Ch 4: Non Linear Classifiers

- The XOR problem

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>XOR</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>B</td>
</tr>
</tbody>
</table>
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)

\[ g_1(x) = g_2(x) = 0 \]

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

\[ y_i = f(g_i(x)) = \begin{cases} 0 & i = 1, 2 \\ 1 & \end{cases} \]

depending on the position of \( x \).

• Phase 2: Find the position of \( x \) w.r.t. both lines, based on the values of \( y_1, y_2 \).
• Equivalently: The computations of the first phase perform a mapping \( \underline{x} \rightarrow \underline{y} = [y_1, y_2]^T \)

<table>
<thead>
<tr>
<th>1st phase</th>
<th>2nd phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>(x_2)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The decision is now performed on the transformed \( y \) 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)

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} = 0 \]
\[ g_2(x) = x_1 + x_2 - \frac{3}{2} = 0 \]
\[ g(y) = y_1 - 2y_2 - \frac{1}{2} = 0 \]
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,

\[ \overline{x} \in R^l \]
\[ \overline{x} \rightarrow \overline{y} = [y_1, \ldots, y_p]^T, \quad y_i \in \{0, 1\} \quad i = 1, 2, \ldots, p \]
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 $x$ 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 $g_1(x) = 0$
to the (-) side of $g_2(x) = 0$
to the (+) side of $g_3(x) = 0$

$-y_1 - y_2 + y_3 + 0.5 = 0$
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 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.
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 $y \in R^p$ 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

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

\[
f(x) = \frac{2}{1 + \exp(-ax)} - 1
\]

\[
f(x) = c \frac{1 - \exp(-ax)}{1 + \exp(-ax)} = c \cdot \tanh\left(\frac{ax}{2}\right)
\]
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
The weight vector (including the threshold) of the \(j^{th}\) neuron in the \(r^{th}\) layer, which is a vector of dimension \(k_{r-1} + 1\) is defined as:

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

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

\[
\mathbf{w}_j^r (\text{new}) = \mathbf{w}_j^r (\text{old}) + \Delta \mathbf{w}_j^r
\]

\[
\Delta \mathbf{w}_j^r = -\mu \frac{\partial J}{\partial \mathbf{w}_j^r}
\]
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.
The Algorithm

Computational flow

1st  2nd  3rd

Input

Pattern

Batch

Desired output
A major problem: The algorithm may converge to a local minimum
The Cost function choice

Examples:

• The Least Squares

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

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

\[ i = 1, 2, \ldots, N \]

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

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

\[ v_j^r(i) = \sum_{k=1}^{k_r-1} w_{jk}^r y_k^{r-1}(i) + w_{jo}^r = \sum_{k=0}^{k_r-1} w_{jk}^r y_k^{r-1}(i) \]

\[ \frac{\partial \mathcal{E}(i)}{\partial w_j^r} = \frac{\partial \mathcal{E}(i)}{\partial v_j^r(i)} \frac{\partial v_j^r(i)}{\partial w_j^r} \quad \text{we define} \quad \frac{\partial \mathcal{E}(i)}{\partial v_j^r(i)} \equiv \delta_j^r(i) \]
\[ \frac{\partial}{\partial w^r_j} v^r_j(i) = \begin{bmatrix} \frac{\partial}{\partial w^r_{j0}} v^r_j(i) \\ \vdots \\ \frac{\partial}{\partial w^r_{jk}} v^r_j(i) \end{bmatrix} = y^{r-1}(i) \]

where

\[ y^{r-1}(i) = \begin{bmatrix} +1 \\ y^{r-1}_{1}(i) \\ \vdots \\ y^{r-1}_{k}(i) \end{bmatrix} \]

\[ \Delta w^r_j = -\mu \sum_{i=1}^{N} \delta^r_j(i) y^{r-1}(i) \]

