# Artificial Neural Networks


## Prof. Johan Suykens

Katholieke Universiteit Leuven
Department of Electrical Engineering, ESAT-STADIUS
Kasteelpark Arenberg 10
B-3001 Leuven (Heverlee), Belgium
Tel: 32/16/32 18 02 Fax: 32/16/32 19 70
E-mail: johan.suykens@esat.kuleuven.be
http://www.esat.kuleuven.be/stadius


Academic year 2013-2014

These course notes relate to the following lectures:

- Lecture 2: Multilayer feedforward networks and backpropagation
- Lecture 3: Training of feedforward neural networks
- Lecture 4: Generalization
- Lecture 6 (partially): Recurrent neural networks
- Lecture 7 (partially): Unsupervised learning
- Lecture 8 (partially): Nonlinear modelling and control
- Lecture 9: Support vector machines

Additional course material and optional background material is available on the Toledo website for the course.

# Contents

# Chapter 1

# Multilayer feedforward networks and backpropagation

## 1.1 Multilayer perceptrons and radial basis function networks

### 1.1.1 Biological neurons and McCulloch-Pitts model

One estimates that the human brain contains over $10^{11}$ neurons and $10^{14}$ synapses in the human nervous system. On the other hand the neuron's switching time is much slower than for transistors in computers, but the connectivity is higher than in today's supercomputers. Biological neurons basically consist of three main parts: the neuron cell body, branching extensions called dendrites for receiving input and axons that carry the neuron's output to the dendrites of other neurons (Fig.1.1) [32].

A simple and popular model for neurons is the McCulloch-Pitts model (Fig.1.2). However, one should be aware that this is a strong mathematical abstraction of reality. The neuron is modelled in this case as a simple static nonlinear element which takes a weighted sum of incoming signals $x_i$ multiplied with interconnection weights $w_i$. After adding a bias term $b$ (or threshold) the resulting activation $a = \sum_i w_i x_i + b$ is sent through a static nonlinearity $f(\cdot)$ (activation

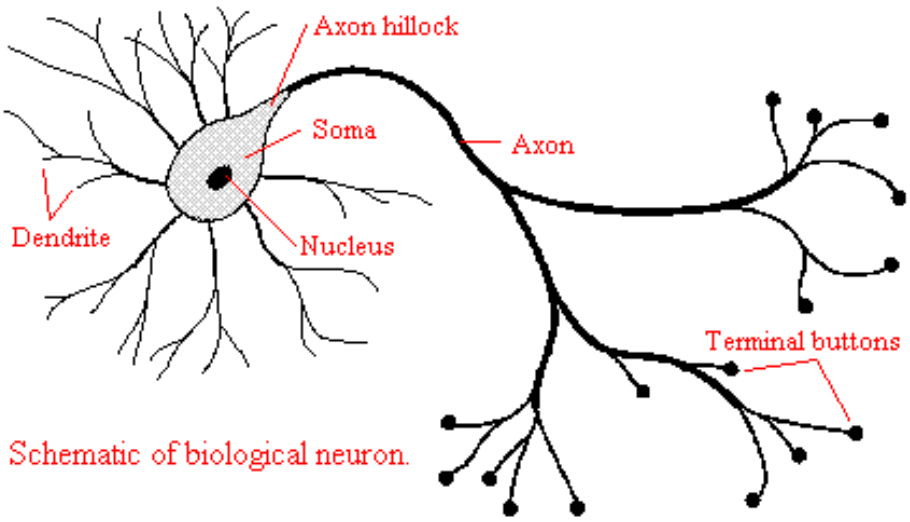



Figure 1.1: *Biological neurons.*

function) yielding the output $y$ such that

$$y = f(\sum_i w_i x_i + b). \tag{1.1}$$

The nonlinearity is typically of the saturation type, e.g. $\tanh(\cdot)$. Biologically this corresponds to the firing of a neuron depending on gathered information of incoming signals that exceeds a certain threshold value.

### 1.1.2 Multilayer perceptrons

A single neuron model is not very powerful as one could expect. However, if one organizes the neurons into a layered network with several layers one obtains models which are able to approximate general continuous nonlinear functions. Such a network consists of one or more hidden layers and an output layer. A multilayer perceptron (MLP) with one hidden layer is shown in Fig.1.3. Mathematically it is described as follows. In matrix-vector notation one has

$$y = W\,\sigma(Vx + \beta) \tag{1.2}$$

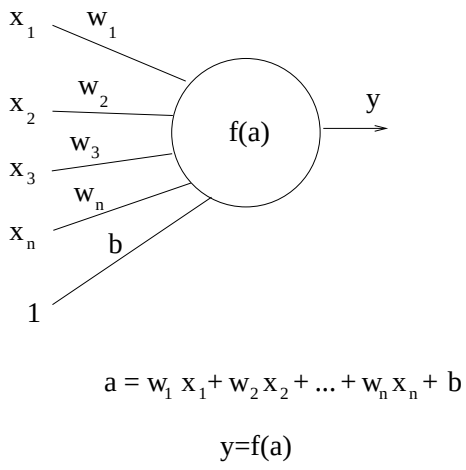



Figure 1.2: *McCulloch-Pitts model of neuron.*

with input $x \in \mathbb{R}^m$, output $y \in \mathbb{R}^l$ and interconnection matrices $W \in \mathbb{R}^{l \times n_h}$, $V \in \mathbb{R}^{n_h \times m}$ for the output layer and hidden layer, respectively. The bias vector is $\beta \in \mathbb{R}^{n_h}$ and consists of the threshold values of the $n_h$ hidden neurons. This notation is more compact than the elementwise notation:

$$y_i = \sum_{r=1}^{n_h} w_{ir}\, \sigma(\sum_{j=1}^{m} v_{rj} x_j + \beta_r), \qquad i = 1, ..., l. \tag{1.3}$$

In these descriptions a linear activation function is taken for the output layer. Depending on the application one might choose other functions as well (Fig.1.4). For problems of nonlinear function estimation and regression one takes a linear activation function in the output layer.

Sometimes a neural network with two hidden layers (Fig.1.5) is chosen although a single hidden layer is sufficient to have a universal approximator (provided a sufficient number of hidden units is taken). In matrix-vector notation one has:

$$y = W\, \sigma(V_2\, \sigma(V_1 x + \beta_1) + \beta_2) \tag{1.4}$$

with input $x \in \mathbb{R}^m$, output $y \in \mathbb{R}^l$ and interconnection matrices $W \in \mathbb{R}^{l \times n_{h_2}}$, $V_2 \in \mathbb{R}^{n_{h_2} \times n_{h_1}}$, $V_1 \in \mathbb{R}^{n_{h_1} \times m}$ and bias vectors $\beta_2 \in \mathbb{R}^{n_{h_2}}$, $\beta_1 \in \mathbb{R}^{n_{h_1}}$ with number of neurons in the hidden layers $n_{h_1}$ and $n_{h_2}$.

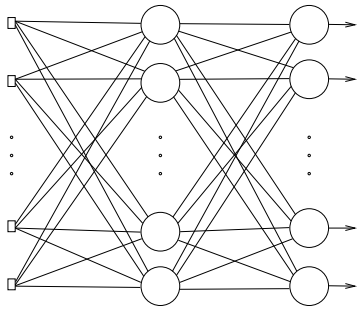


Figure 1.3: *Multilayer perceptron with one hidden layer.*

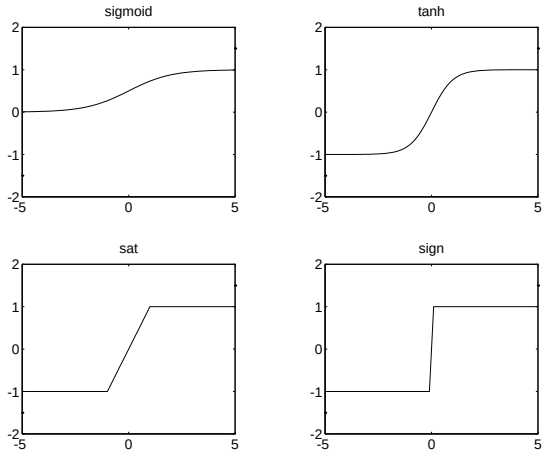



Figure 1.4: *Some typical activation functions.*

In elementwise notation this becomes:

$$y_i = \sum_{r=1}^{n_{h_2}} w_{ir}\, \sigma(\sum_{s=1}^{n_{h_1}} v_{rs}^{(2)}\, \sigma(\sum_{j=1}^{m} v_{sj}^{(1)} x_j + \beta_s^{(1)}) + \beta_r^{(2)}), \qquad i = 1, ..., l \tag{1.5}$$

where the upper indices indicate the layer numbers. Sometimes the inputs are considered to be part of a so-called input layer. However, in order to specify the number of layers of a network and avoid confusion it is prefered to mention the number of hidden layers and define the number of layers to be the sum of the number of hidden layers plus the output layer.

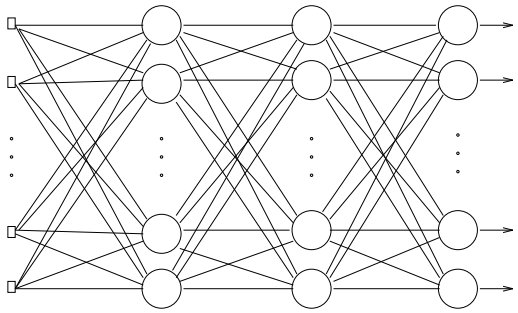


Figure 1.5: *Multilayer perceptron with two hidden layers.*

### 1.1.3 Radial basis function networks

While MLPs contain saturation type nonlinearities, another important class of neural networks called Radial Basis Function networks (RBF) makes use of localized basis functions, typically with Gaussian activation functions organized within one hidden layer (Fig.1.6).

The network description is

$$y = \sum_{i=1}^{n_h} w_i \, h(\|x - c_i\|). \tag{1.6}$$

For a Gaussian activation function this becomes

$$y = \sum_{i=1}^{n_h} w_i \exp(-\|x - c_i\|_2^2/\sigma_i^2)$$

with input $x \in \mathbb{R}^m$, output $y \in \mathbb{R}$, output weights $w \in \mathbb{R}^{n_h}$, centers $c_i \in \mathbb{R}^m$ and widths $\sigma_i \in \mathbb{R}$ $(i = 1, ..., n_h)$ where $n_h$ denotes the number of hidden neurons.

## 1.2 Universal approximation theorems

### 1.2.1 Neural nets are universal approximators

In fact the history on artificial neural networks dates back from the beginning of the previous century around 1900, when Hilbert formulated a list of 23 challenging mathematical problems for the century to come. In his famous 13th problem he formulated the conjecture

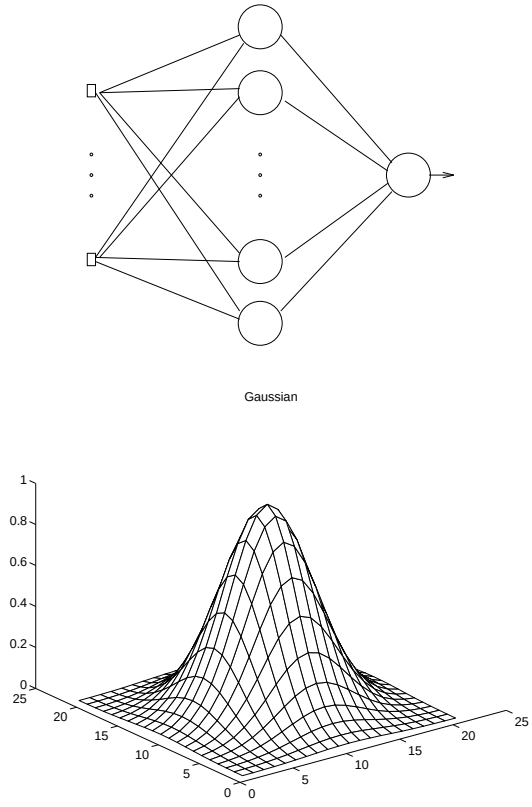



Figure 1.6: *Radial basis function network with Gaussian activation function.*

that there are analytical functions of three variables which cannot be represented as a finite superposition of continuous functions of only two variables. This conjecture was refuted by Kolmogorov and Arnold in 1957.

**Kolmogorov Theorem (1957)** Any continuous function $f(x_1, ..., x_n)$ defined on $[0,1]^n$ $(n \geq 2)$ can be represented in the form

$$f(x) = \sum_{j=1}^{2n+1} \chi_j (\sum_{i=1}^{n} \phi_{ij}(x_i))$$

where $\chi_j, \phi_{ij}$ are continuous functions and $\phi_{ij}$ are also monotone.

This theorem was later refined as follows.

**Sprecher Theorem (1965)** There exists a real monotone increasing function $\phi(x) : [0,1] \rightarrow [0,1]$ depending on $n$ $(n \geq 2)$ having the following property: $\forall \delta > 0, \exists \epsilon$ $(0 < \epsilon < \delta)$ such that every real continuous function $f(x) : [0,1]^n \rightarrow \mathbb{R}$ can be represented as

$$f(x) = \sum_{j=1}^{2n+1} \chi [\sum_{i=1}^{n} \lambda^i \phi(x_i + \epsilon(j-1)) + j - 1]$$

where $\chi$ is a real continuous function and $\lambda$ is a constant.

A link between the Sprecher Theorem and MLP neural networks was made by Hecht-Nielsen.

**Hecht-Nielsen Theorem (1987)** Any continuous mapping $f(x) : [0,1]^n \rightarrow \mathbb{R}^m$ can be represented by a neural net with two hidden layers.

The proof of this Theorem relies on the Sprecher Theorem. Later in 1989 new Theorems were proven which showed that for universal approximation it is sufficient to have MLPs with one hidden layer.

**Hornik Theorem (1989)** [31] Regardless of the activation function (can be discontinuous), the dimension of the input space, a neural

network with one hidden layer can approximate *any* continuous function arbitrarily well (in a certain metric).

These results also hold for networks with multiple outputs. The following Theorem is more specific about the kind of activation functions that are allowed.

**Leshno Theorem (1993)** A standard multilayer feedforward NN with locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy *if and only if* the network's activation function is not a polynomial.

In addition to these universal approximation theorems for MLP neural networks, Theorems have been proven for RBF networks as well, e.g. Park and Sandberg in 1991 showed that it is possible to approximate any continuous nonlinear function by means of an RBF network.

These results mathematically guarantee us that when we parameterize nonlinear functions and nonlinear model structures we obtain universal tools for nonlinear modelling. However, these universal approximation results are *not* constructive in the following sense. First, given a nonlinear function to be approximated, the proof does not provide us with an algorithm for determination of the interconnection weights of the neural nets. Secondly, it is not clear how many hidden neurons are sufficient to approximate a given nonlinear function on a compact interval. These issues will be discussed in the sequel of this course.

