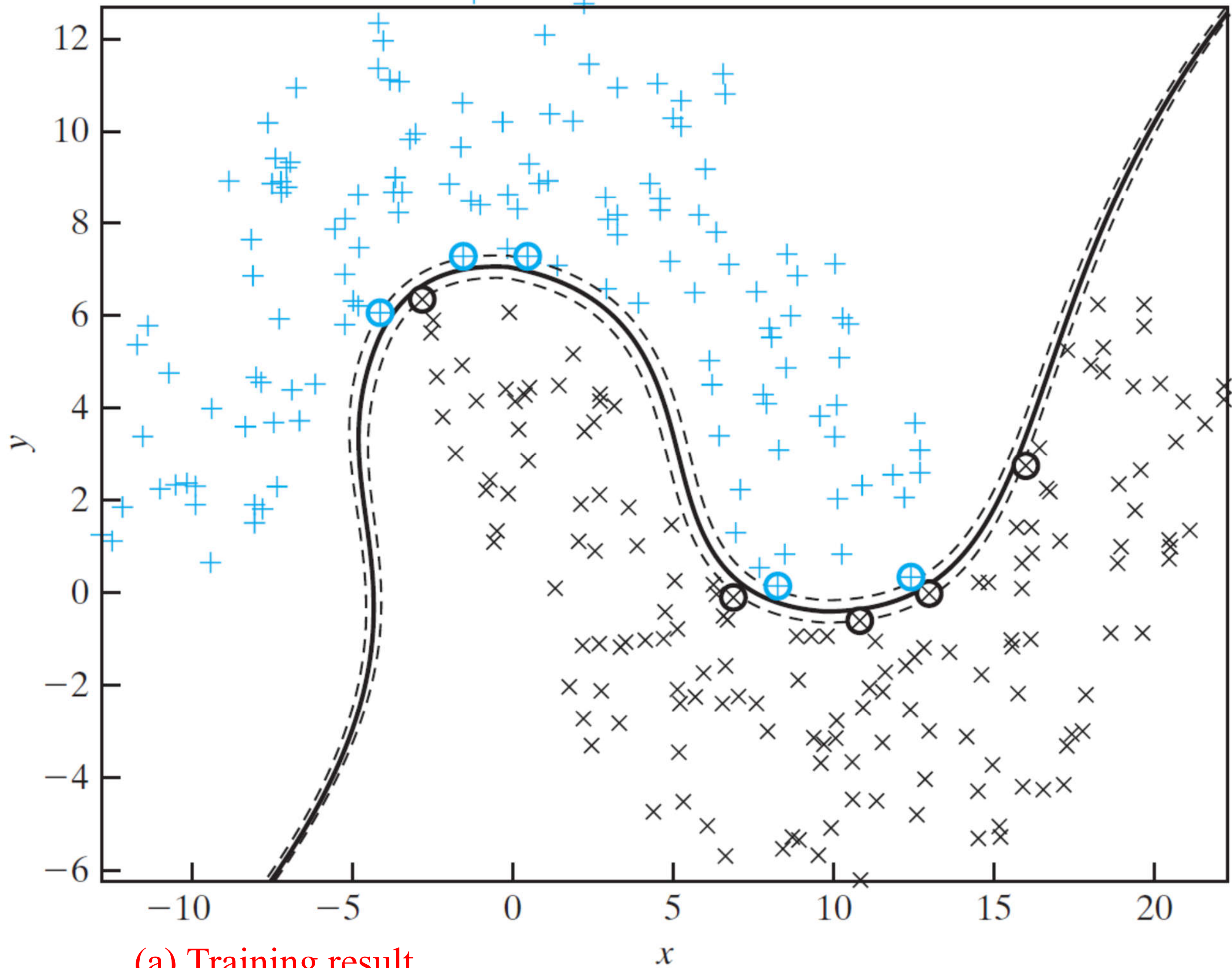
$$\omega_1 \ (\omega_2) \quad \text{if} \quad g(\underline{x}) = \sum_{i=1}^{N_s} \lambda_i y_i K(\underline{x}_i, \underline{x}) + w_0 > (<) \ 0$$

- The SVM Architecture



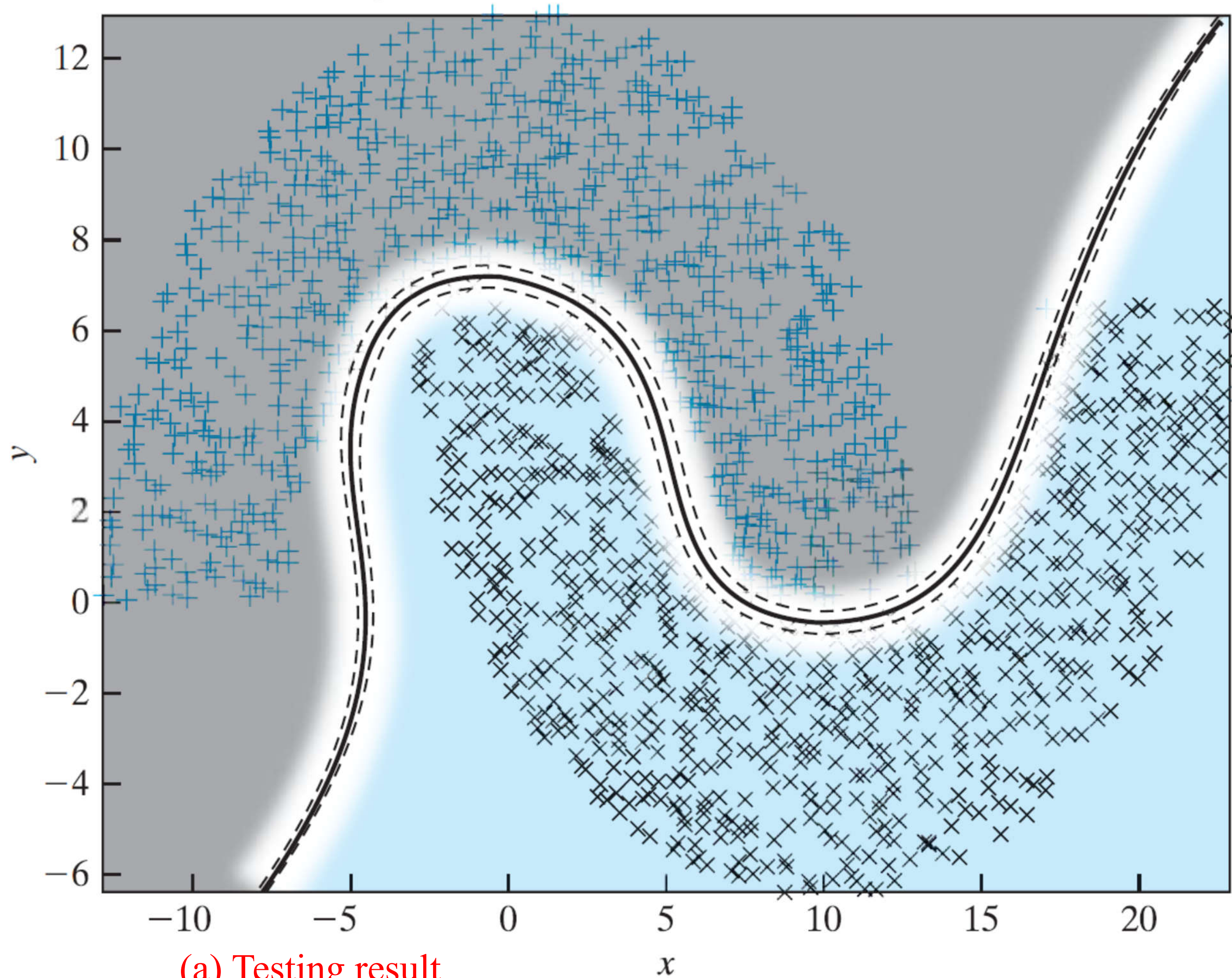


Example of a nonlinear SVM classifier for the case of two nonlinearly separable classes. The Gaussian RBF kernel was used. Dotted lines mark the margin and circled points the support vectors.