**Computation of \( \delta^r_j(i) \) for the Cost Function**

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

1. \( r = L \)

\[ \delta^L_j(i) = \frac{\partial E(i)}{\partial v^L_j(i)} \]

\[ E(i) = \frac{1}{2} \sum_{m=1}^{k_L} e_m^2(i) = \frac{1}{2} \sum_{m=1}^{k_L} (f(v^L_m(i)) - y_m(i))^2 \]
2. \( r < L \)

\[
\frac{\partial \mathcal{E}(i)}{\partial \nu_{j}^{r-1}(i)} = \sum_{k=1}^{k_r} \frac{\partial \mathcal{E}(i)}{\partial \nu_{k}^{r}(i)} \frac{\partial \nu_{k}^{r}(i)}{\partial \nu_{j}^{r-1}(i)}
\]

\[
\delta_{j}^{r-1}(i) = \sum_{k=1}^{k_r} \delta_{k}^{r}(i) \frac{\partial \nu_{k}^{r}(i)}{\partial \nu_{j}^{r-1}(i)}
\]

But

\[
\frac{\partial \nu_{k}^{r}(i)}{\partial \nu_{j}^{r-1}(i)} = \partial \left[ \sum_{m=0}^{k_r-1} \omega_{km}^{r} \nu_{m}^{r-1}(i) \right]
\]

with

\[
\nu_{m}^{r-1}(i) = f(\nu_{m}^{r-1}(i))
\]

Hence,

\[
\frac{\partial \nu_{k}^{r}(i)}{\partial \nu_{j}^{r-1}(i)} = \omega_{kj}^{r} f'(\nu_{j}^{r-1}(i))
\]
\[
\delta_{j}^{r-1}(i) = \left[ \sum_{k=1}^{k_r} \delta_{k}^{r}(i) w_{kj}^{r} \right] f'(v_{j}^{r-1}(i))
\]

\[
\delta_{j}^{r-1}(i) = e_{j}^{r-1}(i) f'(v_{j}^{r-1}(i)) \quad \text{where} \quad f'(x) = af(x)(1 - f(x))
\]

\[
e_{j}^{r-1}(i) = \sum_{k=1}^{k_r} \delta_{k}^{r}(i) w_{kj}^{r}
\]

\[
\delta_{j}^{L}(i) = e_{j}(i) f''(v_{j}^{L}(i))
\]

**The Backpropagation Algorithm**

1- **Initialization**

2- **Forward computations:**

3- **Backward computations:**

4- **Update the weights**

\[
\Delta w_{j}^{r} = -\mu \sum_{i=1}^{N} \delta_{j}^{r}(i) y_{j}^{r-1}(i)
\]

Demo: nnd11bc nnd11fa nnd11gn
VARIATIONS ON THE BACKPROPAGATION THEME

- Use Momentum term

\[ w_{j}^{r} \text{ (new)} = w_{j}^{r} \text{ (old)} + \Delta w_{j}^{r} \text{ (new)} \]

\[ \Delta w_{j}^{r} \text{ (new)} = \alpha \Delta w_{j}^{r} \text{ (old)} - \mu \sum_{i=1}^{N} \delta_{j}^{r} (i) y_{r-1}^{r-1} (i) \]

Adaptive learning factor \( \mu \)

\[ \frac{J(t)}{J(t-1)} < 1, \quad \mu(t) = r_{i} \mu(t-1) \]

\[ \frac{J(t)}{J(t-1)} > c, \quad \mu(t) = r_{d} \mu(t-1) \]

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

\( \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
- \textit{delta-delta} rule
- Conjugate gradient algorithm
- Newton family approaches
- Algorithms based on the Kalman filtering approach
- Levenberg–Marquardt algorithm
- \textit{Quickprop} scheme
The cross-entropy

\[ J = -\sum_{i=1}^{N} E(i) \]

\[ E(i) = \sum_{m=1}^{k_L} \left\{ y_m(i) \ln \hat{y}_m(i) + (1 - y_m(i)) \ln(1 - \hat{y}_m(i)) \right\} \]

- 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.

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 that approximate optimally class a-posteriori probabilities!!!

\[ \hat{y}_m(i) \approx P(\omega_m|x(i)) \]

That is, the probability of class \( \omega_m \) given \( x(i) \).