### 1.2.2 The curse of dimensionality

Universal approximation for neural networks is a nice property. However, one could argue that also polynomial expansions possess this property. So is there really a reason why we should use neural nets instead of these? The answer is yes. The reason is that neural networks are better able to cope with the curse of dimensionality. Barron [21] has shown in 1993 that neural networks can avoid the curse of dimensionality in the sense that the approximation error becomes independent from the dimension of the input space (under certain conditions), which is not the case for polynomial expansions. The approximation error for MLPs with one hidden layer is order of magnitude $\mathcal{O}(1/n_h)$,

but $\mathcal{O}(1/n_p^{2/n})$ for polynomial expansions where $n_h$ denotes the number of hidden units, $n$ the dimension of the input space and $n_p$ the number of terms in the expansion. Consider for example $y = f(x_1, x_2, x_3)$. A polynomial expansion with terms up to degree 7 would contain many many terms

$$\begin{aligned} y \;=\; & a_1x_1 + a_2x_2 + a_3x_3 + a_{11}x_1^2 + a_{22}x_2^2 + a_{33}x_3^2 + \\ & a_{12}x_1x_2 + a_{13}x_1x_3 + a_{23}x_2x_3 + a_{111}x_1^3 + a_{222}x_2^3 + \\ & a_{333}x_3^3 + ... + a_{1111111}x_1^7 + a_{2222222}x_2^7 + .... \end{aligned}$$

This means that for a given number of training data, the number of parameters to be estimated is huge (which should be avoided as we will discuss later in the Chapter on learning and generalization). For MLPs

$$y = w^T \tanh(V \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \beta)$$

the number of interconnection weights grows less dramatically when the dimension of the input space grows.

## 1.3 Backpropagation training

### 1.3.1 Generalized delta rule

The first algorithm invented for the training of multilayer perceptrons (and multilayer feedforward networks in general) is the *backpropagation* method [43]. The invention of this supervised learning method caused a worldwide boom in neural networks research and applications within a large variety of areas. We present the method here in its general form.

Consider a multilayer feedforward neural network with $L$ layers ($L-1$ hidden layers) (index $l = 1, ..., L$), $P$ input patterns and corresponding desired output patterns (index $p = 1, ..., P$) and $N_l$ neurons in layer $l$. For the network description we have the following relation between layer $l$ and layer $l+1$ (Fig.1.7):

$$x_{i,p}^{l+1} = \sigma(\xi_{i,p}^{l+1}), \qquad \xi_{i,p}^{l+1} = \sum_{j=1}^{N_l} w_{ij}^{l+1} x_{j,p}^{l} \tag{1.7}$$

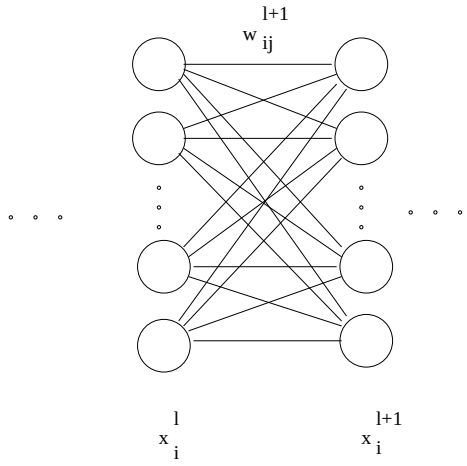



Figure 1.7: *One layer from a feedforward network consisting of a number of L layers.*

where the upper index is the layer index and the lower indices indicate the neuron within a layer and the pattern index. $x_{i,p}^l$ denotes the $i$-th component of the output vector of layer number $l$ for pattern $p$ and $w_{ij}^l$ the $ij$-th entry of the interconnection matrix of layer $l$. Eventually, the activation function $\sigma(\cdot)$ might also change from layer to layer. Before we are in a position to formulate the backpropagation algorithm we first have to define so-called $\delta$ variables

$$\delta_{i,p}^l = \frac{\partial E_p}{\partial \xi_{i,p}^l} \tag{1.8}$$

where $E_p = \frac{1}{2}\sum_{i=1}^{N_L}(x_{i,p}^{desired} - x_{i,p}^L)^2$ is the error for pattern $p$ and $x_{i,p}^{desired}$ denotes the desired output (note that this method is supervised).

The objective function (sometimes called energy function) that one usually optimizes for the neural network is the mean squared error (MSE) on the training set of patterns:

$$\min_{w_{ij}^l} E = \frac{1}{P}\sum_{p=1}^{P} E_p \qquad E_p = \frac{1}{2}\sum_{i=1}^{N_L}(x_{i,p}^{desired} - x_{i,p}^L)^2. \tag{1.9}$$

The backpropagation algorithm (or generalized delta rule) is given

then by the following rule:

$$\begin{cases} \Delta w_{ij}^{l} &= \eta\, \delta_{i,p}^{l}\, x_{j,p}^{l-1} = -\eta\, \frac{\partial E_p}{\partial w_{ij}^{l}} \\ \delta_{i,p}^{L} &= (x_{i,p}^{desired} - x_{i,p}^{L})\, \sigma'(\xi_{i,p}^{L}) \\ \delta_{i,p}^{l} &= (\sum_{r=1}^{N_{l+1}} \delta_{r,p}^{l+1} w_{ri}^{l+1})\, \sigma'(\xi_{i,p}^{l}), \qquad l = 1, ..., L-1 \end{cases} \tag{1.10}$$

with learning rate $\eta$. Note that the last equation is a backward recursive relation on the $\delta_{i,p}^{l}$ variable in the layer index $l$. The backpropagation algorithm is an elegant method in order to obtain analytic expressions for the gradient of the cost function defined on a feedforward network with many layers. One could imagine that obtaining expressions for the gradient in the case of one hidden layer is straightforward. However, suppose one has a network with e.g. 100 layers, it is clear then that obtaining an expression for the gradient becomes far from trivial, while by applying this generalized delta rule it becomes straightforward. The special structure appearing in this generalized delta rule is due to the layered structure of the network. In order to fix the ideas, an application of the generalized delta rule is shown for an MLP with one hidden layer ($L = 2$) in Fig.1.8. In this case the equations become

$$\begin{cases} \Delta w_{ij}^{2} &= \eta\, \delta_{i,p}^{2}\, x_{j,p}^{1} \\ \Delta w_{ij}^{1} &= \eta\, \delta_{i,p}^{1}\, x_{j,p}^{0} \\ \delta_{i,p}^{2} &= (x_{i,p}^{desired} - x_{i,p}^{2})\, \sigma'(\xi_{i,p}^{2}) \\ \delta_{i,p}^{1} &= (\sum_{r=1}^{N_2} \delta_{r,p}^{2} w_{ri}^{2})\, \sigma'(\xi_{i,p}^{1}). \end{cases} \tag{1.11}$$

This backpropagation method can be used either off-line (batch mode) or on-line. Off-line means that one first presents all training patterns before updating the weights after calculating $\Delta w_{ij}^{l}$. Presenting all the training data before doing this update is called one epoch. This corresponds in fact to one iteration step if one takes an optimization theory point of view. On-line updating means that one updates $\Delta w_{ij}^{l}$ each time a new pattern of the training data set is presented. The latter opens the possibility for its application to adaptive signal processing (in fact one can also show that backpropagation is an extension of the well-known LMS algorithm towards layered nonlinear

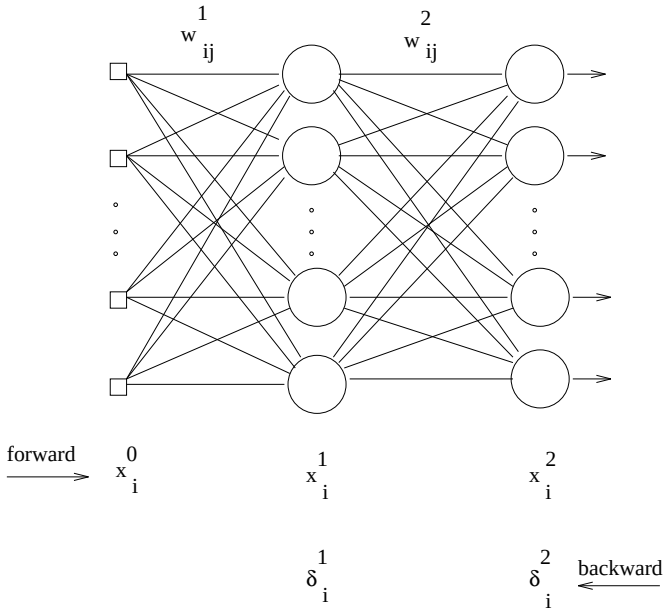



Figure 1.8: *Illustration of backpropagation for an MLP with one hidden layer.*

models). More advanced on-line learning algorithms have also been developed which are based on extended Kalman filtering. Often a momentum term is used in the backpropagation in order to speed up the method. One adapts the interconnection weights then according to

$$\Delta w_{ij}^{l}(k+1) = \eta\, \delta_{i,p}^{l}\, x_{j,p}^{l-1} + \alpha\, \Delta w_{ij}^{l}(k) \tag{1.12}$$

where $k$ is the iteration step and $0 < \alpha < 1$. Often also an adaptive learning rate $\eta$ is taken. The previous change in the interconnection weights is taken into account in this learning rule.

## 1.4 Single neuron case: perceptron algorithm

The perceptron algorithm was one of the early methods in the area of neural networks. Especially in the sixties this algorithm has been investigated. A perceptron consists of a single neuron with a sign acti-

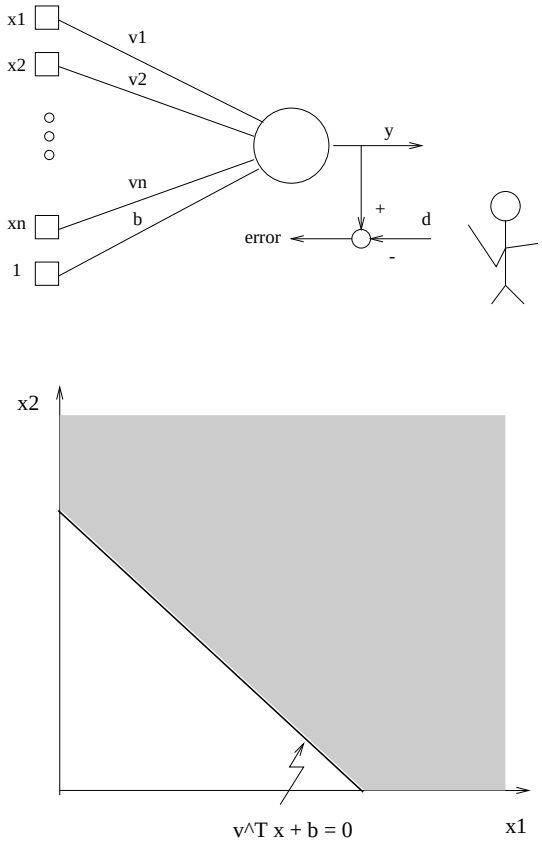



Figure 1.9: *Training of a perceptron.*

vation function (hardlimiter). Given a set of features $x$ the perceptron is able to realize a linear classification into two halfspaces.

The perceptron network (Fig.1.9):

$$\begin{aligned} y &= \text{sign}(v^T x + b) \\ &= \text{sign}(w^T z) \end{aligned} \tag{1.13}$$

is trained in a supervised way for a given training set $\{x^{(i)}, d^{(i)}\}_{i=1}^{N}$. The following notation is used here: augmented input vector $z = [x; 1] \in \mathbb{R}^{n+1}$, $w = [v; b]$, input $x \in \mathbb{R}^n$, output $y \in \mathbb{R}$ and desired outputs $d$.

*Perceptron algorithm*

1. Choose $c > 0$

2. Initialize small random weights $w$. Set $k = 1, i = 1$ and set cost function $E = 0$.

3. The training cycle begins here. Present the $i$-th input and compute the corresponding output

$$y^{(i)} = \text{sign}(w^T z^{(i)}).$$

4. Update the weights

$$w := w + \frac{1}{2} c \left[ d^{(i)} - y^{(i)} \right] z^{(i)}.$$

5. Compute the cycle error

$$E := E + \frac{1}{2} [d^{(i)} - y^{(i)}]^2.$$

6. If $i < N$, then $i := i + 1$ and $k := k + 1$ and go to step 3. Otherwise, go to step 7.

7. If $E = 0$ then stop the training. If $E > 0$ then set $E = 0, i = 1$ and enter a new training cycle by going to step 3.

Note that the weights $w$ are updated by taking into account the errors $[d^{(i)} - y^{(i)}]$ which is also the case in the backpropagation algorithm where the errors are backpropagated by means of the $\delta$ variables. A proof of convergence exists for this perceptron algorithm.

## 1.5 Linear versus non-linear separability

The use of a perceptron model has serious limitations. A simple example of the well-known XOR problem shows that the perceptron is unable to find a decision line which correctly separates the two classes for this problem (Fig.1.10). Indeed the perceptron can only realize a hyperplane in the input space of given features (or straight line in the case of two inputs). Because of this reason there was few research in the area of neural networks in the seventies until the introduction of multilayer perceptrons with the backpropagation algorithm. By adding a hidden layer (Fig.1.11) the XOR problem can be solved because nonlinear decision boundaries can be realized. By means of a

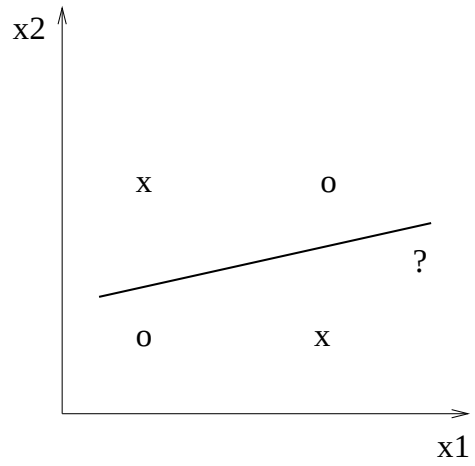



Figure 1.10: *Exclusive-OR (XOR) problem.*

hidden layer one can realize convex regions, and furthermore by means of two hidden layers non-connected and non-convex regions can be realized. The universal approximation ability of neural networks makes it also a powerful tool in order to solve classification problems.

An interesting theorem about separability is Cover's Theorem. It considers the case of continuous input variables in a $d$-dimensional space and addresses the question of what is the probability that a random set of patterns that is generated in this input space is linearly separable. One assigns randomly points to classes $\mathcal{C}_1, \mathcal{C}_2$ with equal probability where each possible assignment for the complete data set is called a dichotomy. For $N$ points there are $2^N$ possible dichotomies. The fraction $F(N,d)$ of these dichotomies which is linearly separable is given by:

$$F(N,d) = \begin{cases} 1 & , \quad N \leq d+1 \\ \frac{1}{2^{N-1}} \sum_{i=0}^{d} \begin{pmatrix} N-1 \\ i \end{pmatrix} & , \quad N \geq d+1 \end{cases} \tag{1.14}$$

with $\begin{pmatrix} N \\ M \end{pmatrix} = \frac{N!}{(N-M)!M!}$ the number of combinations of $M$ objects selected from a total of $N$. Some important conclusions at this point are that if $N < d+1$ any labelling of points will always lead to linear separability and for larger $d$ it becomes likely that more dichotomies are linearly separable. If one takes e.g. $N = 4$ and $d = 2$ one of the

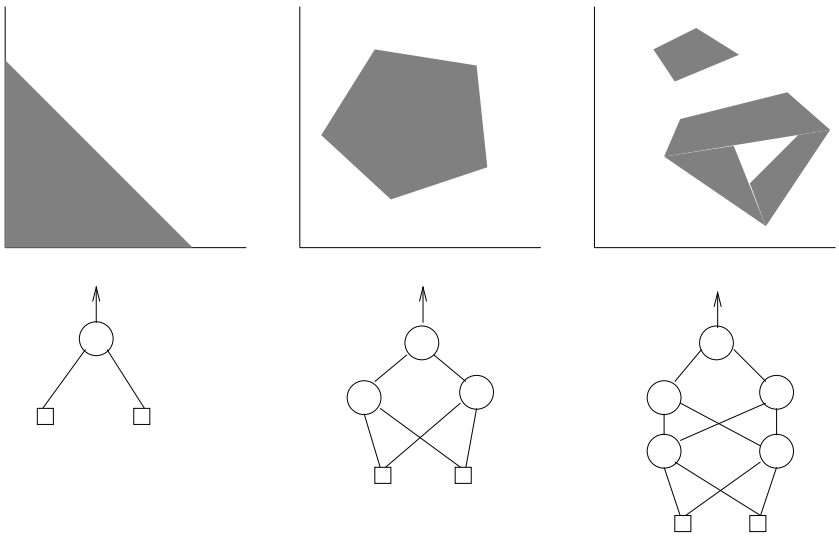


Figure 1.11: *From perceptron to multilayer perceptron (MLP) classifiers.*

possible dichotomies corresponds to the XOR problem configuration. According to Fig.1.12 one can indeed see that not all problems are linearly separable. However, if one considers the same problem e.g. for $d = 5$ instead of $d = 2$ the problem becomes linearly separable. The problem of how to select a good decision boundary will be discussed later. Cover's theorem should rather be considered as a theorem about existence of hyperplanes and not about whether this hyperplane is good or bad in terms of generalization.