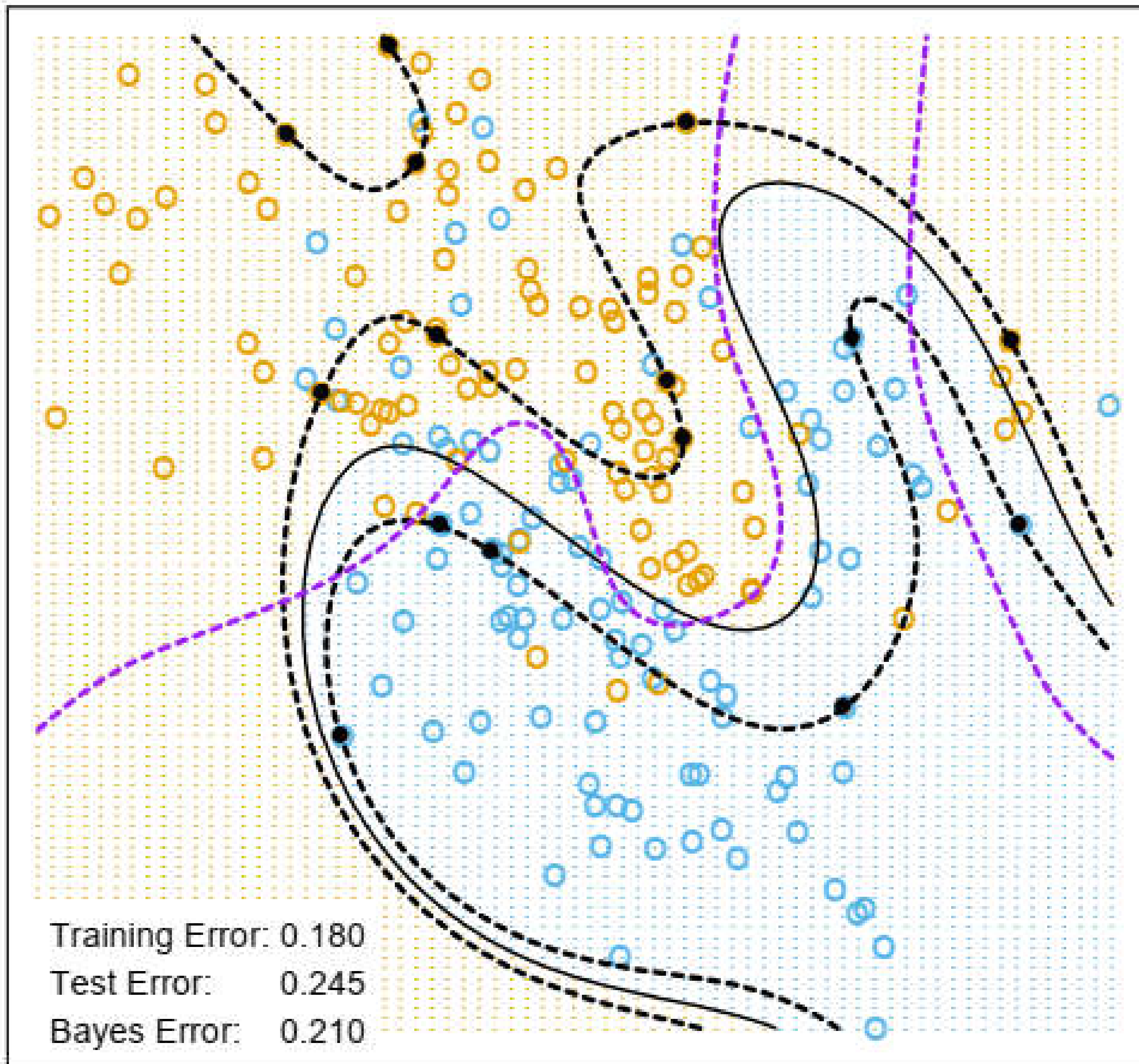


(a) Training result

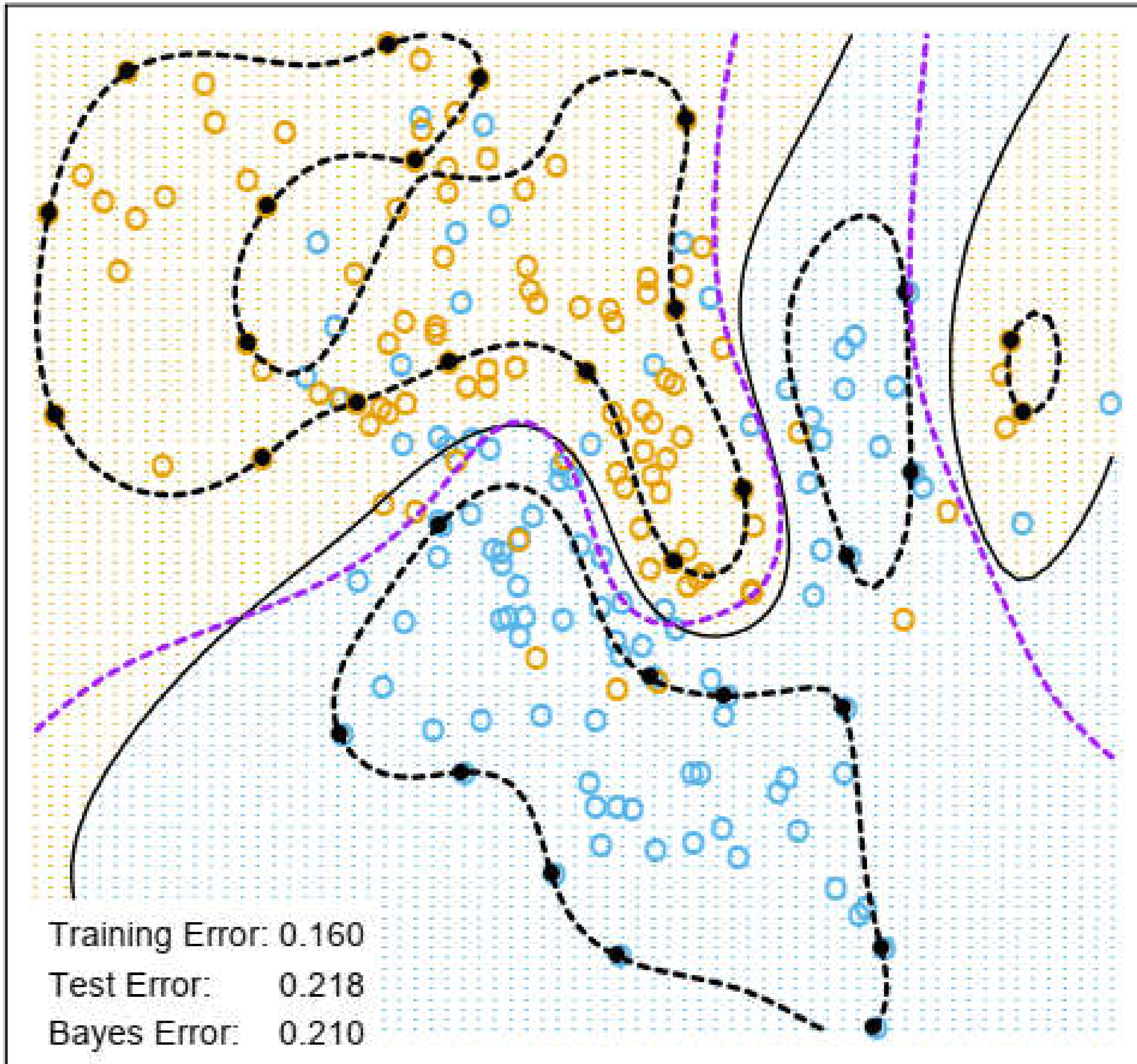
Classification using SVM with distance = −6.5, radius = 10, and width = 6