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

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

+ 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 the Hessian matrix is diagonal

\[ \delta J \approx \frac{1}{2} \sum_i h_{ii} \delta w_i^2 \]
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

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

- 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 favours small values for the weights, e.g.,

$$E_p(\omega) = \sum_{k=1}^{K} h(w_k^2) \quad h(\cdot) \text{ is an appropriately chosen differentiable function.}$$

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

where $w_0 \approx 1$.
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. This leads to the so called overfitting.
Example: NN: (2-3-2-1); Logistic Function with \( a=1 \). (a) The momentum \( \mu=0.05 \), \( \alpha=0.85 \) and (b) The adaptive momentum \( \mu=0.01 \), \( \alpha=0.85 \), \( r_i=1.05 \), \( c=1.05 \), \( r_d=0.7 \).

\[
\begin{align*}
[0.4, 0.9]^T, [2.0, 1.8]^T, [2.3, 2.3]^T, [2.6, 1.8]^T \\
\end{align*}
\]

400 training Samples
The mean values
Variances=0.08
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

\[ x \rightarrow y = \begin{bmatrix} f(g_1(x)) \\ f(g_2(x)) \end{bmatrix} \]

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

- In the more general case:
  - Let \( x \in R^l \) and a nonlinear classification task.
  - \( f_i(.) \), \( i = 1, 2, \ldots, k \)
  
  \( f_i : R^l \rightarrow R \), \( i = 1, 2, \ldots, k \)
Are there any functions and an appropriate \( k \), so that the mapping

\[
x \rightarrow y = \begin{bmatrix} f_1(x) \\ \vdots \\ f_k(x) \end{bmatrix}
\]

transforms the task into a linear one, in the space?

If this is true, then there exists a hyperplane \( w \in \mathbb{R}^k \) so that

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

\[
\text{If } w_0 + w^T y < 0, \quad x \in \omega_2
\]
In such a case this is equivalent with approximating the nonlinear discriminant function \( g(x) \), in terms of \( f_i(x) \), i.e.,

\[
g(x) \approx w_0 + \sum_{i=1}^{k} w_i f_i(x) \quad (\text{or}) \quad 0
\]

Given \( f_i(x) \), 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 \( f_i(x) \) are interpolation functions.
- From the Pattern Recognition point of view, this is justified by Cover’s theorem.
$g(x) \equiv w_0 + \sum_{i=1}^{k} w_i f_i(x) \ (\prec) \ 0$

**Generalized Linear Classification.**

$g(x)$ corresponds to a two-layer network where the nodes of the hidden layer have different activation functions, $f_i(\cdot)$, $i=1, 2, \ldots, k$. 
Capacity of the $l$-dimensional space in Linear Dichotomies

- Assume $N$ points in $\mathbb{R}^l$ assumed to be in general position, that is:

Not $\ell + 1$ of these lie on a $\ell - 1$ 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 \(2^4=16\)
Probability of grouping $N$ points in two linearly separable classes is

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

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

$$g(x) = w_0 + \sum_{i=1}^{l} w_i x_i + \sum_{i=1}^{l-1} \sum_{m=i+1}^{l} w_{im} x_i x_m + \sum_{i=1}^{l} w_{ii} x_i^2$$
\[ \mathbf{x} = [x_1, x_2]^T \quad \mathbf{y} = [x_1, x_2, x_1x_2, x_1^2, x_2^2]^T \]

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

XOR problem

\[ \mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ x_1x_2 \end{bmatrix} \]

\[ y_1 + y_2 - 2y_3 - \frac{1}{4} = 0 \]

\[ g(\mathbf{x}) = -\frac{1}{4} + x_1 + x_2 - 2x_1x_2 \]

\[ >0 \quad \mathbf{x} \in A \]

\[ <0 \quad \mathbf{x} \in B \]
Radial Basis Function Networks (RBF)

- Choose

\[ f(\|x - c_i\|) \]

\[ f(x) = \exp\left( -\frac{1}{2\sigma_i^2} \|x - c_i\|^2 \right) \]

\[ f(x) = \frac{\sigma^2}{\sigma^2 + \|x - c_i\|^2} \]
Equivalent to a single layer network, with RBF activations and linear output node.