## 1.6 Multilayer perceptron classifiers

A popular approach for applying MLPs to classification problems is to consider it as a regression problem (which has been explained in the previous Chapter) with the class labels as target outputs.

Consider an MLP with input vector $x \in \mathbb{R}^m$ and output $y \in \mathbb{R}^l$:

$$y = W \tanh(Vx + \beta). \tag{1.15}$$

The output values $\{-1, +1\}$ (or $\{0, 1\}$) denote then the two classes. In the case of multiple outputs, one can encode $2^l$ classes by means of

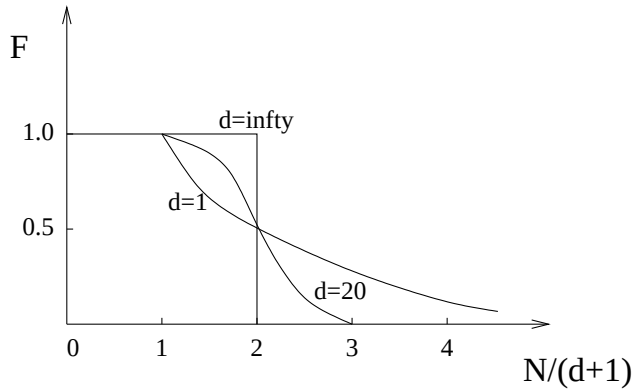



Figure 1.12: *Fraction $F(N, d)$ of dichotomies which is linearly separable according to Cover's Theorem, where $N$ is the number of points and $d$ is the dimension of the input space.*

$l$ outputs, e.g. for $l = 2$ one has the following option

| | $y_1$ | $y_2$ |
|---|---|---|
| Class 1 | $+1$ | $+1$ |
| Class 2 | $+1$ | $-1$ |
| Class 3 | $-1$ | $+1$ |
| Class 4 | $-1$ | $-1$ |

but on the other hand one can also utilize one output per class:

| | $y_1$ | $y_2$ | $y_3$ | $y_4$ |
|---|---|---|---|---|
| Class 1 | $\boxed{+1}$ | $-1$ | $-1$ | $-1$ |
| Class 2 | $-1$ | $\boxed{+1}$ | $-1$ | $-1$ |
| Class 3 | $-1$ | $-1$ | $\boxed{+1}$ | $-1$ |
| Class 4 | $-1$ | $-1$ | $-1$ | $\boxed{+1}$ |

which is better from the viewpoint of information theory. For example in trying to recognize the letters of the alphabet by an MLP one can take 26 outputs (Fig.1.13). Training can be done then in the same way as for regression with the class labels as target values for the outputs. After training of the classifier, decisions are made by the classifier as follows:

$$y = \text{sign}[W \tanh(Vx + \beta)] \tag{1.16}$$

where the sign function is a hardlimiter for obtaining a binary decision. If the output does not correspond to one of the defined class codes,

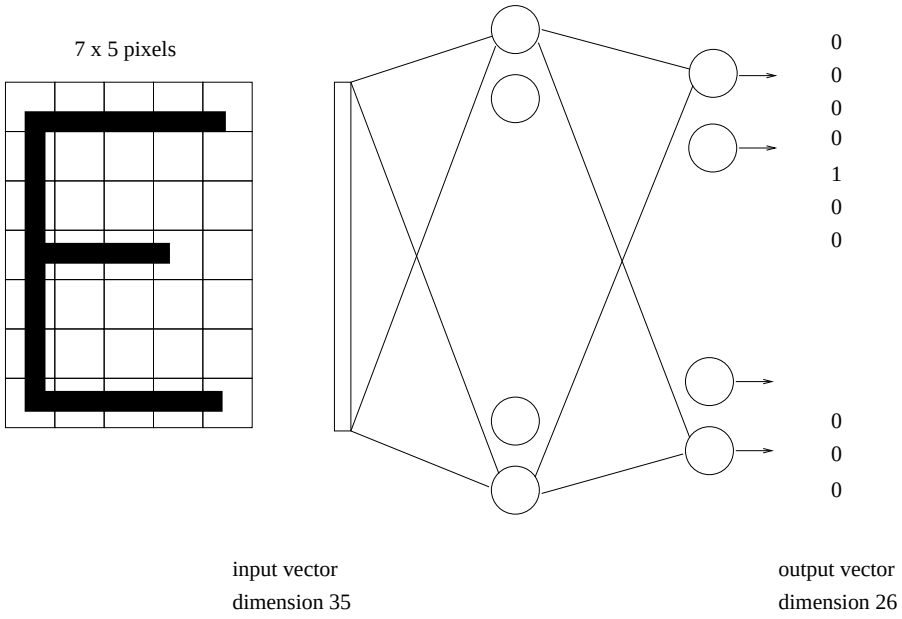



Figure 1.13: *Illustration of a regression approach of MLPs to a multi-class classification problem.*

one can assign it to the class which is closest in terms of Hamming distance.

# Chapter 2

# Training of feedforward neural networks

## 2.1 Learning and optimization

### 2.1.1 From steepest descent to Newton method

When we consider the case of off-line learning the minimization of the objective function can be studied from the viewpoint of optimization theory, where many efficient and reliable methods have been developed. For the sake of notational convenience, let us denote the problem

$$\min_{w_{ij}^l} E = \frac{1}{P}\sum_{p=1}^{P} E_p \qquad E_p = \frac{1}{2}\sum_{i=1}^{N_L}(x_{i,p}^{desired} - x_{i,p}^{L})^2$$

as the unconstrained nonlinear optimization

$$\min_{x\in\mathbb{R}^n} f(x)$$

where $f, x$ correspond to the cost function and the unknown interconnection weights, respectively.

The simplest optimization algorithm is of course a steepest descent algorithm

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

where $x_k$ denotes the $k$-th iterate. In this case the search direction corresponds to minus the gradient of the cost function i.e. backprop-

agation without momentum term. However, in the area of *local* optimization algorithms more advanced methods exist which are much faster [7].

Let us consider a current point $x_0$ in the search space and consider a Taylor expansion of the cost function around the point

$$f(x) \simeq f(x_0) + g^T \Delta x + \frac{1}{2} \Delta x^T H \Delta x. \tag{2.1}$$

Then $\Delta x = x - x_0$ denotes the step, $g = \nabla f(x_0)$ the gradient at $x_0$ and $H = \nabla^2 f(x_0)$ the Hessian at $x_0$. Locally, the optimal step is given by

$$\frac{\partial f}{\partial(\Delta x)} = g + H\,\Delta x = 0 \;\rightarrow\; \Delta x = -H^{-1}\, g. \tag{2.2}$$

One can see that the optimal step is determined by the gradient and the inverse of the Hessian matrix. A sequence of points in the search space $\{x_0, x_1, x_2, ...\}$ is generated by applying these search directions and calculating the optimal stepsize along these directions. It is well-known that the Newton method converges quadratically which is much faster than the steepest descent algorithm.

Unfortunately, one encounters a number of problems when trying to apply the Newton method to the training of neural networks. A first problem is that it often occurs that the Hessian has zero eigenvalues which means that one cannot take the inverse of the matrix. A second problem is that computing the second order derivatives analytically for neural networks is very complicated. We have seen that even the calculation of the gradient by means of the backpropagation method is not that simple. One can overcome these two problems by considering Levenberg-Marquardt and quasi-Newton methods.

One also has to be aware of the fact that there are very many local minima solutions when training neural networks. MLP architectures contain many symmetries (sign flip symmetries of the weights and permutation of the number of hidden units). In general, for $n_h$ number of hidden units one has $n_h! 2^{n_h}$ weight symmetries that lead to the same input/output mapping for the networks. One also starts from small random interconnection weights in order to avoid (too) bad local minima solutions.

### 2.1.2 Levenberg-Marquardt method

The Levenberg-Marquardt method is obtained by imposing an additional constraint $\|\Delta x\|_2 = 1$ to the problem

$$\min_{\Delta x} f(x) = f(x_0) + g^T \Delta x + \frac{1}{2}\Delta x^T H \Delta x. \tag{2.3}$$

This leads to the Lagrangian function

$$\mathcal{L}(\Delta x, \lambda) = f(x_0) + g^T \Delta x + \frac{1}{2}\Delta x^T H \Delta x + \frac{1}{2}\lambda(\Delta x^T \Delta x - 1) \tag{2.4}$$

and the solution follows from $\frac{\partial \mathcal{L}}{\partial(\Delta x)} = 0$, $\frac{\partial \mathcal{L}}{\partial \lambda} = 0$ where

$$\frac{\partial \mathcal{L}}{\partial(\Delta x)} = g + H\Delta x + \lambda \Delta x = 0 \;\rightarrow\; \Delta x = -[H + \lambda I]^{-1} g. \tag{2.5}$$

Note that for $\lambda = 0$ this corresponds to the Newton method and for $\lambda \rightarrow \infty$ this becomes a steepest descent algorithm. By adding a positive definite matrix $\lambda I$ to $H$ with $\lambda$ positive, one can always invert this matrix by taking $\lambda$ sufficiently large. At every iteration step a suitable value for $\lambda$ is selected. Note that for $\lambda$ small this method will converge faster. In the case of a sum squared error cost function (as used for neural networks) the Hessian $H$ has a special structure which can be exploited. Often an approximation is taken then for $H$ based on a Jacobian matrix.

### 2.1.3 Quasi-Newton methods

In quasi-Newton methods one aims at building up an approximation for the Hessian based upon gradient information only during the iterative learning process. From $f(x) = f(x_0) + g^T \Delta x + \frac{1}{2}\Delta x^T H \Delta x$ it follows that $\nabla_x f = g + H(x - x_0)$ or

$$H d_k = y_k \tag{2.6}$$

with $d_k = x_{k+1} - x_k$ and $y_k = g_{k+1} - g_k$ in general. This means that there is a linear mapping between changes in the gradient and changes in position. Assume now that the function $f$ is nonquadratic and $B_k$ is an estimate for $H$. We have then

$$\begin{array}{lcll} B_{k+1} & = & B_k + \Delta B & \text{(Hessian update)} \\ B_{k+1} d_k & = & y_k & \text{(Quasi-Newton condition).} \end{array} \tag{2.7}$$

Consider now rank 1 and rank 2 updates of the Hessian with the goal of building up curvature information. When taking the *rank 1 update*

$$\Delta B = q\, zz^T \tag{2.8}$$

$q$ and $z$ follow from the Quasi-Newton condition $(B_k + q\, zz^T)d_k = y_k$ which gives $z = y_k - B_k d_k$, $q = \frac{1}{z_k^T d_k}$. One obtains then the symmetric rank 1 update formula

$$B_{k+1} = B_k + \frac{(y_k - B_k d_k)(y_k - B_k d_k)^T}{(y_k - B_k d_k)^T d_k} \tag{2.9}$$

which starts with $B_0 = I$ as steepest descent.

For the *rank 2 update*

$$\Delta B = q_1\, z_1 z_1^T + q_2\, z_2 z_2^T \tag{2.10}$$

the quasi-Newton condition becomes $(B_k + q_1\, z_1 z_1^T + q_2\, z_2 z_2^T)d_k = y_k$. A possible choice is then $z_1 = y_k$ , $q_1 = \frac{1}{z_1^T d_k}$, $z_2 = B_k d_k$ , $q_2 = -\frac{1}{z_2^T d_k}$. This results into the BFGS (Broyden, Fletcher, Goldfarb, Shanno) formula

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T d_k} - \frac{(B_k d_k)(B_k d_k)^T}{(B_k d_k)^T d_k}. \tag{2.11}$$

Instead of the Quasi-Newton condition $B_{k+1} d_k = y_k$ one can also consider the condition

$$d_k = R_{k+1} y_k \tag{2.12}$$

in order to avoid direct inversion of the Hessian $H$ in $\Delta x = -H^{-1}\, g$. In other words, one updates an approximation for the inverse Hessian instead of the Hessian itself. A well-known method that one can obtain then is the DFP (Davidon, Fletcher, Powell) formula

$$R_{k+1} = R_k + \frac{d_k d_k^T}{d_k^T g_k} - \frac{R_k y_k y_k^T R_k}{y_k^T R_k y_k}. \tag{2.13}$$

These modifications to the Newton method result into a superlinear speed of convergence. Unfortunately, when the neural networks contain many interconnection weights, it becomes hard to store the matrices into computer memory. Therefore, for large scale neural networks conjugate gradient methods are to be preferred [1, 38, 50].

## 2.2 Methods for large scale problems

A theory of conjugate gradient algorithms has been developed originally for solving linear systems $Ax = y$ with $A = A^T > 0$ in an iterative way. In this case it is related to the optimization of a quadratic cost function $f(x) = c + b^T x + \frac{1}{2} x^T A x$, with $x \in \mathbb{R}^n$. One has the algorithm

$$\begin{cases} p_0 &= -g_0 \\ x_{k+1} &= x_k + \alpha_k p_k, \qquad \alpha_k = \frac{g_k^T g_k}{p_k^T A p_k} \\ g_{k+1} &= g_k + \alpha_k A p_k \\ p_{k+1} &= -g_{k+1} + \beta_k p_k, \qquad \beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k} \end{cases} \tag{2.14}$$

where $p_k$ denotes the search direction at iteration $k$ and $g_k$ the gradient of the cost function. The method starts as a steepest descent algorithm.

This conjugate gradient method possesses some nice properties. One can show that the search directions are so-called conjugated with respect to the matrix $A$, i.e. the property $p_i^T A p_j = \delta_{ij}$ holds with $\delta_{ij}$ the Kronecker delta. The factors $\alpha_k$ that determine the steplength are optimal in the sense of $\frac{d}{d\alpha} f(x_k + \alpha p_k)|_{\alpha=\alpha_k} = 0$ meaning that $[g(x_{k+1})]^T p_k = 0$ (vectors $p_k$ and $g(x_{k+1})$ are orthogonal). One can show also that the algorithm converges in at most $n$ steps, depending on the condition number of the matrix $A$. This conjugate gradient algorithm will also be used for solving least squares support vector machines which will be discussed in the last Chapter.