(a) Testing result



SVM - Degree-4 Polynomial in Feature Space



SVM - Radial Kernel in Feature Space

- **Remarks:** If the kernel function is the RBF, then the architecture is the same as the RBF network architecture. However, the approach followed here is different.
- In the SVM, the number of nodes as well as the centers are the result of the optimization procedure.
- If the hyperbolic tangent function (sigmoid) is chosen as a kernel, the resulting architecture is a special case of a two-layer perceptron. Once more, the number of nodes is the result of the optimization procedure. This is important. Although the SVM architecture is the same as that of a two-layer perceptron, the training procedure is entirely different for the two methods. The same is true for the RBF networks.
- In the SVM the computational complexity is independent of the dimensionality of the kernel space, where the input feature space is mapped. Thus, the curse of dimensionality is bypassed. In other words, one designs in a high-dimensional space without having to adopt explicit models using a large number of parameters, as this would be dictated by the high dimensionality of the space. This also has an influence on the good generalization properties of SVMs.

❖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.

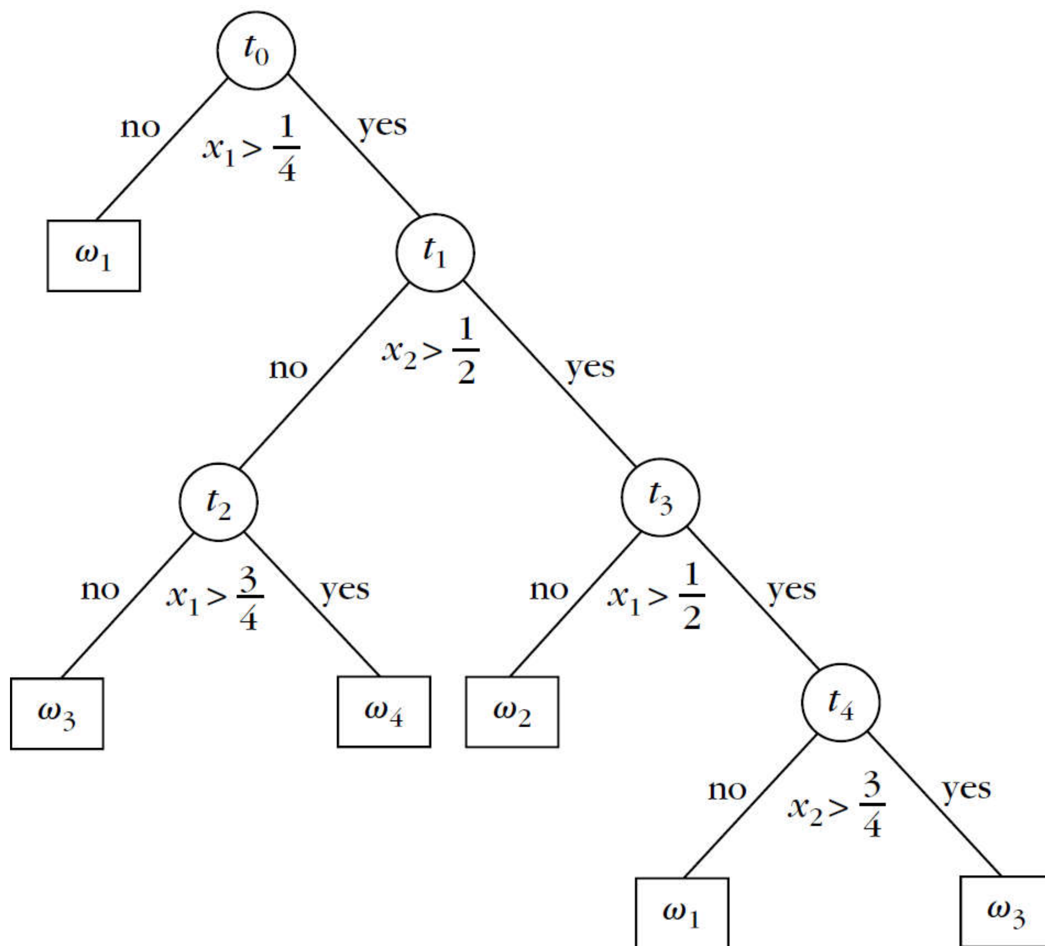
- Upon the arrival of a feature vector, sequential decisions, assigning features to specific regions, are performed along a path of nodes of an appropriately constructed tree.

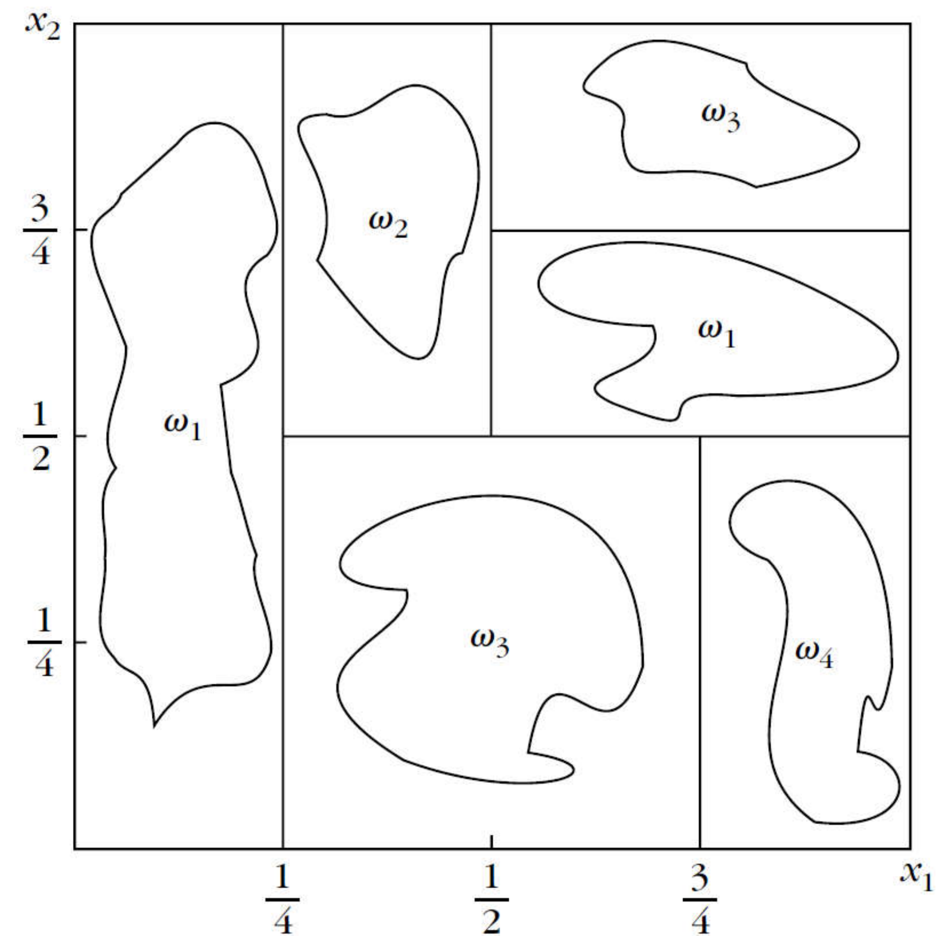
- The sequence of decisions is applied to individual features, and the queries performed in each node are of the type:

$$\text{is feature } x_i \leq \alpha ?$$

where α 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

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

➢ The choice of the specific x_i and the value of the threshold α, 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:

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

and

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

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:

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

- Note: $I(t)$ is the entropy associated with the subset X_t .