\[ f_i(x) = \exp \left( -\frac{\|x - c_i\|^2}{2\sigma_i^2} \right) \]

\[ g(x) = w_0 + \sum_{i=1}^{k} w_i \exp \left( -\frac{(x - c_i)^T(x - c_i)}{2\sigma_i^2} \right) \]
Example: The XOR problem

- Define:

\[ c_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad c_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \sigma_1 = \sigma_2 = \frac{1}{\sqrt{2}} \]

\[ f_i(x) = \exp\left(-\|x - c_i\|^2\right), \quad y = y(x) = \begin{bmatrix} \exp(-\|x - c_1\|^2) \\ \exp(-\|x - c_2\|^2) \end{bmatrix} \]

- \[
\begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0.135 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 0.135 \end{bmatrix},
\begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0.368 \\ 0.368 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.368 \\ 0.368 \end{bmatrix}\]
\[ g(x) = \exp(-\|x - c_1\|^2) + \exp(-\|x - c_2\|^2) - 1 = 0 \]

\[ g(y) = y_1 + y_2 - 1 = 0 \]
Training of the RBF networks

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

$$y = \left[ \exp \left( \frac{-\|x - c_1\|^2}{2\sigma_1^2} \right), \ldots, \exp \left( \frac{-\|x - c_k\|^2}{2\sigma_k^2} \right) \right]^T$$

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

Classification using RBF with distance $= -6$, radius $= 10$, and width $= 6$. 

[Diagram showing classification results]
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 $w^T x = c$ 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:

\[ x \in \mathbb{R}^l \rightarrow y \in \mathbb{R}^k, \quad k > l \]

Then use SVM in \( \mathbb{R}^k \)

Recall that in this case the dual problem formulation will be

\[
\begin{align*}
\text{maximize} \quad & \left( \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j y_i^T y_j \right) \\
\text{where} \quad & y_i \in \mathbb{R}^k
\end{align*}
\]
Also, the classifier will be

\[ g(y) = w^T y + w_0 \]

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

where \( x \rightarrow y \in R^k \)

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

Let \( \mathbf{x} = [x_1, x_2]^T \in \mathbb{R}^2 \)

Let \( \mathbf{x} \rightarrow \mathbf{y} = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \in \mathbb{R}^3 \)

Then, it is easy to show that

\[
\mathbf{y}_i^T \mathbf{y}_j = (\mathbf{x}_i^T \mathbf{x}_j)^2
\]
Mercer’s Theorem

Let \( x \rightarrow \Phi(x) \in H \)  \( H: \) Hilbert Space

Then, the inner product in \( H \) is represented as:

\[
\langle \Phi(x), \Phi(z) \rangle = \sum_r \Phi_r(x) \Phi_r(z) = K(x, z)
\]

where

\[
\int_c \int_c K(x, z) g(x) g(z) dx \, dz \geq 0
\]

for any \( g(x), x \):

\[
\int_c g^2(x) dx < +\infty \quad C: \text{Compact (finite) Subset of } \mathbb{R}^l.
\]

\( K(x, z) \) is a 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

- **Polynomial:**
  \[ K(x, z) = (x^T z + 1)^q, \quad q > 0 \]

- **Radial Basis Functions:**
  \[ K(x, z) = \exp \left( -\frac{||x - z||^2}{\sigma^2} \right) \]

- **Hyperbolic Tangent:**
  \[ K(x, z) = \tanh (\beta x^T z + \gamma) \]
  for appropriate values of \( \beta, \gamma \).
SVM Formulation

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

• Step 2: 
  \[
  \max_{\lambda} \left( \sum_{i} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j K(x_i, x_j) \right)
  \]
  subject to: \( 0 \leq \lambda_i \leq C, \quad i = 1, 2, \ldots, N \)
  \[
  \sum_{i} \lambda_i y_i = 0
  \]

This results to an implicit combination

\[
\overline{w} = \sum_{i=1}^{N_s} \lambda_i y_i \varphi(x_i)
\]
• Step 3: Assign $x$ to

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

• 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.
Classification using SVM with distance = -6.5, radius = 10, and width = 6

(a) Training result
Classification using SVM with distance = $-6.5$, radius = 10, and width = 6