The application of the conjugate gradient algorithm to non-quadratic smooth cost functions (which is the case for neural networks) is done as follows:

$$\begin{cases} p_0 &= -g_0 \\ x_{k+1} &= x_k + \alpha_k p_k, \ \ \alpha_k \text{ s.t. } \min f(x_k + \alpha_k p_k) \text{ (Line search)} \\ p_{k+1} &= -g_{k+1} + \beta_k p_k \end{cases} \tag{2.15}$$

with

$$\begin{aligned} \beta_k &= \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k} \qquad \text{(Fletcher} - \text{Reeves)} \\ \beta_k &= \frac{g_{k+1}^T (g_{k+1} - g_k)}{g_k^T g_k} \qquad \text{(Polak} - \text{Ribiere).} \end{aligned} \tag{2.16}$$

Note that in the equation $p_{k+1} = -g_{k+1} + \beta_k p_k$ one also has a momentum effect, but a difference with the backpropagation with momentum

term is that $\beta_k$ is now automatically adjusted during the iterative process. In these algorithms often a restart procedure is applied after $n$ steps, i.e. one resets the search direction again to $p_0 = -g_0$. Modified versions of these algorithms have been successfully applied to the optimization of neural networks [38]. The advantages of conjugate gradient methods are that they are faster than backpropagation and that no storage of matrices is required.

## 2.3 Overfitting problem

Assume that training data are generated from $h(x) = 0.5+0.4\sin(2\pi x)$ with training data $x$ generated in the interval $[0.1, 1]$ with steps of 0.1. Test data are taken in $[0.01, 1]$ with steps of 0.01 (except the points that belong to the training set). Then Gaussian noise with standard deviation 0.05 is added to the data. From the training data polynomials of degree $d$ with $d \in \{1, 2, ..., 10\}$ are estimated. For example, for the polynomial model of degree 3

$$y = a_1 x + a_2 x^2 + a_3 x^3 + b \tag{2.17}$$

an overdetermined set of linear equations is constructed from the given data $\{x_n, y_n\}_{n=1}^{10}$

$$\begin{bmatrix} x_1 & x_1^2 & x_1^3 & 1 \\ x_2 & x_2^2 & x_2^3 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{10} & x_{10}^2 & x_{10}^3 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{10} \end{bmatrix}. \tag{2.18}$$

This overdetermined system is of the form

$$\mathcal{A}\theta = \mathcal{B}, \qquad e = \mathcal{A}\theta - \mathcal{B} \tag{2.19}$$

with $\mathcal{A} \in \mathbb{R}^{p\times q}(p > q)$, $\mathcal{B} \in \mathbb{R}^p$ and unknown parameter vector $\theta \in \mathbb{R}^q$. By taking a cost function in linear least squares sense

$$\min_\theta J_{LS}(\theta) = \frac{1}{2}e^T e = \frac{1}{2}(\mathcal{A}\theta - \mathcal{B})^T(\mathcal{A}\theta - \mathcal{B}). \tag{2.20}$$

the condition for optimality

$$\frac{\partial J_{LS}}{\partial \theta} = \mathcal{A}^T\mathcal{A}\theta - \mathcal{A}^T\mathcal{B} = 0 \tag{2.21}$$

gives the solution

$$\theta_{LS} = (\mathcal{A}^T\mathcal{A})^{-1}\mathcal{A}^T\mathcal{B} = \mathcal{A}^\dagger\mathcal{B} \tag{2.22}$$

where $\mathcal{A}^\dagger$ denotes the pseudo inverse matrix. The results are shown on Fig.2.1 and Fig.2.2 for the training and test set and for different degrees of the polynomials. One can observe that for the higher order polynomial (order 7) the solution starts oscillating. Let us now modify the least squares cost function by an additional term which aims at keeping the norm of the solution vector small. This technique is called **regularization** or **ridge regression** in the context of linear systems. One solves

$$\min_\theta J_{ridge}(\theta) = J_{LS}(\theta) + \frac{1}{2}\lambda\|\theta\|_2^2\,, \quad \lambda > 0. \tag{2.23}$$

The condition for optimality is given by

$$\frac{\partial J_{ridge}}{\partial\theta} = \mathcal{A}^T\mathcal{A}\theta + \lambda\theta - \mathcal{A}^T\mathcal{B} = 0 \tag{2.24}$$

with solution

$$\theta_{ridge} = (\mathcal{A}^T\mathcal{A} + \lambda I)^{-1}\mathcal{A}^T\mathcal{B}. \tag{2.25}$$

This technique is useful when $\mathcal{A}^T\mathcal{A}$ is ill conditioned. The results of ridge regression for the order 7 polynomial model is shown on Fig.2.3 for different values of the regularization constant $\lambda$. Fig.2.4 conceptually shows the bias-variance trade-off in terms of the regularization constant $\lambda$. A large value $\lambda$ decreases the variance but leads to a larger bias. This value of $\lambda$ is chosen as a trade-off solution by minimizing the sum of the variance and the bias square contributions.

## 2.4 Regularization and early stopping

In order to obtain models with good generalization ability we discuss now methods of regularization, early stopping and cross-validation.

### 2.4.1 Regularization

Consider a training set $\{x_n, t_n\}_{n=1}^N$ where $x_n \in \mathbb{R}^m$ denotes the input data and $t_n \in \mathbb{R}$ the target (output) data. The (static) model is denoted as $y(x_n; w)$ with output $y \in \mathbb{R}$ and parameters $w$ (the notations

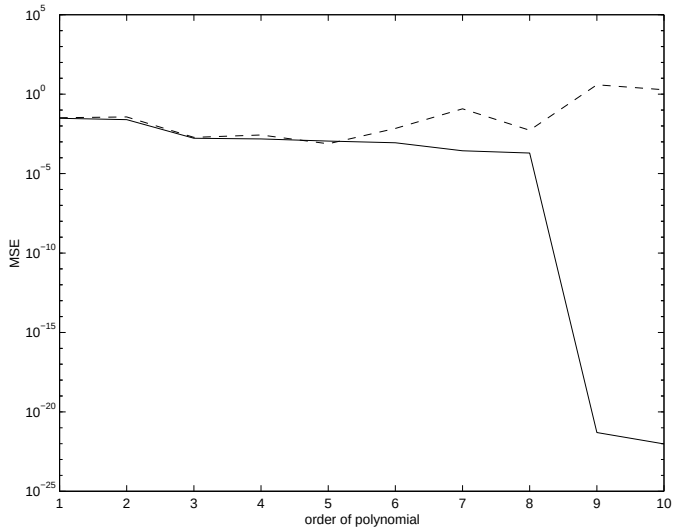



Figure 2.1: *Comparison of the polynomial models on the toy problem of estimating a noisy sinusoidal function: training set (-) and test set (- -) performance is shown as a function of the order of the polynomial.*

and explanation according to [1] is followed in this section and the next chapter).

For a given data set and model one often applies **regularization**

$$\tilde{E}(w) = E(w) + \nu\Omega(w) \tag{2.26}$$

to the original cost function $E$

$$E(w) = \frac{1}{N}\sum_{n=1}^{N}[t_n - y(x_n; w)]^2 \tag{2.27}$$

and $\nu$ a positive real regularization constant. Usually a weight decay term is taken (similar to ridge regression)

$$\Omega(w) = \frac{1}{2}\sum_{i} w_i^2 \tag{2.28}$$

where the vector $w$ contains all interconnection weights and bias terms of the neural network model. This additional term aims at keeping the interconnection weight values small.

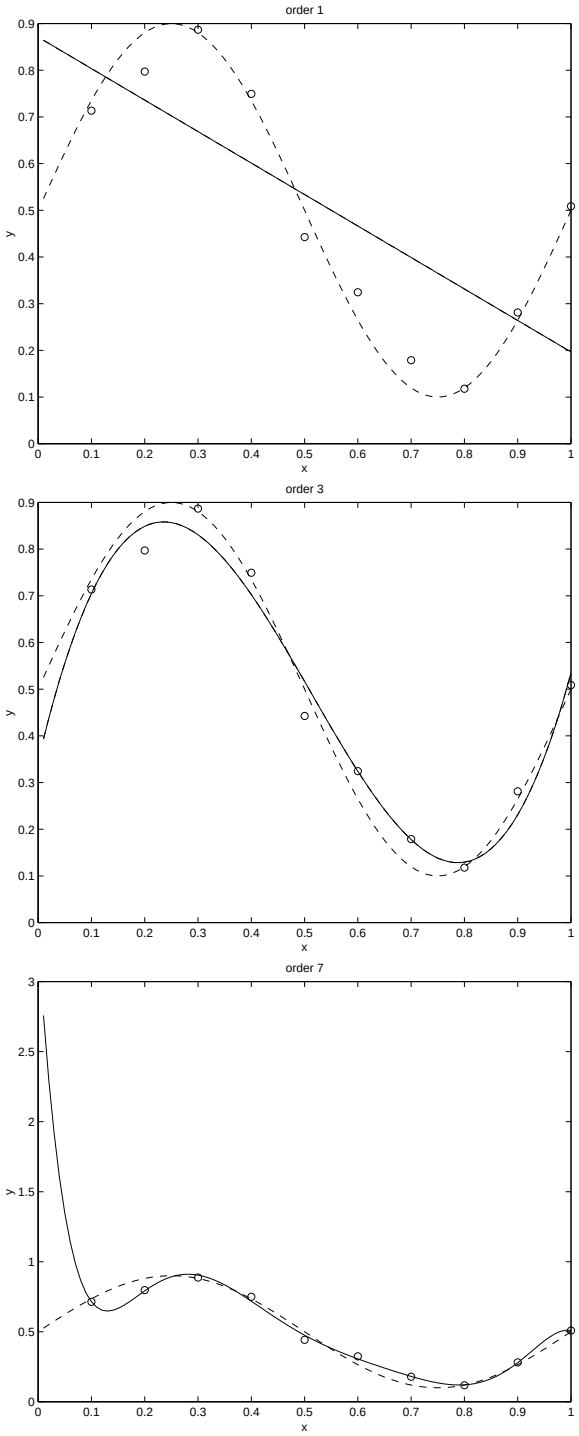



Figure 2.2: *Estimation of a noisy sinusoidal function: polynomial models of order 1,3 and 7.*

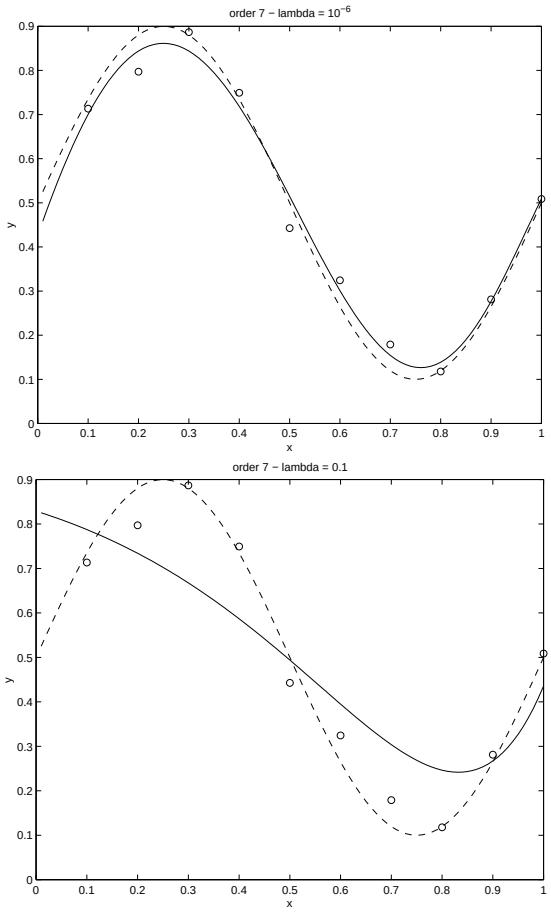



Figure 2.3: *Estimation of a noisy sinusoidal function: ridge regression applied to the polynomial model of order 7: (Top)* $\lambda = 10^{-6}$ *which avoids the oscillation; (Bottom)* $\lambda = 0.1$ *results in too much regularization.*

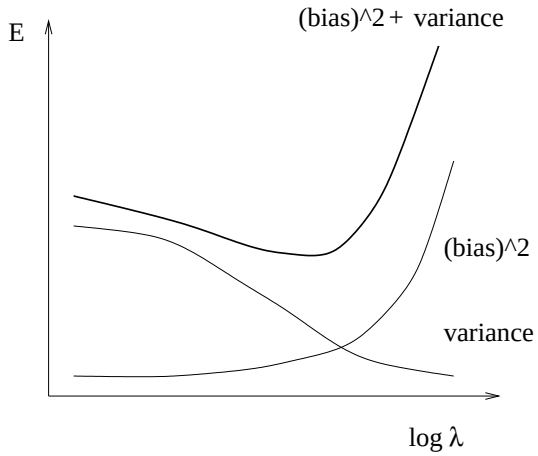



Figure 2.4: *Bias-variance trade-off when adding a regularization term to the cost function. The regularization constant $\lambda$ is chosen such that the sum of bias and variance contributions is minimized.*

Let us analyse now the influence of this weight decay term. Consider the case of a quadratic cost function which can be related to a Taylor expansion at a local point in the weight space. We have

$$E(w) = E_0 + b^T w + \frac{1}{2} w^T H w \tag{2.29}$$

with $E_0$ a constant and the regularized version

$$\tilde{E}(\tilde{w}) = E(\tilde{w}) + \nu \frac{1}{2} \tilde{w}^T \tilde{w}. \tag{2.30}$$

The gradient of these cost functions is

$$\begin{array}{rcl} \frac{\partial E}{\partial w} & = & b + Hw = 0 \\ \frac{\partial \tilde{E}}{\partial \tilde{w}} & = & b + H\tilde{w} + \nu\tilde{w} = 0. \end{array} \tag{2.31}$$

Consider the eigenvalues and eigenvectors of $H$:

$$Hu_j = \lambda_j u_j \tag{2.32}$$

and expand $w$ and $\tilde{w}$ as follows

$$w = \sum_j w_j u_j, \qquad \tilde{w} = \sum_j \tilde{w}_j u_j. \tag{2.33}$$

One can write then $Hw - H\tilde{w} - \nu\tilde{w} = 0$ or $\sum_j Hw_j u_j - \sum_j H\tilde{w}_j u_j - \nu \sum_j \tilde{w}_j u_j = 0$ which gives $\sum_j (\lambda_j w_j - \lambda_j \tilde{w}_j - \nu\tilde{w}_j) u_j = 0$. Finally this results into

$$\tilde{w}_j = \frac{\lambda_j}{\lambda_j + \nu} w_j$$

meaning that if $\lambda_j \gg \nu$ then $\tilde{w}_j \simeq w_j$ and if $\lambda_j \ll \nu$ the components $|\tilde{w}_j| \ll |w_j|$ are suppressed. Hence thanks to the regularization mechanism one can implicitly work with less parameters than the number of unknown interconnection weights. The so-called **effective number of parameters**

$$\gamma = \sum_j \frac{\lambda_j}{\lambda_j + \nu} \tag{2.34}$$

is then characterized by the number of eigenvalues $\lambda_j$ that is larger than the regularization constant $\nu$. In general a large regularization constant $\nu$ large will lead to a smaller model structure giving smaller variance but larger bias. A small value for $\nu$ will keep the model structure larger which gives large variance but smaller bias.

For MLPs one may in principle also take different penalty factors for each layer, e.g.

$$\Omega(w) = \nu_1 \frac{1}{2} \sum_{w \in \mathcal{W}_1} w^T w + \nu_2 \frac{1}{2} \sum_{w \in \mathcal{W}_2} w^T w \tag{2.35}$$

where $\mathcal{W}_1, \mathcal{W}_2$ denote the sets of weights for layer 1 and 2, although this is not often done in practice.

### 2.4.2 Early stopping and validation set

Instead of applying regularization to the cost function one might minimize the original cost function $E(w)$ instead of the regularized $\tilde{E}(w)$ but do **early stopping** of the training process when the minimal error on a separate validation set is obtained. One works then with three sets: a training set for estimation of the model, a validation set to stop the training process earlier and a test set which contains data that are not used to derive the model (Fig.2.5).