Example 4.2

In a tree classification task, the set X_t, associated with node t, contains $N_t = 10$ vectors. Four of these belong to class ω_1, four to class ω_2, and two to class ω_3, in a three-class classification task. The node splitting results into two new subsets X_{tY}, with three vectors from ω_1, and one from ω_2, and X_{tN} with one vector from ω_1, three from ω_2, and two from ω_3. The goal is to compute the decrease in node impurity after splitting.

We have that

$$I(t) = -\frac{4}{10}\log_2\frac{4}{10} - \frac{4}{10}\log_2\frac{4}{10} - \frac{2}{10}\log_2\frac{2}{10} = 1.521$$

$$I(t_Y) = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} = 0.815$$

$$I(t_N) = -\frac{1}{6}\log_2\frac{1}{6} - \frac{3}{6}\log_2\frac{3}{6} - \frac{2}{6}\log_2\frac{2}{6} = 1.472$$

Hence, the impurity decrease after splitting is

$$\Delta I(t) = 1.521 - \frac{4}{10}(0.815) - \frac{6}{10}(1.472) = 0.315$$

- 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:

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

➢ Summary of an OBCT algorithmic scheme:

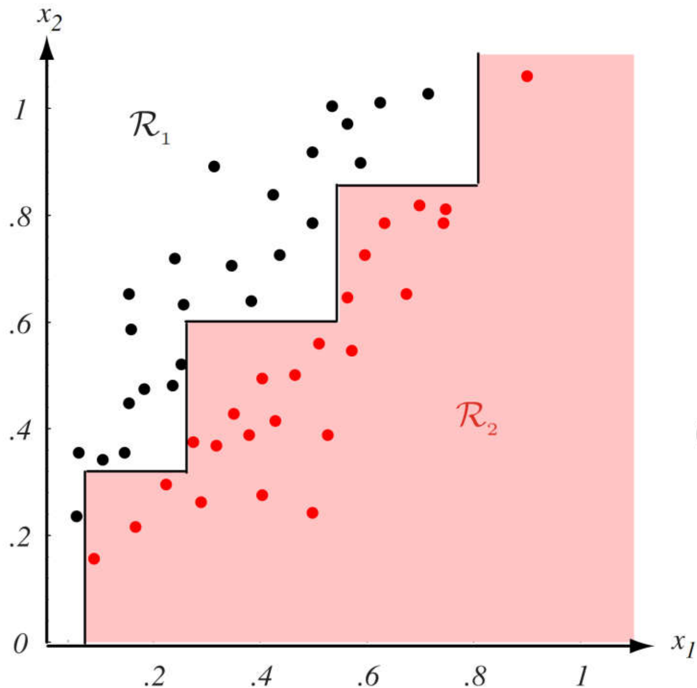
- Begin with the root node, i.e., $X_t = X$
- For each new node t
  - * For every feature $x_k, k = 1, 2, \ldots, l$
    - For every value $\alpha_{kn}, n = 1, 2, \ldots, N_{tk}$
      - Generate X_{tY} and X_{tN} according to the answer in the question: is $x_k(i) \leq \alpha_{kn},\ i = 1, 2, \ldots, N_t$
      - 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 $\alpha_{k_0 n_0}$ 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 $x_{k_0} \leq \alpha_{k_0 n_0}$
- End

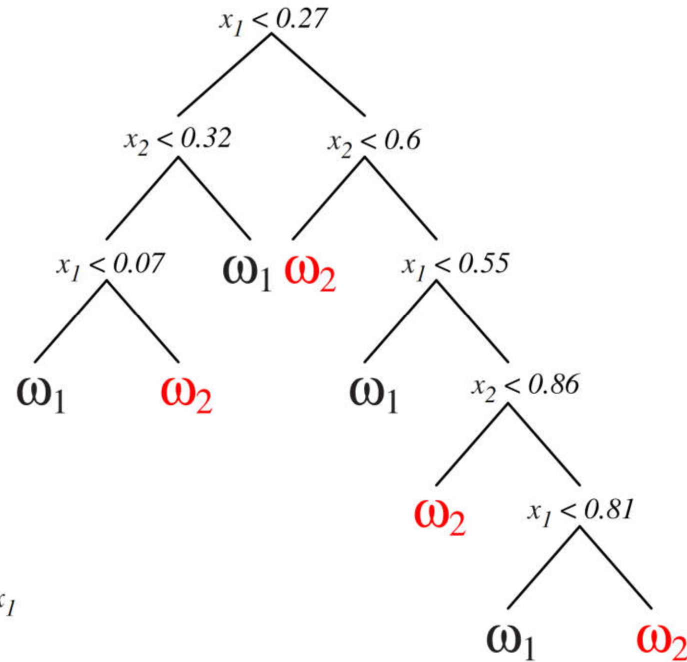
❖ Remarks:

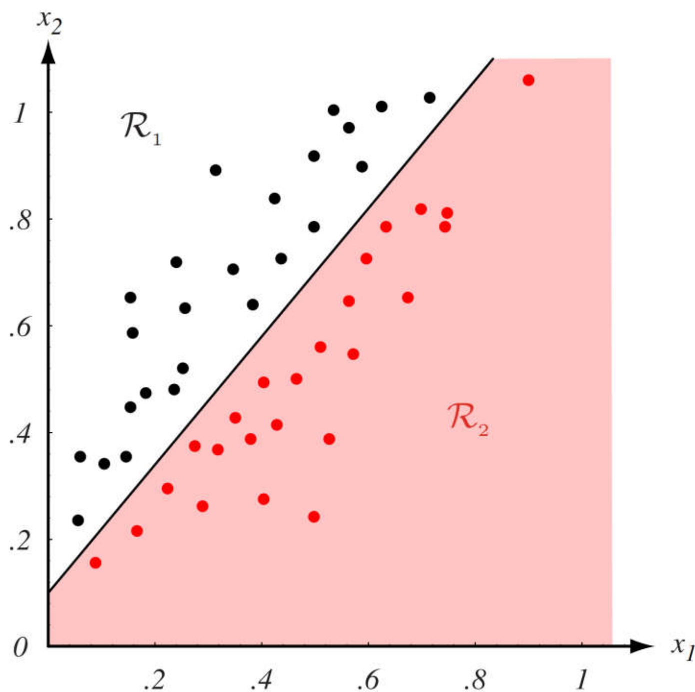
- A variety of node impurity measures can be defined.
- A critical factor in the design is the size of the tree. Usually one grows a tree to a large size and then applies various pruning techniques.
- Decision trees belong to the class of unstable classifiers. This can be overcome by a number of "averaging" techniques. Bagging is a popular technique. Using bootstrap techniques in X, various trees are constructed, T_i, $i=1, 2, \ldots, B$ for B variants, $X_1, X_2, \ldots, X_B$, of the training set,. The decision is taken according to a majority voting rule.
- More general partition of the feature space, via hyperplanes not parallel to the axis, is possible via questions of the type:

$$\text{Is } \sum_{k=1}^{l} c_k x_k \leq \alpha \,?$$

Feature choice







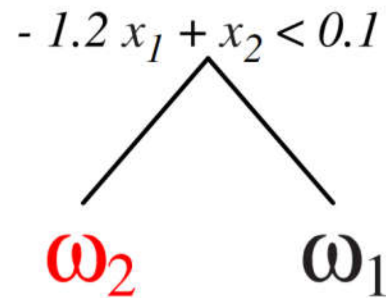
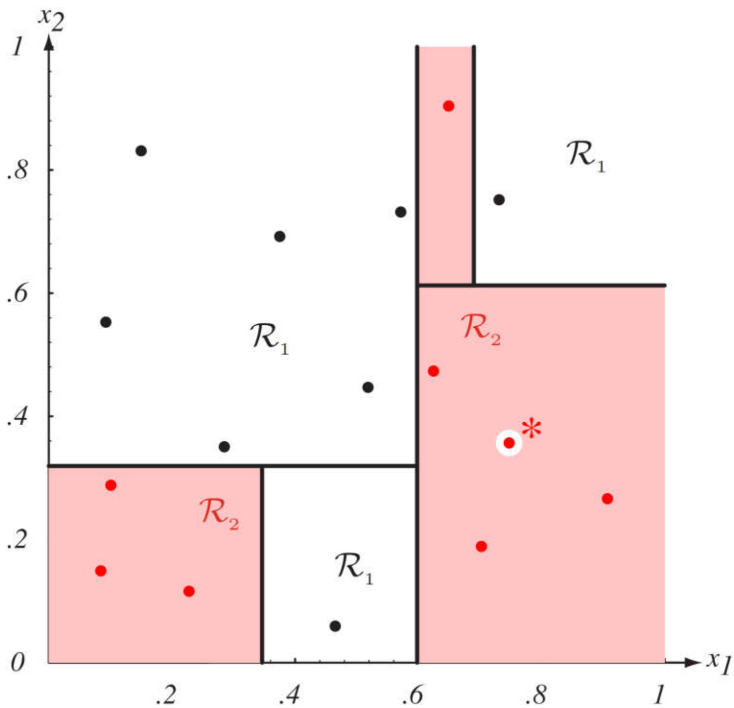
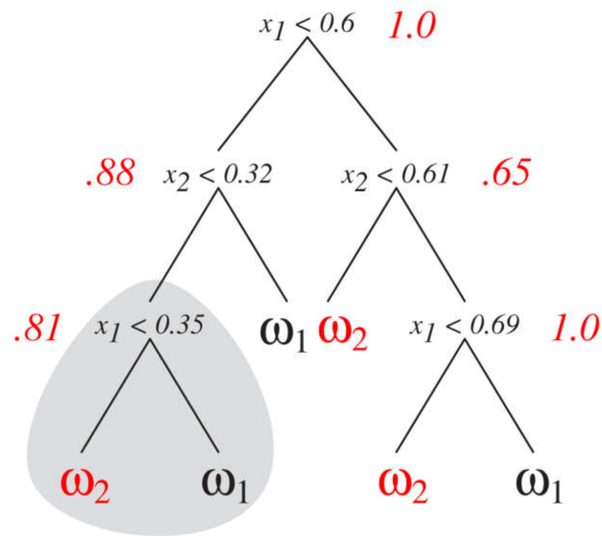
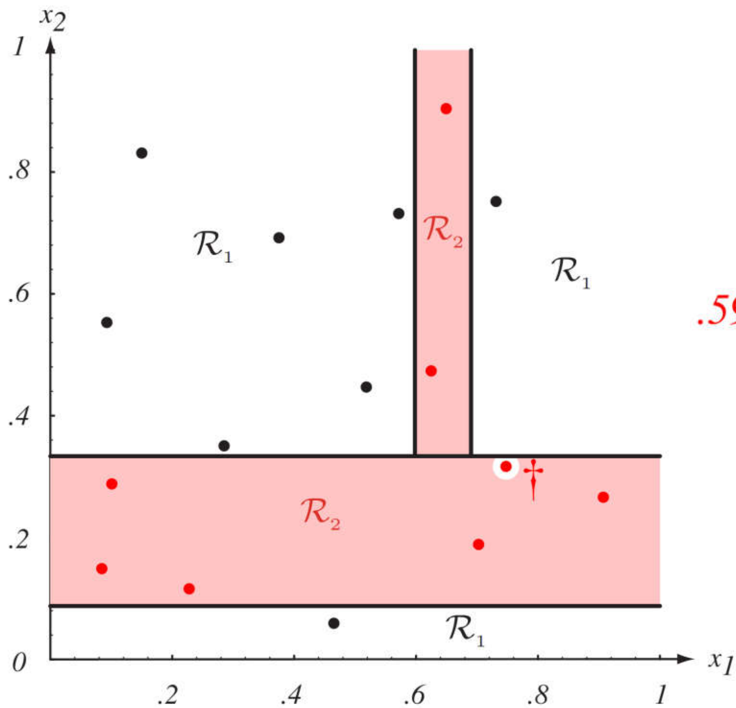


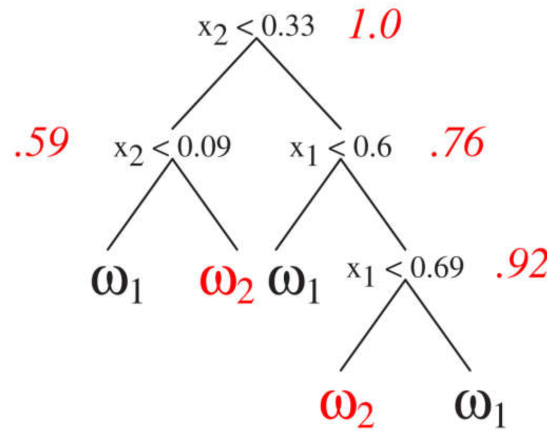
Figure 8.5: If the class of node decisions does not match the form of the training data, a very complicated decision tree will result, as shown at the top. Here decisions are parallel to the axes while in fact the data is better split by boundaries along another direction. If however “proper” decision forms are used (here, linear combinations of the features), the tree can be quite simple, as shown at the bottom.





Entropy impurity at nonterminal nodes is shown in red and impurity at each leaf node is 0





Instability or sensitivity of tree to training points; alteration of a single point leads to a very different tree; this is due to discrete & greedy nature of CART (Classification And Regression Trees)

Example

Consider the following table

Day	Outlook	Temp.	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Weak	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Strong	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Example

- We want to build a decision tree for the tennis matches
- The schedule of matches depend on the weather (Outlook, Temperature, Humidity, and Wind)
- Calculating the information gains for each of the weather attributes:
  - For the Outlook
  - For the Temperature
  - For the Humidity
  - For the Wind

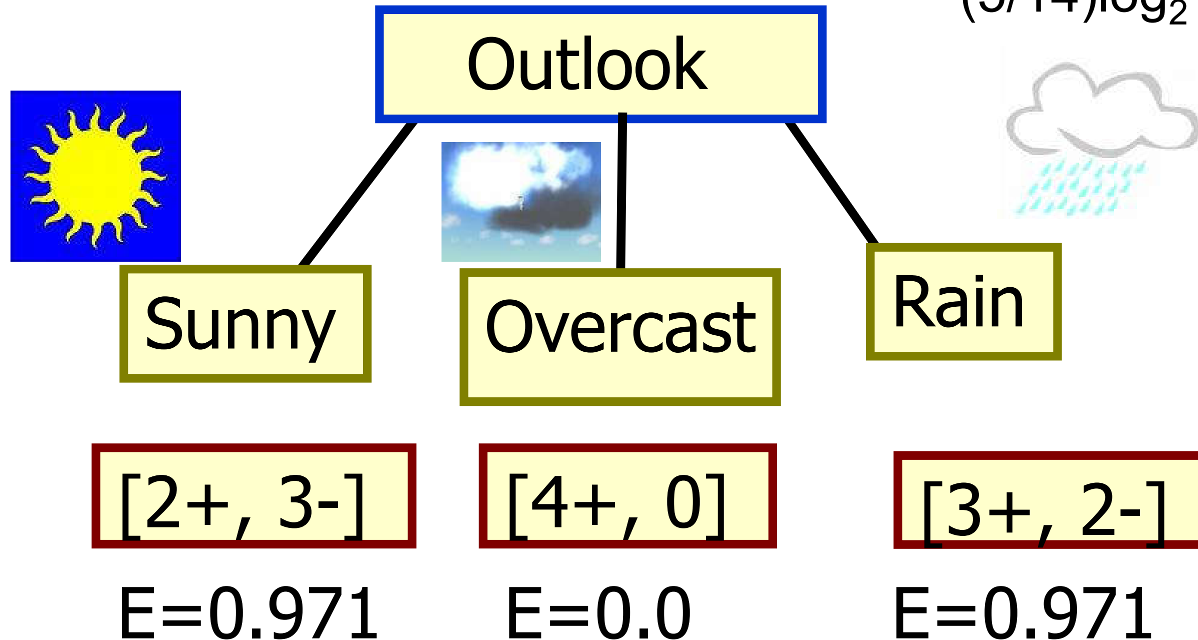
Information gain (IG) *measures how much "information" a feature gives us about the class.*