(a) Testing result
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:

  \[ x_i \leq \alpha \]

  where \( \alpha \) 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

$$x_i \leq \alpha$$

➤ The choice of the specific $x_i$ and the value of the threshold $\alpha$, 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 $\omega_i$. The decrease in node impurity is defined as:

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

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


• 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 $\omega_j$, 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 \( \alpha_{kn_0} \) 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 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:

$$\text{Is } \sum_{k=1}^{l} c_k x_k \leq \alpha ?$$
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 $\omega_1$, four to class $\omega_2$, and two to class $\omega_3$, in a three-class classification task. The node splitting results into two new subsets $X_{tY}$, with three vectors from $\omega_1$, and one from $\omega_2$, and $X_{tN}$ with one vector from $\omega_1$, three from $\omega_2$, and two from $\omega_3$. 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

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

$$P(\omega_i \mid x), i = 1, 2, \ldots, M$$
**Product Rule:** Assign $x$ to the class $\omega_i$:

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

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

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

**Sum Rule:** Assign $x$ to the class $\omega_i$:

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

(Using the alternative KL distance formulation.)

- 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 $x$ to the class for which there is a consensus or when at least $l_c$ of the classifiers agree on the class label of $x$ where:

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

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 \( \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \)
with \( y_i \in \{-1,1\}, i = 1, 2, \ldots, N \).

Construct an optimally designed classifier of the form:
\[
    f(x) = \text{sign}\{F(x)\}
\]
where:
\[
    F(x) = \sum_{k=1}^{K} \alpha_k \phi(x; \theta_k)
\]
where \( \phi(x; \theta_k) \) denotes the base classifier that returns a binary class label:
\[
    \phi(x; \theta_k) \in \{-1, 1\}
\]
\( \theta \) is a parameter vector.
The essence of the method.

Design the series of classifiers:

\[ \phi(x; \theta_1), \phi(x; \theta_2), \ldots, \phi(x; \theta_k) \]

The parameter vectors

\[ \theta_k, k = 1, 2, \ldots, K \]

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 (highly complex):

\[ \arg \min_{\alpha_k; \theta_k, k: 1, K} \sum_{i=1}^{N} \exp(-y_i F(x_i)) \]

It penalizes the samples that are wrongly classified much more heavily than those correctly classified.
Updating the weights for each sample \( x_i, i = 1, 2, ..., N \)

\[
W_{i}^{(m+1)} = \frac{w_{i}^{(m)} \exp(-y_{i} \alpha_{m} \phi(x_{i}; \theta_{m}))}{Z_{m}}
\]

- \( Z_{m} \) is a normalizing factor common for all samples.

- \( \alpha_{m} = \frac{1}{2} \ln \frac{1-P_{m}}{P_{m}} \), \( Z_{m} = \sum_{i=1}^{N} w_{i}^{(m)} \exp(-y_{i} \alpha_{m} \phi(x_{i}; \theta_{m})) \)

where \( P_{m} < 0.5 \) (by assumption) is the error rate of the optimal classifier \( \phi(x; \theta_{m}) \) at stage \( m \). Thus \( \alpha_{m} > 0 \).

- The term: \( \exp(-y_{i} \alpha_{m} \phi(x_{i}; \theta_{m})) \)

  takes a large value if \( y_{i} \phi(x_{i}; \theta_{m}) < 0 \) (wrong classification) and a small value in the case of correct classification \( \{ y_{i} \phi(x_{i}; \theta_{m}) > 0 \} \)

- 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: \( w^{(1)}_i = \frac{1}{N}, \ i = 1, 2, \ldots, N \)

• Initialize: \( m = 1 \)

• Repeat
  
  – Compute optimum \( \theta_m \) 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 \( N \)
    
    * \( w^{(m+1)}_i = w^{(m)}_i \exp (-y_i \alpha_m \phi(x_i; \theta_m)) \)
    
    * \( Z_m = Z_m + w^{(m+1)}_i \)
  
  – End{}\{For\}
  
  – For \( i = 1 \) to \( N \)
    
    * \( w^{(m+1)}_i = (w^{(m+1)}_i)/Z_m \)
  
  – 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.

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