Somewhat surprisingly, it is possible to show that early stopping is closely related to regularization. Conceptually this can be understood as follows according to [1]. A quadratic approximation of the cost

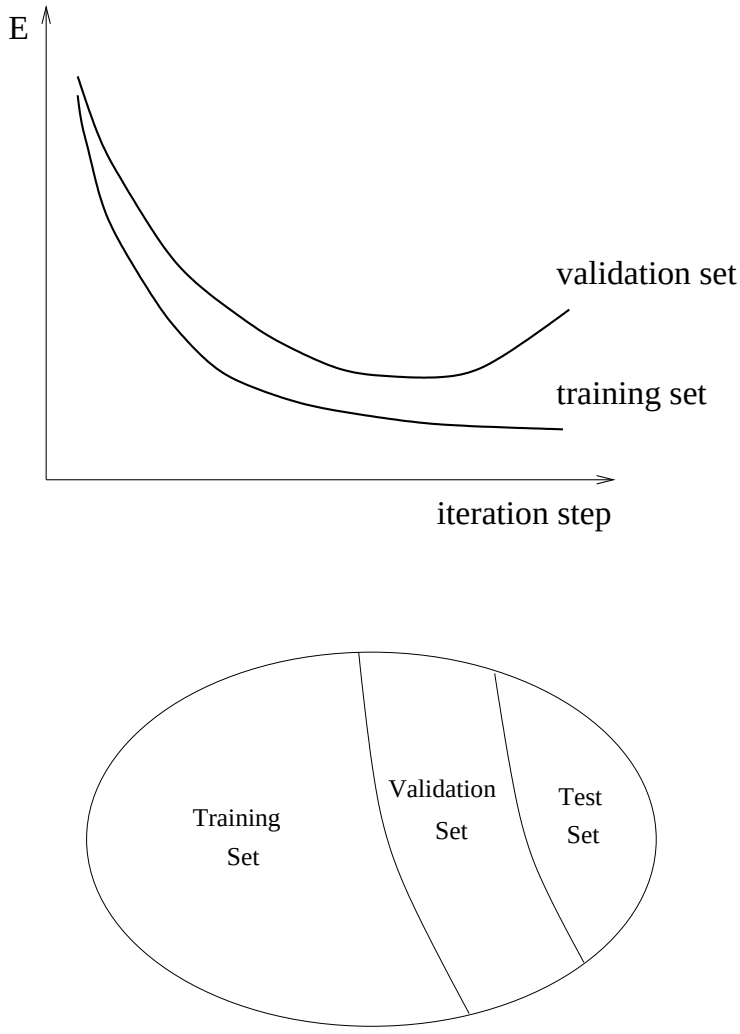



Figure 2.5: *Training, validation and test set where the validation set is used for early stopping of the training process. The designer is responsible for making a good choice!*

function at the minimum $w^*$ gives

$$E = E_0 + \frac{1}{2}(w - w^*)^T H (w - w^*) \tag{2.36}$$

with constant $E_0$ and positive definite Hessian $H$. Consider a simple gradient descent

$$w^{(\tau)} = w^{(\tau-1)} - \eta \nabla E \tag{2.37}$$

with iteration step $\tau$, learning rate $\eta$ and $w^{(0)} = 0$ for simplicity. One can show then that

$$w_j^{(\tau)} = \{1 - (1 - \eta\lambda_j)^\tau\} w_j^* \tag{2.38}$$

where $w_j = w^T u_j$ with $u_j, \lambda_j$ eigenvectors and eigenvalues of $H$ respectively. As $\tau \to \infty$ one has $w^{(\tau)} \to w^*$, provided $|1 - \eta\lambda_j| < 1$. Assume that the training is stopped then after $\tau$ steps, one obtains

$$\begin{array}{rl} w_j^{(\tau)} \simeq w_j^* & \text{when} \quad \lambda_j \gg 1/(\eta\tau) \\ |w_j^{(\tau)}| \ll |w_j^*| & \text{when} \quad \lambda_j \ll 1/(\eta\tau). \end{array} \tag{2.39}$$

Hence $1/(\eta\tau)$ plays a similar role as the regularization parameter $\nu$.

# Chapter 3

# Generalization

## 3.1 Interpretation of network outputs

When deriving models in general the goal is not to memorize data but rather to model the underlying generator of the data, characterized by $p(x,t)$ which is the joint probability density of inputs $x$ and targets $t$. One has

$$p(x,t) = p(t|x)p(x) \tag{3.1}$$

with $p(t|x)$ the probability density of $t$ given a particular value of $x$ and $p(x)$ the unconditional density of $x$.

According to [1] let us consider now the cost $E$ in the limit $N \to \infty$, i.e. for an infinite data set size:

$$\begin{aligned} E &= \lim_{N\to\infty} \frac{1}{2N} \sum_{n=1}^{N} \{y(x_n;w) - t_n\}^2 \\ &= \tfrac{1}{2} \int\int \{y(x;w) - t\}^2 p(t,x)\, dtdx \\ &= \tfrac{1}{2} \int\int \{y(x;w) - t\}^2 p(t|x)p(x)\, dtdx. \end{aligned} \tag{3.2}$$

Defining the conditional averages

$$\begin{aligned} \langle t|x \rangle &= \int t\, p(t|x) dt \\ \langle t^2|x \rangle &= \int t^2\, p(t|x) dt \end{aligned} \tag{3.3}$$

one has

$$\begin{aligned}\{y-t\}^2 &= \{y-\langle t|x\rangle+\langle t|x\rangle-t\}^2 \\ &= \{y-\langle t|x\rangle\}^2+2\{y-\langle t|x\rangle\}\{\langle t|x\rangle-t\}+\{\langle t|x\rangle-t\}^2.\end{aligned} \tag{3.4}$$

As a result one obtains

$$E = \tfrac{1}{2}\int\{y(x;w)-\langle t|x\rangle\}^2p(x)dx+\tfrac{1}{2}\int\{\langle t^2|x\rangle-\langle t|x\rangle^2\}p(x)dx. \tag{3.5}$$

The first term means that at the local minimum $w^*$ of the error function, we have

$$y(x;w^*)=\langle t|x\rangle, \tag{3.6}$$

meaning that the output approximates the conditional average of the target data. The second term represents the intrinsic noise on the data and sets a lower limit on the achievable error.

## 3.2 Bias and variance

In practice we often have only one specific and finite data set $D$. In order to eliminate the dependency on a specific data set $D$ one considers

$$\mathcal{E}_D[\{y(x)-\langle t|x\rangle\}^2]$$

where $\mathcal{E}_D$ denotes the ensemble average. The question is then how close the estimated mapping is to the true one. In order to answer this question we write

$$\begin{aligned}\{y(x)-\langle t|x\rangle\}^2 &= \{y(x)-\mathcal{E}_D[y(x)]+\mathcal{E}_D[y(x)]-\langle t|x\rangle\}^2 \\ &= \{y(x)-\mathcal{E}_D[y(x)]\}^2+\{\mathcal{E}_D[y(x)]-\langle t|x\rangle\}^2+ \\ &\quad 2\{y(x)-\mathcal{E}_D[y(x)]\}\{\mathcal{E}_D[y(x)]-\langle t|x\rangle\}.\end{aligned} \tag{3.7}$$

By taking the expectation over the ensemble of the data sets one gets

$$\mathcal{E}_D[\{y(x)-\langle t|x\rangle\}^2] = \{\mathcal{E}_D[y(x)]-\langle t|x\rangle\}^2+\mathcal{E}_D[\{y(x)-\mathcal{E}_D[y(x)]\}^2]. \tag{3.8}$$

The bias and variance follow then from these two terms. The first term is given by

$$(\text{bias})^2=\frac{1}{2}\int\{\mathcal{E}_D[y(x)]-\langle t|x\rangle\}^2p(x)dx. \tag{3.9}$$

The second term gives

$$\text{variance} = \frac{1}{2}\int \mathcal{E}_D[\{y(x) - \mathcal{E}_D[y(x)]\}^2]p(x)dx. \tag{3.10}$$

A first illustration of the bias-variance trade-off is given in terms of two extreme cases shown in Fig.3.1. Suppose

$$t_n = h(x_n) + \epsilon_n \tag{3.11}$$

with true function $h(x)$ and $y(x)$ an estimate of $h(x)$. Consider the following two extremes:

1. Fix $y(x) = g(x)$ independent of any data set. Then:

   $$\mathcal{E}_D[y(x)] = g(x) = y(x)$$

   which gives zero variance but a large bias.

2. Consider an exact interpolant of the data. Then

   $$\mathcal{E}_D[y(x)] = \mathcal{E}_D[h(x) + \epsilon] = h(x) = \langle t|x\rangle$$

   which gives zero bias, but large variance according to

   $$\mathcal{E}_D[\{y(x) - \mathcal{E}_D[y(x)]\}^2] = \mathcal{E}_D[\{y(x) - h(x)\}^2] = \mathcal{E}_D[\epsilon^2].$$

In order to make the theoretical arguments more specific, consider e.g. a situation where one generates 100 data sets by sampling a true underlying function $h(x)$ and adding noise. Note that $h(x)$ is known in this experiment, but in a real situation it would of course be unknown. Estimate the mappings $y_i(x)$ for $i = 1, 2, ..., 100$, e.g. 100 MLPs where each MLP results from a generated data set. One gets

$$\begin{aligned}
\overline{y}(x) \qquad &= \frac{1}{100}\sum_{i=1}^{100} y_i(x) \\
(\text{Bias})^2 \quad &= \sum_n \{\overline{y}(x_n) - h(x_n)\}^2 \\
\text{Variance} \quad &= \sum_n \frac{1}{100}\sum_{i=1}^{100}\{y_i(x_n) - \overline{y}(x_n)\}^2.
\end{aligned}$$

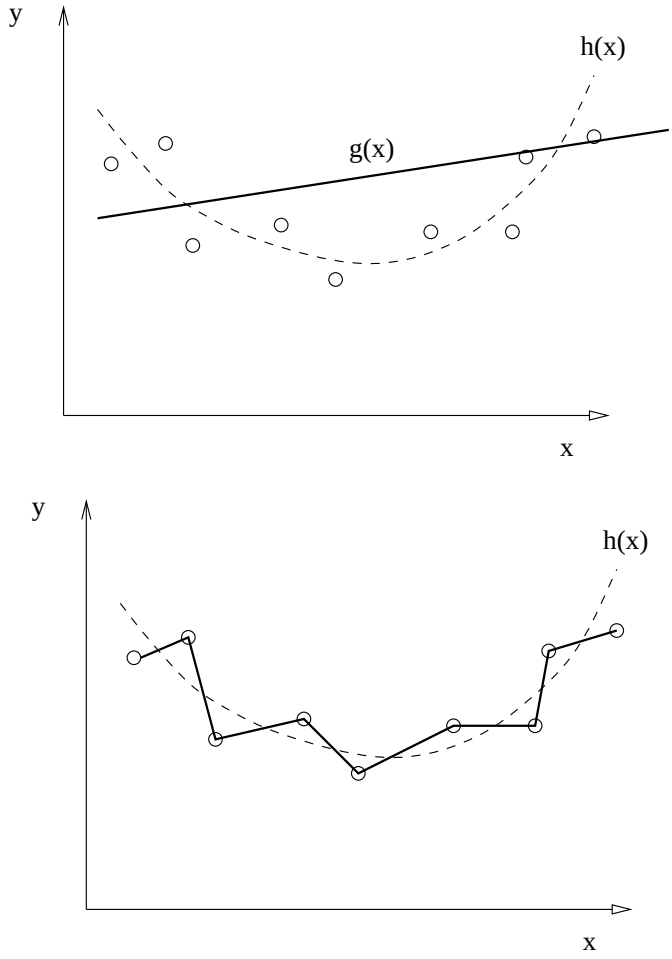



Figure 3.1: *Illustration of bias-variance trade-off by means of two extreme cases: (Top) fixing a function $g(x)$ independent of the data: zero variance but large bias; (Bottom) exact interpolant of the data under all circumstances: zero bias but large variance. The function $h(x)$ is the true function.*

### 3.2.1 Cross-validation

Working with a specific validation set has two main disadvantages. First the results might be quite sensitive with respect to the specific data points belonging to that validation set. Secondly when working with a training, validation and test set a part of the training data can no longer be used for training as it belongs then to the validation set.

A good procedure is then to apply **cross-validation** (Wahba, 1975) (Fig.3.2). One divides the training set into a number of $S$ segments and trains in each run on $S-1$ segments. The error on the sum of the segments that were left out in the $S$ runs serves then as a validation set performance. A typical choice (which is both computationally attractive and of good statistical quality) is $S = 10$ (called 10-fold cross validation). In the extreme case one can take $S = N$ meaning that one has $N$ runs with $N-1$ data points (called leave-one-out cross-validation). This is only recommended for small data sets and certainly not for datamining applications with millions of data points.

## 3.3 Complexity criteria

So far we have stressed that it is dangerous to do the training solely on the basis of a training set without looking at the performance on independent sets, otherwise a bad overfitting solution will be the result.

In fact this is also the message of complexity criteria which state that one should not only try to minimize training errors but also keep the model complexity as low as possible. This is basically the Occam's razor[1] For nonlinear models the following complexity criterion holds (Moody, 1992)

$$\text{GPE} = \text{Training Error} + \frac{\gamma}{N}\sigma^2$$

where GPE is the generalized prediction error and $\gamma = \sum_i \lambda_i/(\lambda_i + \nu)$ is the effective number of parameters. $\lambda_i$ denote the eigenvalues of the Hessian of the unregularized cost function and $\nu$ the regularization constant. $\sigma^2$ is the variance of the noise on the data and $N$ the number

[1]William Occam (1280-1349) was a monk who claimed that "*No more things should be presumed to exist than are absolutely necessary*" (Entia non sunt multiplicanda praeter necessitatem).

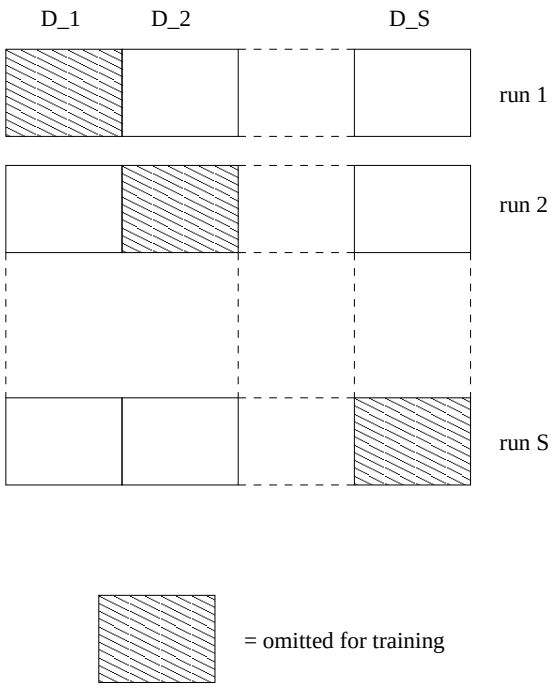



Figure 3.2: *In the cross-validation method a number of $S$ runs are done where in each run a different part of the data set is omitted and the validation error is finally checked as the sum of error costs on the sets that were omitted.*

of training data. Eigenvalues $\lambda_i \ll \nu$ do not contribute to the sum. This criterion is an extension of the well-known Akaike information criterion which states the following for the prediction error (PE):

$$\text{PE} = \text{Training Error} + \frac{W}{N}\sigma^2$$

where $W$ is the number of adjustable parameters of the model. This Akaike criterion is only valid for linear models.