Information Gain = entropy(parent) – [average entropy(children)]

For the Outlook

[positive, Negative] → S=[9+,5-]
E=0.940

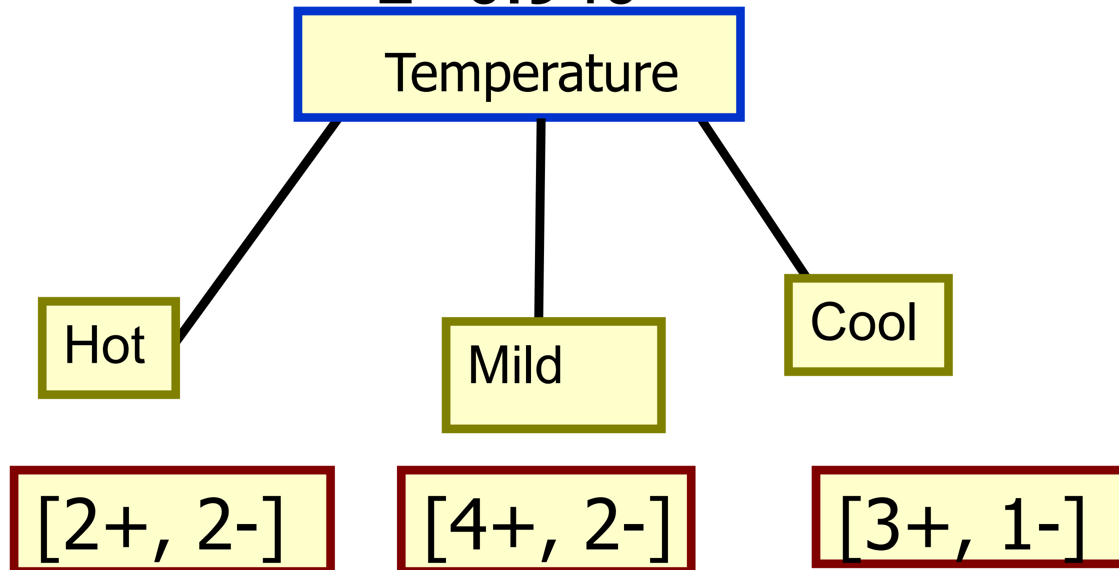
Entropy(rootNode.subset) = $-(9/14)\log_2(9/14) - (5/14)\log_2(5/14)=0.940$



Gain(S,Outlook)

=0.940 – (5/14)*0.971 – (4/14)*0.0 – (5/14)*0.0971
=0.247

For the Temperature



Gain(S,Temperature)

=0.029

For the Humidity

S=[9+,5-]
E=0.940

Humidity

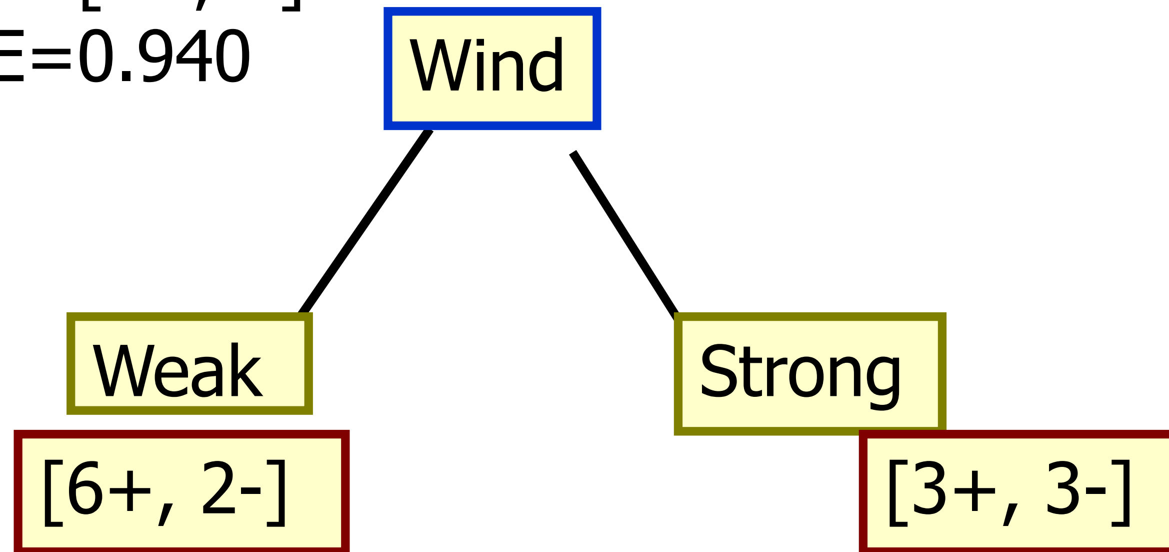
High → [3+, 4-]

Normal → [6+, 1-]

Gain(S,Humidity)

=0.940-(7/14)*0.985 – (7/14)*0.592
=0.151

For the Wind



Gain(S,Wind):