## 3.4 Pruning

In order to improve the generalization performance of the trained models one can remove interconnection weights that are irrelevant. This procedure is called *pruning*. We discuss here methods of optimal brain damage, optimal brain surgeon and weight elimination.

In **Optimal Brain Damage** (Le Cun, 1990) one considers the error cost function change due to small changes in the interconnection weights:

$$\delta E \simeq \sum_i \frac{\partial E}{\partial w_i}\delta w_i + \frac{1}{2}\sum_i \sum_j H_{ij}\delta w_i \delta w_j + \mathcal{O}(\delta w^3) \tag{3.12}$$

where $H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$ is the Hessian. One takes the following assumption after convergence:

$$\delta E \simeq \frac{1}{2}\sum_i H_{ii}\,\delta w_i^2 \tag{3.13}$$

and one measures the relative importance of the interconnection weights by

$$H_{ii}w_i^2/2$$

which are also called *saliency values*. The algorithm looks as follows:

*Pruning algorithm (optimal brain damage):*

1. Choose a relatively large initial network architecture.
2. Train the network in the usual way until some stopping criterion is satisfied.
3. Compute the saliencies $H_{ii}w_{ii}^2/2$.

4. Sort weights by saliency and delete low-saliency weights

5. Go to 2 and repeat until some overall stopping criterion is reached.

Optimal brain damage has been applied to the recognition of handwritten zip codes, where networks with 10000 interconnection weights have been pruned by a factor 4.

Another pruning method is **Optimal Brain Surgeon** (Hassibi and Stork, 1993). This method can be understood as follows. By neglecting higher order terms one has after convergence

$$\delta E = \frac{1}{2}\delta w^T H \delta w. \tag{3.14}$$

Setting a weight $w_i$ to zero corresponds to $\delta w_i = -w_i$ or

$$e_i^T \delta w + w_i = 0 \tag{3.15}$$

where $e_i$ is a unit vector. One considers then the optimization problem

$$\min_{\delta w} \delta E = \frac{1}{2}\delta w^T H \delta w \quad \text{s.t.} \quad e_i^T \delta w + w_i = 0 \tag{3.16}$$

with Lagrangian

$$\mathcal{L}(\delta w, \lambda) = \frac{1}{2}\delta w^T H \delta w - \lambda(e_i^T \delta w + w_i) \tag{3.17}$$

and conditions for optimality:

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial(\delta w)} = H\delta w - \lambda e_i = 0 \;\rightarrow\; \delta w = \lambda H^{-1} e_i \\ \frac{\partial \mathcal{L}}{\partial \lambda} = e_i^T \delta w + w_i = 0 \;\rightarrow\; \lambda e_i^T H^{-1} e_i = \lambda [H^{-1}]_{ii} = -w_i. \end{cases} \tag{3.18}$$

This results into

$$\delta w = -\frac{w_i}{[H^{-1}]_{ii}} H^{-1} e_i \tag{3.19}$$

and

$$\delta E_i = \frac{1}{2}\frac{w_i^2}{[H^{-1}]_{ii}}. \tag{3.20}$$

The pruning algorithm looks then as follows:

*Pruning algorithm (optimal brain surgeon):*

1. Train a relatively large network to a minimum of the error function.

2. Evaluate the inverse Hessian $H^{-1}$.

3. Evaluate $\delta E_i$ for each value of $i$ and select the value of $i$ which gives the smallest increase in error.

4. Update all the weights according to $\delta w = -\frac{w_i}{[H^{-1}]_{ii}} H^{-1} e_i$.

5. Go to 3 and repeat until some stopping criterion is reached.

The performance of this algorithm is better than optimal brain damage.

In order to eliminate weights one may also take a different regularization term instead of weight decay. The following technique is called **Weight Elimination** (Weigend, 1990):

$$\tilde{E} = E + \nu \sum_i \frac{(w_i/c)^2}{1 + (w_i/c)^2}. \tag{3.21}$$

This algorithm is more likely to eliminate weights (i.e. putting weights to zero) than the weight decay method. A drawback is the choice of the additional tuning parameter $c$.

## 3.5 Committee networks and combining models

A common approach is to train several models and select the best individual model. However, one might improve the results in view of the bias-variance trade-off by forming a committee of networks and combining the models.

Let us consider $L$ trained networks $y_i(x)$, $i = 1, ..., L$. This could be either a number of $L$ trained MLPs or $L$ totally different kind of static models. Assume the true regression function $h(x)$ is such that

$$y_i(x) = h(x) + \epsilon_i(x) \quad i = 1, ..., L \tag{3.22}$$

where $\epsilon_i(x)$ is the error function related to the $i$-th network. The average sum-of-squares error for the individual model $y_i(x)$ is then

$$E_i = \mathcal{E}[\{y_i(x) - h(x)\}^2] = \mathcal{E}[\epsilon_i^2]. \tag{3.23}$$

In 1993 Perrone showed that the performance of the committee network can be better than the performance of the best single network. One can take a simple averaged committee network or a weighted average committee network.

A simple **Average Committee Network** is given by

$$y_{COM}(x) = \frac{1}{L}\sum_{i=1}^{L} y_i(x). \tag{3.24}$$

The error of the committee network is

$$E_{COM} = \mathcal{E}[\,(\frac{1}{L}\sum_{i=1}^{L} y_i(x) - h(x))^2\,] = \mathcal{E}[\,(\frac{1}{L}\sum_{i=1}^{L} \epsilon_i)^2\,]. \tag{3.25}$$

One can show that

$$(\sum_{i=1}^{L} \epsilon_i)^2 \leq L\sum_{i=1}^{L} \epsilon_i^2 \;\Rightarrow\; E_{COM} \leq E_{AV} \tag{3.26}$$

where

$$E_{AV} = \frac{1}{L}\sum_{i=1}^{L} E_i = \frac{1}{L}\sum_{i=1}^{L} \mathcal{E}[\epsilon_i^2]. \tag{3.27}$$

Hence the results improve by this averaging network.

One can further improve the committee method by taking a **Weighted Average Committee Network** (Fig.3.3):

$$\begin{aligned} y_{COM}(x) &= \sum_{i=1}^{L} \alpha_i y_i(x) \\ &= h(x) + \sum_{i=1}^{L} \alpha_i \epsilon_i(x) \end{aligned} \tag{3.28}$$

where $\sum_{i=1}^{L} \alpha_i = 1$. One considers the correlation matrix

$$C_{ij} = \mathcal{E}[\epsilon_i(x)\epsilon_j(x)] \tag{3.29}$$

where in practice one works e.g. with a finite-sample approximation on training data

$$C_{ij} = \frac{1}{N}\sum_{n=1}^{N}[y_i(x_n) - t_n][y_j(x_n) - t_n]. \tag{3.30}$$

The committee error equals

$$\begin{aligned} E_{COM} &= \mathcal{E}[\{y_{COM}(x) - h(x)\}^2] \\ &= \mathcal{E}[\,(\sum_{i=1}^{L}\alpha_i\epsilon_i)\,(\sum_{j=1}^{L}\alpha_j\epsilon_j)\,] \\ &= \sum_{i=1}^{L}\sum_{j=1}^{L}\alpha_i\alpha_j C_{ij} = \alpha^T C\alpha. \end{aligned} \tag{3.31}$$

An optimal choice of $\alpha$ is made as follows:

$$\min_{\alpha}\frac{1}{2}\alpha^T C\alpha \;\text{ s.t. }\; \sum_{i=1}^{L}\alpha_i = 1. \tag{3.32}$$

From the Lagrangian

$$\mathcal{L}(\alpha,\lambda) = \frac{1}{2}\alpha^T C\alpha - \lambda(\sum_{i=1}^{L}\alpha_i - 1)$$

one obtains the following conditions for optimality:

$$\begin{cases} \frac{\partial L}{\partial \alpha} &= C\alpha - \lambda\vec{1} = 0 \\ \frac{\partial L}{\partial \lambda} &= \vec{1}^T\alpha - 1 = 0 \rightarrow \lambda = \frac{1}{\vec{1}^T C^{-1}\vec{1}} \end{cases} \tag{3.33}$$

with optimal solution

$$\alpha = \frac{C^{-1}\vec{1}}{\vec{1}^T C^{-1}\vec{1}} \tag{3.34}$$

and corresponding committee error

$$E_{COM} = 1/(\vec{1}^T C^{-1}\vec{1}) \tag{3.35}$$

where $\vec{1} = [1; 1; ...; 1]$. In case of an ill-conditioned matrix $C$ one can apply additional regularization to the $C$ matrix or also impose $\alpha_i \geq 0$ to avoid large negative and positive weights. Other popular methods which are related to committees are bagging and boosting[2] [**?**, 28].

[2]At http://www.boosting.org/ many material about tutorials, papers and software is available.

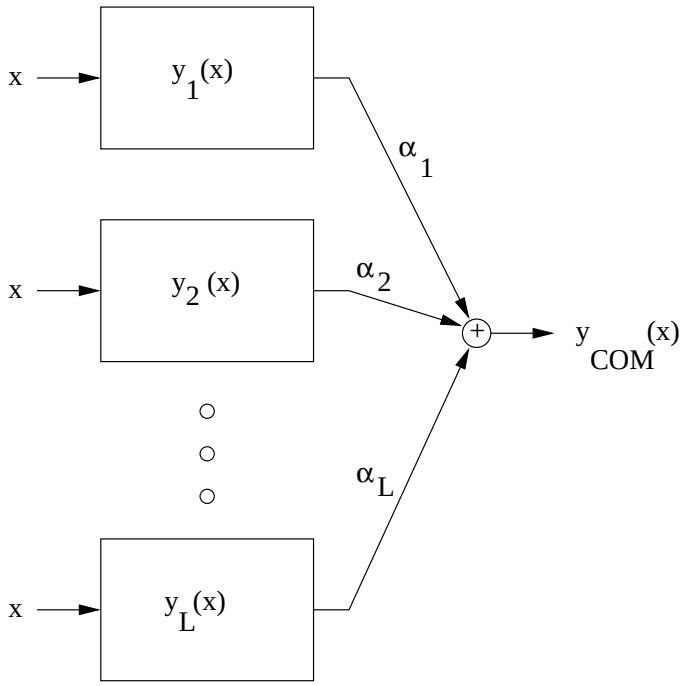



Figure 3.3: *Combining models using a committee network.*

# Chapter 4

# Unsupervised Learning

In this Chapter we discuss aspects of dimensionality reduction, nonlinear PCA, cluster algorithms, vector quantization and self-organizing maps. This chapter is mainly based on [1, 2, 10, 12, 41]. In unsupervised learning operations are only done on the input space data and not on target output data. Often these methods are applied in order to get a better idea of the underlying structure and density of the data and extracting knowledge from it. In methods like self-organizing maps one has an additional objective in mind of visualizing the data.

## 4.1 Dimensionality reduction and nonlinear PCA

Often one tries do reduce large dimensional input spaces to lower dimensions such as in PCA analysis. A reduction of the input space for parametric models usually leads to less parameters to be estimated which is desirable from the viewpoint of complexity criteria.

Consider patterns $x \in \mathbb{R}^d$ in the original space and transformed inputs $z \in \mathbb{R}^m$ in a lower dimensional space with $m \ll d$. Conceptually this problem can be understood as an encoding/decoding problem where one tries to minimize the reconstruction error. For the encoder mapping

$$z = G(x) \tag{4.1}$$

and decoder mapping

$$\hat{x} = F(z) \tag{4.2}$$

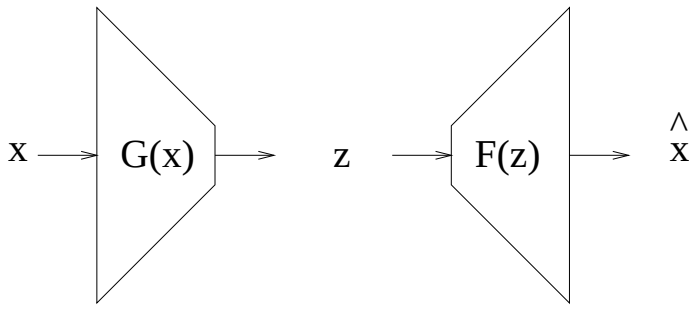



Figure 4.1: *Information bottleneck in dimensionality reduction. Linear PCA is obtained for linear mappings $F(\cdot), G(\cdot)$. Nonlinear PCA analysis is obtained by considering nonlinear mappings that can be parameterized by MLPs.*

one has the following objective in mind of minimizing the squared distortion error

$$\begin{aligned} \min E &= \frac{1}{N}\sum_{i=1}^{N} \|x_i - \hat{x}_i\|_2^2 \\ &= \frac{1}{N}\sum_{i=1}^{N} \|x_i - F(G(x_i))\|_2^2. \end{aligned} \tag{4.3}$$

An important special case is when $F(\cdot), G(\cdot)$ are linear mappings. It can be proven that this corresponds to PCA analysis. However, one can take these mappings also nonlinear and parameterize it e.g. by means of MLPs. In that case it leads to so-called nonlinear PCA analysis. The variable $z$ in the reduced dimension leads to an information bottleneck (Fig.4.1). These problems have been studied also in rate-distortion theory within information theory. Some examples about dimensionality reduction and linear versus nonlinear PCA analysis are shown in Fig.4.2, Fig.4.3 and Fig.4.4.

## 4.2 Cluster algorithms

An important class of unsupervised learning methods are cluster algorithms. In this case one aims at finding groups of points which are located close to each other according to a chosen distance measure.

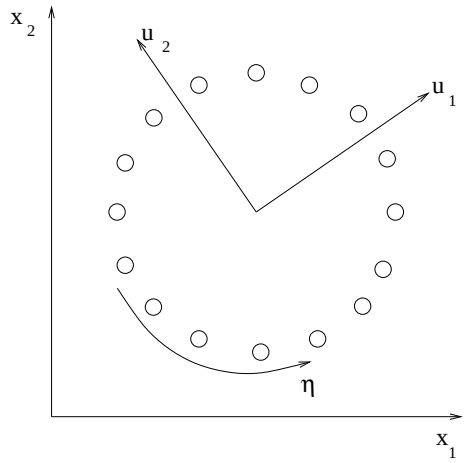



Figure 4.2: *The data set in two dimensions has intrinsic dimensionality 1. The data can be explained in terms of the single parameter $\eta$, while linear PCA is unable to detect the lower dimensionality.*

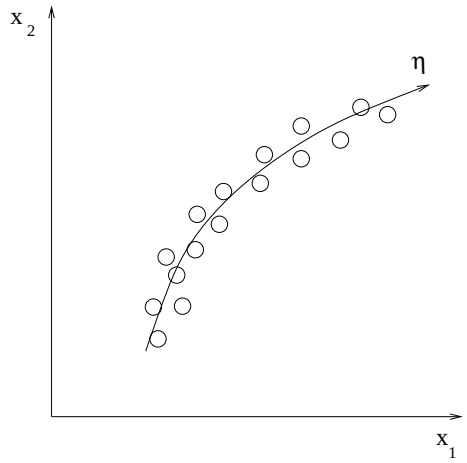



Figure 4.3: *Addition of a small level of noise to data in two dimensions having an intrinsic dimensionality of 1 can increase its dimensionality to 2. Nevertheless, the data can be represented to a good approximation by a single variable $\eta$.*

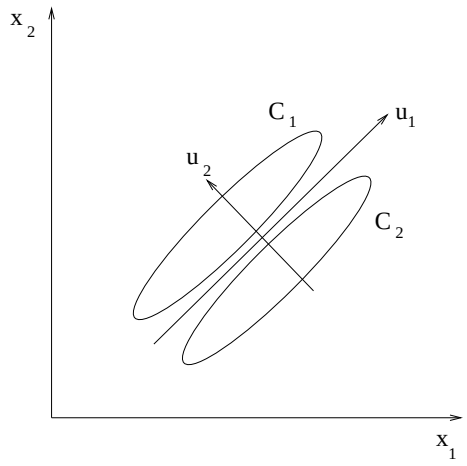



Figure 4.4: *In this simple classification problem linear PCA analysis would discard the discriminatory information. Suppose data are taken from two Gaussian distributed classes $\mathcal{C}_1$ and $\mathcal{C}_2$. Dimensionality reduction to one dimension would give a projection of the data onto vector $u_1$ which would remove all ability to discriminate between the two classes.*

A well-known method is the $K$-means algorithm which works as follows.

*$K$-means cluster algorithm*

1. Choose $K$ initial cluster centers $z_1(1), ..., z_K(1)$.

2. At iteration step $k$, distribute the samples $\{x\}$ among the $K$ cluster domains by

$$x \in S_j(k) \text{ if } \|x - z_j(k)\| < \|x - z_i(k)\| \ , \ \forall i = 1, 2, ..., K$$

   where $S_j(k)$ denotes the set of samples whose cluster center is $z_j(k)$ at iteration step $k$.

3. New cluster centers:

$$z_j(k+1) = \frac{1}{N} \sum_{x \in S_j(k)} x, \quad j = 1, 2, ..., K.$$

This minimizes the performance index

$$J_j = \sum_{x \in S_j(k)} \|x - z_j(k+1)\|^2, \quad j = 1, 2, ..., K.$$

4. If $z_j(k+1) = z_j(k)$ for $j = 1, 2, ..., K$ the algorithm has converged. Otherwise go to step 2.

Note that in this method one has to choose the number of centers $K$ (Fig.4.5). The method also depends on the initial choice of the clusters. The performance of the method is characterized by the performance indices $J_j$ for each of the clusters $j = 1, 2, ..., K$. These indices can be combined into a single performance measure. The $K$-means algorithm can also be considered as a rough approximation to the E-step of the EM algorithm for a mixture of Gaussians. Density estimation methods such as mixture models can indeed also be considered as unsupervised learning. There also exist many other clustering methods, e.g. isodata algorithm, hierarchical clustering, agglomerative clustering, divisive clustering etc. [5, 8, 12].

## 4.3 Vector quantization

Vector quantization is a method which is somewhat related to clustering but while the precise objective of clustering is sometimes rather vague (finding interesting groups of samples), in vector quantization one optimizes a quantization error (distortion measure) for a fixed number of prototype vectors.

Given data $x(k)$ for $k = 1, 2, ...$ and initial prototype centers $c_j(0)$ for $j = 1, 2, ..., m$ ($m$ centers) the vector quantization method (competitive learning or stochastic approximation version) is given by the following algorithm.

*Vector quantization algorithm*

1. Determine nearest center $c_j(k)$ to data point $x(k)$. For the squared error loss function the nearest neighbour rule is

$$j = \arg\min_l \|x(k) - c_l(k)\|.$$

Finding the nearest center is often called competition among centers.

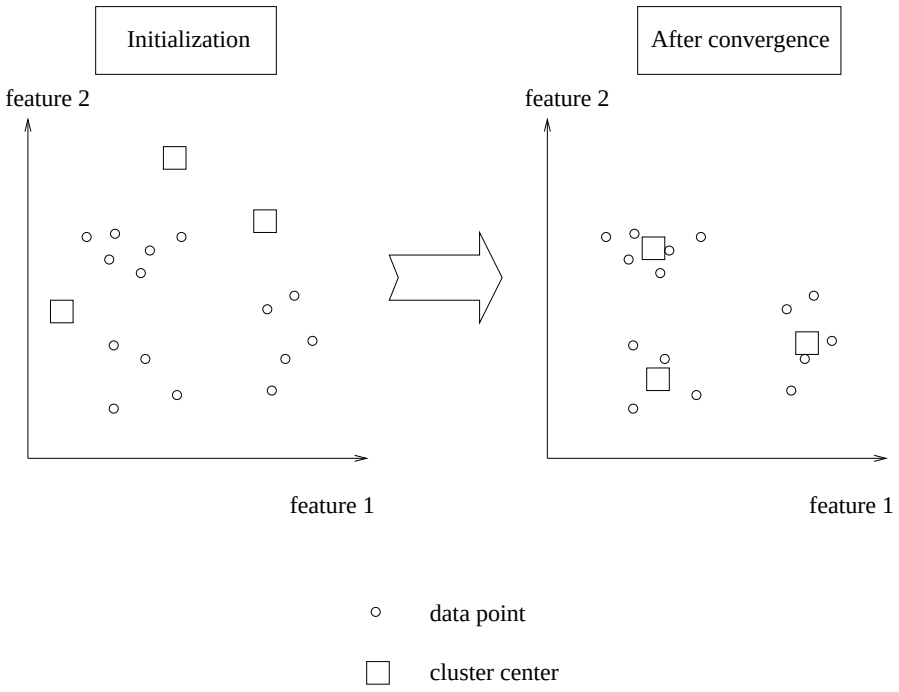



Figure 4.5: *K-means clustering algorithm.*

2. Update centers:

$$\begin{aligned} c_j(k+1) &= c_j(k) + \gamma(k)[x(k) - c_j(k)] \\ k &:= k+1 \end{aligned}$$

where $\gamma(k)$ should meet the conditions for stochastic approximation,

i.e. $\lim_{k\to\infty} \gamma(k) = 0$, $\sum_{k=1}^{\infty} \gamma(k) = \infty$, $\sum_{k=1}^{\infty} \gamma^2(k) < \infty$.

The vector quantizer $Q$ is a mapping $Q : \mathbb{R}^d \to \mathcal{C}$ where $\mathcal{C} = \{c_1, c_2, ..., c_m\}$ (winning units). The space $\mathbb{R}^d$ is partitioned into the regions $\mathcal{R}_1, ..., \mathcal{R}_m$ where $\mathcal{R}_j = \{x \in \mathbb{R}^d : Q(x) = c_j\}$. The algorithm minimizes a distortion measure of the form

$$\int \|x - \sum_j c_j I(x \in \mathcal{R}_j)\|^2 p(x) dx \tag{4.4}$$

where $\mathcal{R}_j$ corresponds to the $j$-th partition region and $I(x \in \mathcal{R}_j) = 1$ if $x \in \mathcal{R}_j$ and 0 otherwise. The partition regions of a vector quantizer are non-overlapping and cover the entire input space $\mathbb{R}^d$. The optimal vector quantizer has the so-called nearest-neighbor partition (or Voronoi partition) illustrated in Fig.4.6.

## 4.4 Self-organizing maps

Self-organising maps are largely based on vector quantization but in addition it aims at having a visual representation on a low dimensional map. Hence, self-organining maps (SOM) try to represent the underlying density of the input data by means of prototype vectors and at the same time one projects the higher dimensional input data to a map of neurons (also called nodes or units) such that the data can be visualised. Typically one has a projection to a 2-dimensional grid of neurons. In this way the SOM compresses information while preserving the most important topological and metric relationships of the primary data items on the display.

Consider input training data $x_i \in \mathbb{R}^d$ for $i = 1, ..., N$, prototype vectors $c_j \in \mathbb{R}^d$ for $j = 1, ..., b$ and map coordinates $z_j \in \mathbb{R}^2$ for $j = 1, ..., b$ where $N$ denotes the number of training data and $b$ the number of neurons. One can say that the neurons have in fact two

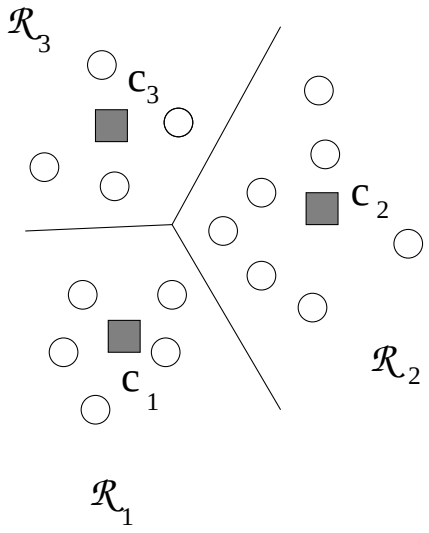



Figure 4.6: *Vector quantization leading to Voronoi partition with prototype vectors* $c_1, c_2, c_3$ *and regions* $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$.

positions: in the 'input' space one has the prototypes $c_j \in \mathbb{R}^d$ while in the 'output' space one has the map coordinates $z_j \in \mathbb{R}^2$ both for $j = 1, ..., b$.

Dimensionality reduction is done by projecting to a 2-dimensional space with coordinates $z \in \mathbb{R}^2$. One typically takes a 2-dimensional grid of neurons: $\psi = \{\psi_1, \psi_2, ..., \psi_b\}$, where $\psi(j)$ denotes the $j$-th element of $\psi$. A simple illustration is given for $b = 16$ neurons in Fig.4.7. One can take several possible grid choices e.g. hexagonal grid, rectangular grid or others (Fig.4.8). A typical choice for number of neurons is $b = 5\sqrt{N}$, where the computational load increases quadratically with $b$.

For the SOM algorithm there exist both batch versions and on-line adaptive versions which we discuss now. The batch algorithm is given by

*SOM Batch algorithm (off-line):*
Repeat until convergence:

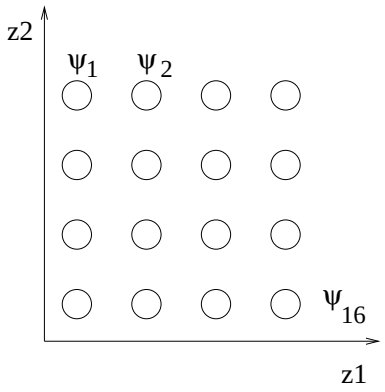



Figure 4.7: *Simple illustration of a 2-dimensional grid of neurons in SOM with* 16 *neurons.*

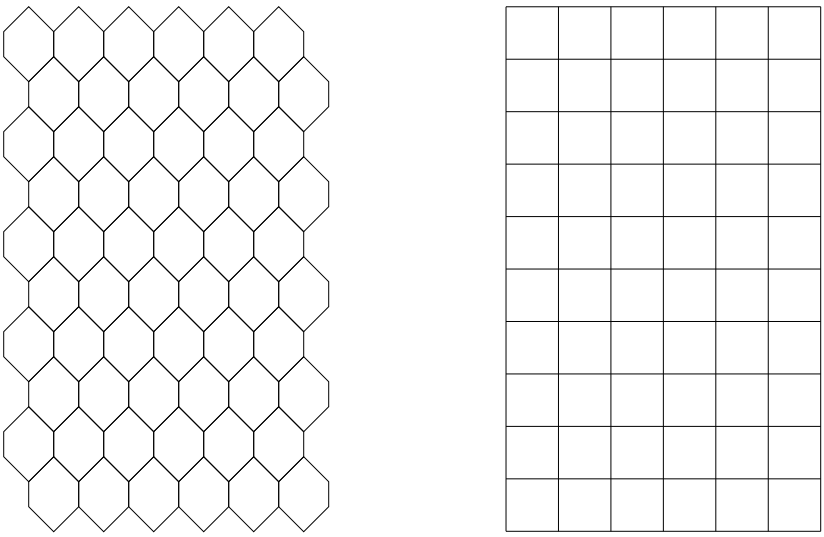


Figure 4.8: *Some possible grid choices for the SOM.*

1. Projection:
$$\begin{aligned} j_* &= \arg\min_j \|x_i - c_j\|_2^2 \\ \hat{z}_i &= \psi(j_*),\ i = 1, ..., N. \end{aligned}$$

2. Update centers
$$c_j = F(\psi(j), \sigma),\ j = 1, ..., b$$
where
$$F(z, \sigma) = \frac{\sum_{i=1}^N K_\sigma(z, z_i) x_i}{\sum_{i=1}^N K_\sigma(z, z_i)}$$
with
$$K_\sigma(z, z_i) = \exp(-\|z - z_i\|^2 / 2\sigma^2).$$

3. Decrease $\sigma$:
$$\sigma(k) = \sigma_{\text{initial}} (\frac{\sigma_{\text{final}}}{\sigma_{\text{initial}}})^{k/k_{max}}$$
at iteration step $k$. The initial value of $\sigma$ is chosen such that the neighborhood covers all the unit. The final value controls the smoothness of the mapping.

The online SOM algorithm is given by:

*SOM On-line algorithm:*

1. Determine winning unit (also called best matching unit)
$$z(k) = \psi(\arg\min_j \|x(k) - c_j(k-1)\|)$$

2. Update all units
$$\begin{aligned} c_j(k) &= c_j(k-1) + \beta(k) K_{\sigma(k)}(\psi(j), z(k)) \left[x(k) - c_j(k-1)\right] \\ k &:= k + 1 \end{aligned}$$
for $j = 1, ..., b$.

3. Decrease learning rate $\beta(k)$ and width $\sigma(k)$.

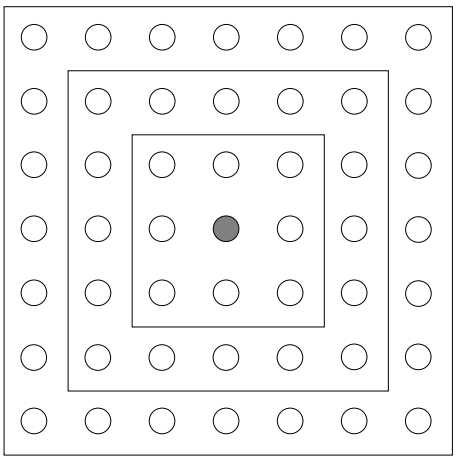


Figure 4.9: *Examples of SOM neighborhood size functions (sizes 1,2,3). In a similar way sizes can be controlled by $\sigma$ when using the $K_\sigma(z, z_i) = \exp(-\|z - z_i\|^2/2\sigma^2)$ function.*

The batch version is usually faster than the on-line version and is often preferred. The initialization of the SOM can be done either at random or based upon the two principal eigenvectors from PCA analysis. Missing values within the data set are usually excluded from the distance calculations.

For the neighborhood functions several choices are possible. In the case of the choice $K_\sigma(z, z_i) = \exp(-\|z - z_i\|^2/2\sigma^2)$ the neighborhood size is controlled by $\sigma$.

One also often works with neighborhood sizes 1, 2 or 3 for the neurons, which is illustrated in Fig.4.9.

Nice visualizations can be made by SOMs. It is important, however, to carefully interpret the results after the training of the SOM. One gets insight by looking at the color or black/white maps. Depending on color definition by the user, dark areas might mean that the data are very dense and clustered in that region (many data close to each other). This information is obtained by calculating distances between the prototype vectors. In case the class labels of the data are given (supervised information) one can also show these on the SOM map. In the so-called WEBSOM, the SOM method has been applied to problems of webmining where millions of documents have

been processed (Fig.4.10) [34].