=0.940 - (8/14)*0.811 - (6/14)*1.0
=0.048

Selecting the Next Attribute

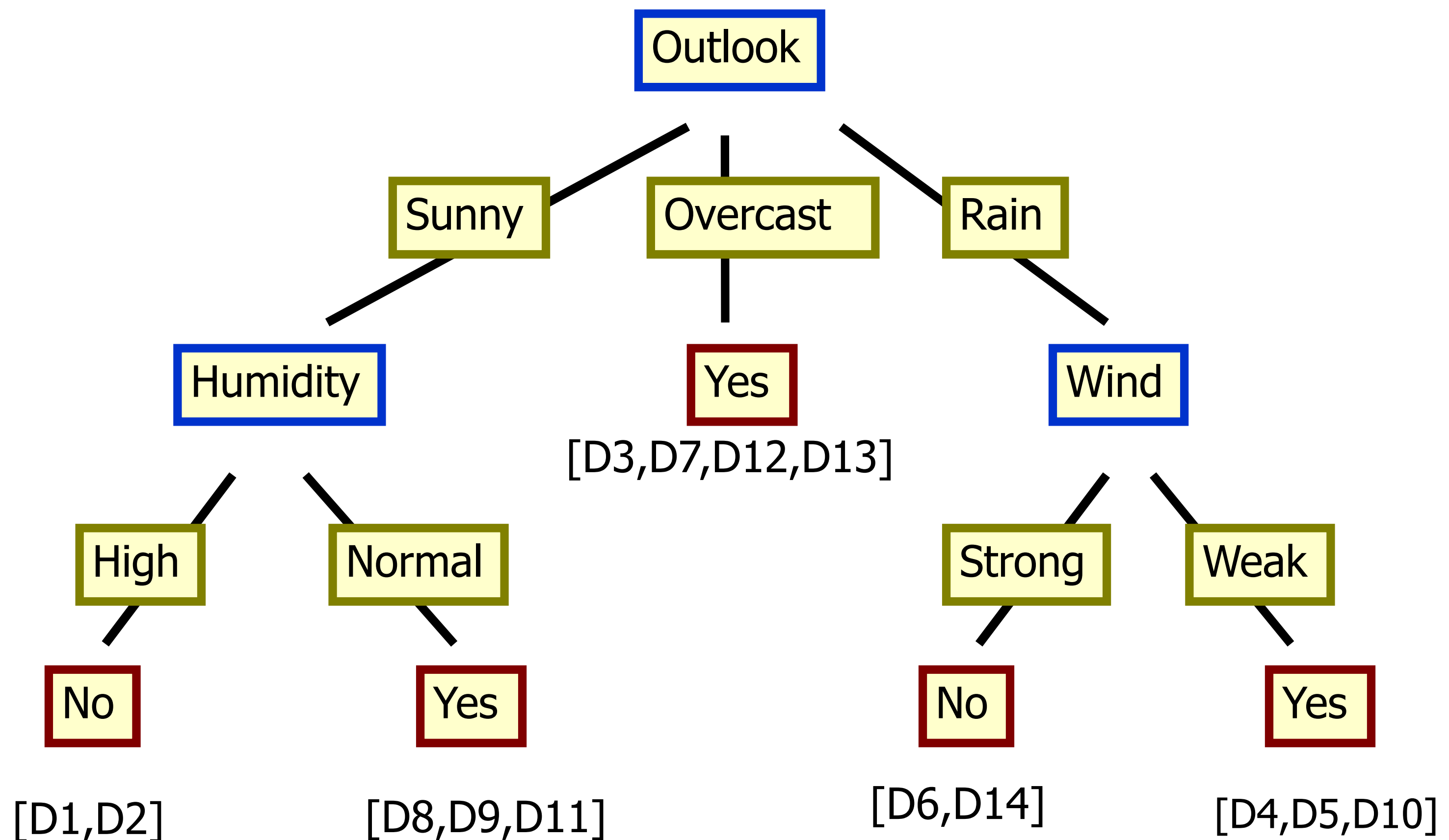
The information gain values for the 4 attributes are:

- Gain(S,Outlook) =**0.247**
- Gain(S,Humidity) =0.151
- Gain(S,Wind) =0.048
- Gain(S,Temperature) =0.029

where S denotes the collection of training examples

Complete tree

- Then here is the complete tree:

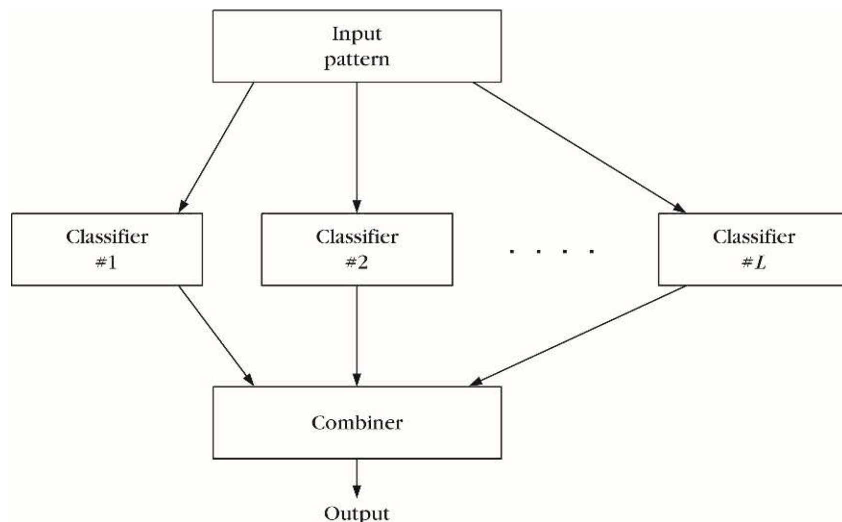


❖ Combining Classifiers

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:



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

❖ Product Rule: Assign $\underline{x}$ to the class ω_i : $i = \arg\max_{k} \prod_{j=1}^{L} P_j(\omega_k \mid \underline{x})$

where $P_j(\omega_k \mid \underline{x})$ is the respective posterior prob. of the jth classifier.

Proof: By minimizing the average Kullback–Leibler (KL) distance between probabilities, by employing Lagrange multiplies optimization.

$$D_{av} = \frac{1}{L} \sum_{j=1}^{L} D_j , \quad D_j = \sum_{i=1}^{M} P(\omega_i \mid \underline{x}) \ln \frac{P(\omega_i \mid \underline{x})}{P_j(\omega_i \mid \underline{x})}$$

Taking into account that $\sum_{i=1}^{M} P_j(\omega_i \mid \underline{x}) = 1$ Optimization →

$$P(\omega_i \mid \underline{x}) = \frac{1}{C} \prod_{j=1}^{L} \left(P_j(\omega_i \mid \underline{x}) \right)^{\frac{1}{L}}, \quad C = \sum_{i=1}^{M} \prod_{j=1}^{L} \left(P_j(\omega_i \mid \underline{x}) \right)^{\frac{1}{L}}$$

Neglecting all the terms common to all classes classification rule is Assign $\underline{x}$ to the class:

$$\max_{\omega_i} \prod_{j=1}^{L} P_j(\omega_i \mid \underline{x})$$

❖ Sum Rule: Assign $\underline{x}$ to the class ω_i : $\quad i = \arg\max_{k} \sum_{j=1}^{L} P_j\left(\omega_k \mid \underline{x}\right)$

Proof: Using the alternative KL distance formulation.

$$D_{av} = \frac{1}{L}\sum_{j=1}^{L} D_j, \quad D_j = \sum_{i=1}^{M} P_j(\omega_i \mid \underline{x}) \ln \frac{P_j(\omega_i \mid \underline{x})}{P(\omega_i \mid \underline{x})}$$

Optimization $\rightarrow$ $$P(\omega_i \mid \underline{x}) = \frac{1}{L}\sum_{j=1}^{L} P_j(\omega_i \mid \underline{x})$$

- Although the product rule often produces better results than the sum rule, it may lead to less reliable results when the outputs of some of the classifiers result in values close to zero.

❖ 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 (adaptive boosting) Algorithm.

Let the training data be $\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}$ with $y_i \in \{-1,1\}$, $i = 1, 2, \ldots, N$.

Construct an optimally designed classifier of the form:

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

where:

$$F(\underline{x}) = \sum_{k=1}^{K} \alpha_k \phi(\underline{x}; \underline{\theta}_k)$$

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

$$\phi(\underline{x}; \underline{\theta}_k) \in \{-1, 1\}$$

$\underline{\theta}$ is a parameter vector.

- The essence of the method.

Design the series of classifiers:

$$\phi(\underline{x};\underline{\theta}_1),\ \phi(\underline{x};\underline{\theta}_2),\ \ldots,\ \phi(\underline{x};\underline{\theta}_K)$$

The parameter vectors

$$\underline{\theta}_k,\ k=1,2,\ldots,K$$

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.

  - Optimizing to find the unknown parameters (highly complex) :

$$\arg\min_{\alpha_k;\boldsymbol{\theta}_k,k:1,K}\sum_{i=1}^{N}\exp\left(-y_i F(\mathbf{x}_i)\right)$$

  - It penalizes the samples that are wrongly classified much more heavily than those correctly classified.

Optimization

- Stage-wise optimization (suboptimal)
- At each step, a new parameter is considered and optimization is carried out with respect to this parameter, leaving unchanged the previously optimized ones.
- We define $F_m(\boldsymbol{x})$ to denote the result of the partial sum up to m terms.

$$F_m(\underline{x}) = \sum_{k=1}^{m} \alpha_k \phi\left(\underline{x}; \underline{\theta}_k\right), m = 1, 2, \ldots, K$$

- Recursion form is: $F_m(\underline{x}) = F_{m-1}(\underline{x}) + \alpha_m \phi\left(\underline{x}; \underline{\theta}_m\right)$
- The task at step m is to compute

$$(\alpha_m, \underline{\theta}_m) = \arg\min_{\alpha, \boldsymbol{\theta}} J(\alpha, \underline{\theta})$$

- Where the cost function is defined as

$$J(\alpha, \underline{\theta}) = \sum_{i=1}^{N} \exp\left(-y_i \left(F_{m-1}(\underline{x}_i) + \alpha \phi\left(\underline{x}_i; \underline{\theta}\right)\right)\right)$$

- First, α will be considered constant, and the cost will be optimized with respect to the base classifier $\phi(\underline{x}_i;\underline{\theta})$.
- That is, the cost to be minimized is now simplified to

$$\underline{\theta}_m = \arg\min_{\underline{\theta}} \sum_{i=1}^{N} \exp\left(-y_i F_{m-1}(\underline{x}_i)\right) \exp\left(-y_i \alpha \phi\left(\underline{x}_i;\underline{\theta}\right)\right)$$

$$= \arg\min_{\underline{\theta}} \sum_{i=1}^{N} w_i^{(m)} \exp\left(-y_i \alpha \phi\left(\underline{x}_i;\underline{\theta}\right)\right); \quad w_i^{(m)} \equiv \exp\left(-y_i F_{m-1}(\underline{x}_i)\right)$$

- Since each $w_i^{(m)}$ depends neither on α nor on $\phi\left(\underline{x}_i;\underline{\theta}\right)$, it can be regarded as a weight associated with the sample point $\underline{x}_i$.
- Since the base classifier is binary $\phi\left(\underline{x}_i;\underline{\theta}\right), \in \{-1,1\}$ it is easy to see that minimizing is equivalent to designing the optimal classifier $\phi\left(\underline{x}_i;\underline{\theta}_m\right)$ so that the *weighted* empirical error (the fraction of the training samples that are wrongly classified) is minimum.

That is, $$\underline{\theta}_m = \arg\min_{\underline{\theta}} \left\{ P_m = \sum_{i=1}^{N} w_i^{(m)} I\left(1 - y_i \phi\left(\underline{x}_i; \underline{\theta}\right)\right) \right\}.$$

- To guarantee that the value of the weighted empirical error rate remains in the interval [0, 1], the weights must sum to one.

$$\sum_{y_i \phi\left(\underline{x}_i; \underline{\theta}_m\right) < 0} w_i^{(m)} = P_m, \qquad \sum_{y_i \phi\left(\underline{x}_i; \underline{\theta}_m\right) > 0} w_i^{(m)} = 1 - P_m$$

- Combining above Eqs. the optimum value, α_m, results from

$$\alpha_m = \arg\min_{\alpha} \left\{ \exp(-\alpha)(1 - P_m) + \exp(\alpha)(P_m) \right\}$$

- Taking the derivative with respect to α and equating to zero, we obtain

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

- Once α_m and $\phi\left(\underline{x}_i; \underline{\theta}_m\right)$ have been computed, the weights for the next step are readily available via the iteration.

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

$$w_i^{(m+1)} = \frac{w_i^{(m)} \exp\left(-y_i \alpha_m \phi\left(\underline{x}_i ; \underline{\theta}_m\right)\right)}{Z_m}$$

- Z_m is a normalizing factor common for all samples.
- $Z_m = \sum_{i=1}^{N} w_i^{(m)} \exp\left(-y_i \alpha_m \phi\left(\underline{x}_i ; \underline{\theta}_m\right)\right), \quad \alpha_m = \frac{1}{2} \ln \frac{1-P_m}{P_m}$

  where $P_m < 0.5$ (by assumption) is the error rate of the optimal classifier $\phi\left(\underline{x} ; \underline{\theta}_m\right)$ at stage m. Thus $\alpha_m > 0$.
- The term: $\boxed{\exp\left(-y_i \alpha_m \phi\left(\underline{x}_i ; \underline{\theta}_m\right)\right)}$

  takes a <u>large</u> value if $y_i \phi\left(\underline{x}_i ; \underline{\theta}_m\right) < 0$ (wrong classification) and a <u>small</u> value in the case of correct classification

  $$\{y_i \phi\left(\underline{x}_i ; \underline{\theta}_m\right) > 0\}$$
- 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: $w_i^{(1)} = \frac{1}{N}, \; i = 1, 2 \ldots, N$
  - Initialize: $m = 1$
  - Repeat
    - Compute optimum $\boldsymbol{\theta}_m$ in $\phi(\cdot; \boldsymbol{\theta}_m)$ 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
      - $w_i^{(m+1)} = w_i^{(m)} \exp\left(-y_i \alpha_m \phi(\boldsymbol{x}_i; \boldsymbol{\theta}_m)\right)$
      - $Z_m = Z_m + w_i^{(m+1)}$
    - 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.
  - $f(\cdot) = \text{sign}(\sum_{k=1}^{K} \alpha_k \phi(\cdot, \theta_k))$

Properties

- Boosting has relative immunity to overfitting.
- It has been verified that, although the number of terms, K, and consequently the associated number of parameters can be quite high, the error rate on a test set does not increase but keeps decreasing and finally *levels off at a certain value*.
- It has been observed that the test error continues to decrease long after the error on the training set has become zero.

Example 4.3:

Let us consider a two-class classification task. The data reside in the 20-dimensional space and obey a Gaussian distribution of **unit covariance** matrix and **mean** values $[-a,-a, \ldots ,-a]^T$, $[a, a, . . . , a]^T$, respectively, for each class, where $a= 2/\sqrt{20}$. The training set consists of 200 points (100 from each class) and the test set of 400 points (200 from each class) independently generated from the points of the training set.

To design a classifier using the AdaBoost algorithm, we chose as a seed the weak classifier known as **stump**. This is a very "naive" type of tree, consisting of a single node, and classification of a feature vector $\underline{\boldsymbol{x}}$ is achieved on the basis of the value of only one of its features, say, x_i . Thus, if $x_i < 0$, $\underline{x}$ is assigned to class A. If $x_i > 0$, it is assigned to class B.

The decision about the choice of the specific feature, x_i , to be used in the classifier was randomly made. Such a classifier results in a training error rate slightly better than 0.5.

The AdaBoost algorithm was run on the training data for 2000 iteration steps. Figure 4.30 verifies the fact that the training error rate converges to zero very fast. The test error rate keeps decreasing even after the training error rate becomes zero and then levels off at around 0.05.

Figure 4.31 shows the margin distributions, over the training data points, for four different training iteration steps. It is readily observed that the algorithm is indeed greedy in increasing the margin. Even when only 40 iteration steps are used for the AdaBoost training, the resulting classifier classifies the majority of the training samples with large margins.

Using 200 iteration steps, all points are correctly classified (positive margin values), and the majority of them with large margin values. From then on, more iteration steps further improve the margin distribution by pushing it to higher values.

The margin of a training example with respect to a classifier f is defined as

$$\text{margin}_f(\underline{x}, y) = \frac{yF(\underline{x})}{\sum_{k=1}^{K}\alpha_k} = \frac{y\sum_{k=1}^{K}\alpha_k\phi_k\left(\underline{x};\underline{\theta}_k\right)}{\sum_{k=1}^{K}\alpha_k}$$

The margin lies in the interval [-1, 1] and is positive if and only if the respective pattern is classified correctly.

❖ Remarks:

- Training error rate tends to zero after a few iterations. The test error levels to some value.
- AdaBoost minimizes the upper bound of the training error by properly choosing the optimal weak classifier and voting weight.
- AdaBoost is greedy in reducing the margin that samples leave from the decision surface.