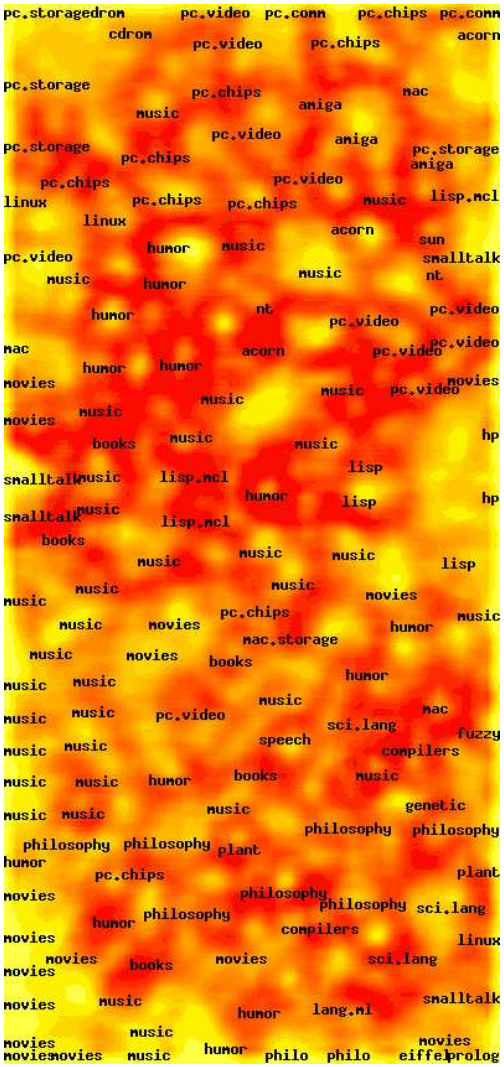



Figure 4.10: *Example of a SOM map after training where insight about the clustering of the data is obtained from the color or black/white regions. This figure illustrates a result from webmining by means of the WEBSOM http://websom.hut.fi/websom/.*

# Chapter 5

# Nonlinear modelling

## 5.1 Model structures and parameterizations

### 5.1.1 State space models

In order to explain how neural networks can be used within a context of dynamical systems and how one comes from feedforward to recurrent networks, we discuss first some elements of systems theory in order to put these problems within the right perspective [15, 16, 45].

It is well-known that discrete time linear systems with input vector $u \in \mathbb{R}^m$, output vector $y \in \mathbb{R}^l$ and state vector $x \in \mathbb{R}^n$ can be represented in state space form as

$$\left\{ \begin{array}{rcl} x_{k+1} & = & Ax_k + Bu_k \\ y_k & = & Cx_k. \end{array} \right. \tag{5.1}$$

In the context of nonlinear systems one can consider nonlinear state space descriptions

$$\left\{ \begin{array}{rcl} x_{k+1} & = & f(x_k, u_k) \\ y_k & = & g(x_k) \end{array} \right. \tag{5.2}$$

where $f(\cdot) : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $g(\cdot) : \mathbb{R}^n \rightarrow \mathbb{R}^l$ are nonlinear mappings. When one parameterizes these nonlinear functions by means of a feedforward neural network (such as MLP and RBF) one obtains a recurrent neural network.

While the stability of linear systems can be completely understood by checking the eigenvalues of the system matrix $A$ (for discrete time

systems all eigenvalues should belong to the open unit disk in the complex plane), stability issues of nonlinear systems are much more complicated. They can possess e.g. multiple equilibrium points, limit cycles, chaotic behaviour etc. Let us consider a simple example to illustrate this for an autonomous system with $x_k \in \mathbb{R}^3$

$$x_{k+1} = f(x_k) \tag{5.3}$$

parameterized by an MLP as

$$x_{k+1} = W \tanh(V x_k) \tag{5.4}$$

with $W, V \in \mathbb{R}^{3\times 3}$. By taking random choices for these matrices several kinds of behaviour can be obtained even in such a simple neural network. As shown in Fig.5.1, depending on random choices of $W$ and $V$, one might obtain global asymptotic stability, multiple equilibria, quasi-periodic behaviour and chaos. This example shows that dynamical models which are parameterized by neural nets can represent complex behaviour, but on the other hand it also means that the analysis of such systems is highly non-trivial [47].

The systems that have been discussed here are deterministic. However, in many real life situations the system is corrupted by noise. System representations that take this into account are

$$\left\{ \begin{array}{rcl} x_{k+1} & = & Ax_k + Bu_k + w_k \\ y_k & = & Cx_k + v_k \end{array} \right. \tag{5.5}$$

for the linear case and

$$\left\{ \begin{array}{rcl} x_{k+1} & = & f(x_k, u_k, w_k) \\ y_k & = & g(x_k, v_k) \end{array} \right. \tag{5.6}$$

for the nonlinear case where $w_k, v_k$ denote the process noise and observation (or measurement) noise, respectively.

### 5.1.2 Input/output models

Let us take a look now at input/output (I/O) models instead of state space models. Usually one employs a NARX (Nonlinear ARX (Auto Regressive with eXogenous input)) model structure

$$\hat{y}_k = f(y_{k-1}, y_{k-2}, ..., y_{k-p}, u_{k-1}, u_{k-2}, ..., u_{k-p}) \tag{5.7}$$

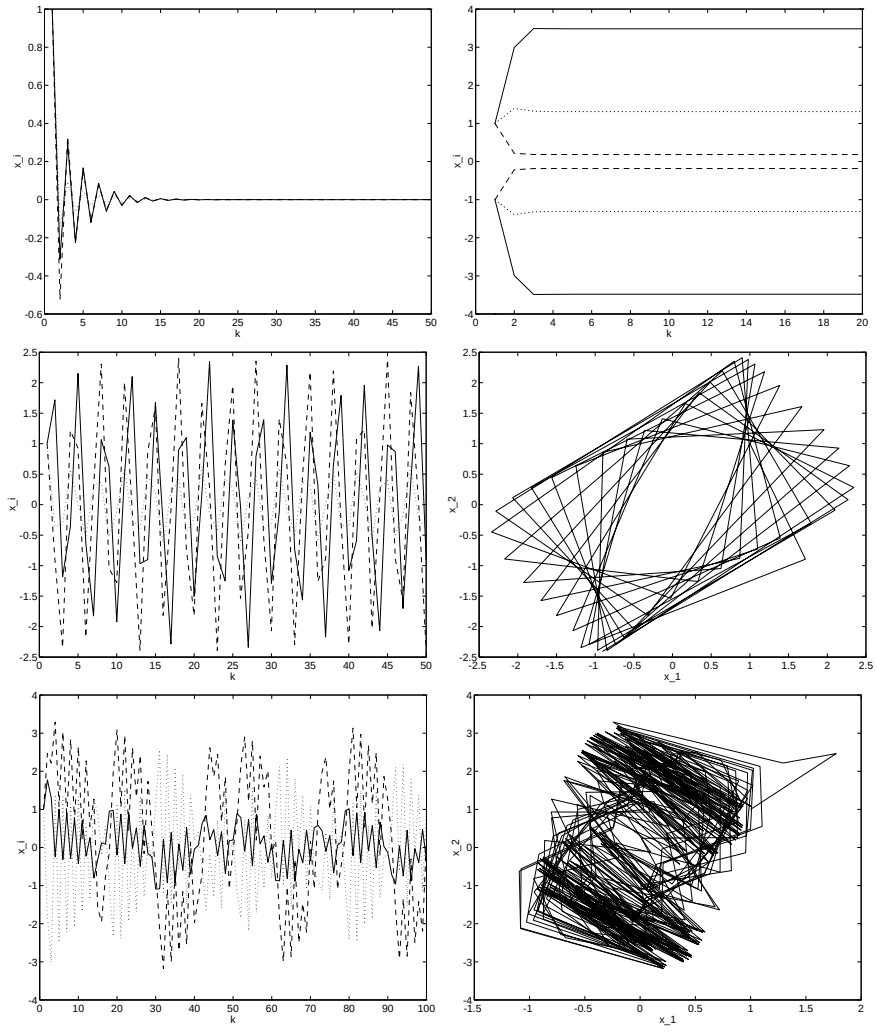



Figure 5.1: *Example of several kinds of behaviour in a simple recurrent network: (Top-Left) global asymptotic stability; (Top-Right) multiple equilibria; (Middle) quasi-periodic behaviour; (Bottom) chaos.*

where $y_k$ denotes the measured output at discrete time instant $k$, $u_k$ the input at time $k$ and $\hat{y}_k$ the estimated output at time $k$. The number $p$ corresponds to the order of the system. A parameterization of the nonlinearity $f(\cdot)$ by neural networks leads to a static feedforward model from the viewpoint of neural networks, despite the fact that this is used in a dynamical systems context. The reason is that the equation of this model does not have a recursion on the variable $\hat{y}_k$ (it appears only at the left hand side of the equation). The parameterization by an MLP with one hidden layer gives

$$\hat{y}_{k+1} = w^T \sigma(V z_{k|k-p} + \beta) \tag{5.8}$$

with $z_{k|k-p} = [y_{k-1}; y_{k-2}; ...; y_{k-p}; u_{k-1}; u_{k-2}; ...; u_{k-p}]$.

In the so-called NARMAX (Nonlinear ARMAX (Auto Regressive Moving Average with eXogenous input)) one also tries to model the noise influence by taking the following model structure

$$\begin{aligned} y_k = f( \quad & y_{k-1}, y_{k-2}, ..., y_{k-p}, u_{k-1}, u_{k-2}, ..., u_{k-p}, \\ & \epsilon_{k-1}, \epsilon_{k-2}, ..., \epsilon_{k-q}) + \epsilon_k \end{aligned} \tag{5.9}$$

with error $\epsilon_k = y_k - \hat{y}_k$. Another model structure which leads to recurrent networks instead of feedforward networks is the NOE (Nonlinear OE (Output Error)) model structure

$$\hat{y}_k = f(\hat{y}_{k-1}, \hat{y}_{k-2}, ..., \hat{y}_{k-p}, u_{k-1}, u_{k-2}, ..., u_{k-p}). \tag{5.10}$$

Note that one has a recursion now on the variable $\hat{y}_k$ in contrast with the NARX model.

In system identification one has given input/output data available in order to model the system (Fig.5.2). After choosing a model structure and parameterizing it by means of neural networks a cost function is formulated in the unknown interconnection weights and optimized by a certain optimization algorithm (Fig.5.3).

### 5.1.3 Time-series prediction models

Model structures for time-series predictions (Fig.5.4) are obtained by omitting the external input signals $u_k$ from the previous NARX models. One has

$$\hat{y}_{k+1} = f(y_k, y_{k-1}, ..., y_{k-p}) \tag{5.11}$$

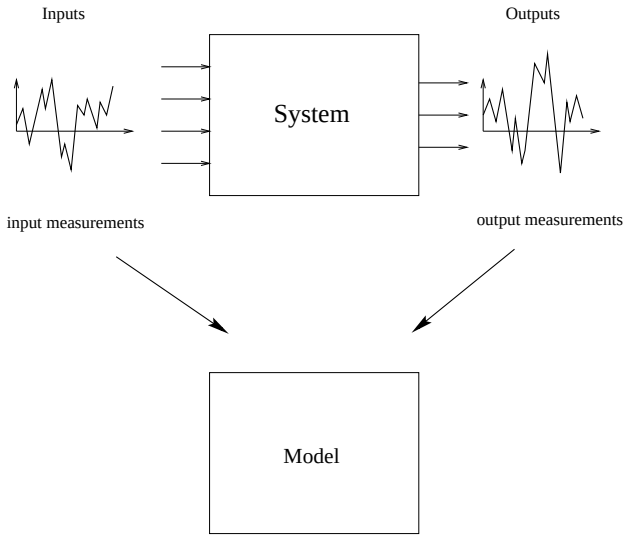



Figure 5.2: *In system identification one aims at estimating models from given input/output data measurements* $\{u_k, y_k\}$ *on systems.*

which is parameterized as

$$\hat{y}_{k+1} = w^T \tanh(V\,[y_k; y_{k-1}; ...; y_{k-p}] + \beta). \tag{5.12}$$

It is not necessary that the past values $y_k, y_{k-1}, ..., y_{k-p}$ are subsequent in time; certain values could be omitted or values at different time scales could be taken. In order to generate predictions, the true values $y_k$ are replaced by the estimated values $\hat{y}_k$ and the iterative prediction is generated by the recurrent network

$$\hat{y}_{k+1} = w^T \tanh(V\,[\hat{y}_k; \hat{y}_{k-1}; ...; \hat{y}_{k-p}] + \beta) \tag{5.13}$$

for a given initial condition.

## 5.2 Recurrent networks and dynamic backpropagation

Previously we have discussed the backpropagation algorithm for feedforward networks. In fact it is basically a method for analytically computing the gradient of a cost function which is defined on a static network (static nonlinear function or feedforward neural network). When

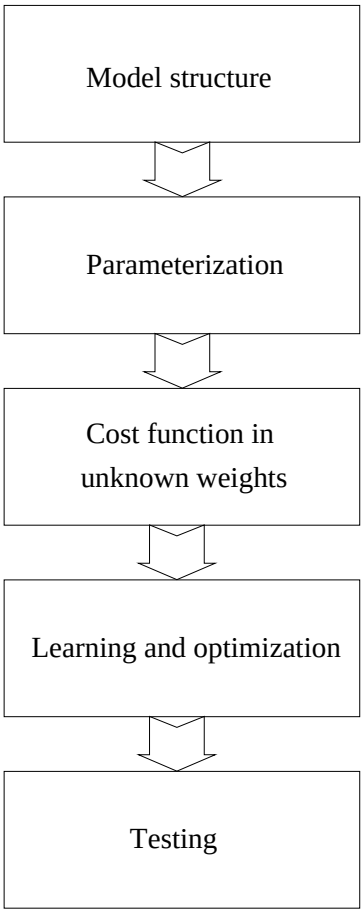
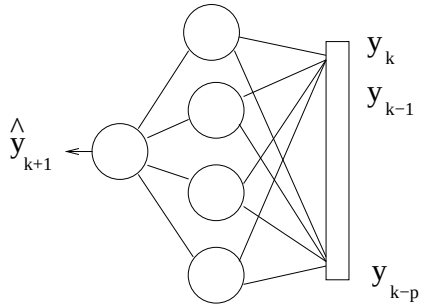



Figure 5.3: *Several stages in nonlinear modelling and system identification.*

*Training mode*




*Iterative prediction mode*

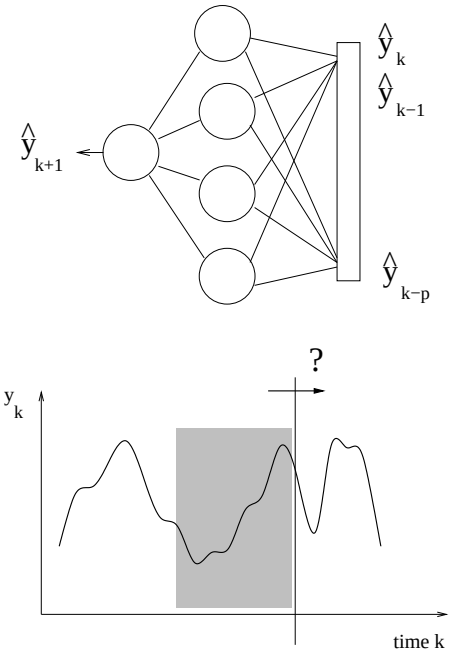



Figure 5.4: *Time-series prediction with neural networks: (Top) identification with a NARX model structure; (Middle) iterative prediction as a recurrent network by replacing the true values $y_k$ with the estimated values $\hat{y}_k$ at the input of the network; (Bottom) illustrative example of time-series prediction.*

the cost function is defined on a *recurrent* neural network (i.e. a network that contains feedback interconnections) then the computation becomes (even) more complicated.

Let us start and illustrate the ideas by a simple example. Consider a nonlinear state space model

$$\dot{x}(t) = f[x(t), \alpha, u(t)], \quad x(t_0) = x_0 \tag{5.14}$$

with state vector $x(t) \in \mathbb{R}^n$, input vector $u(t) \in \mathbb{R}^m$. Suppose that $\alpha \in \mathbb{R}$ is some unknown parameter to be adjusted in order to minimize the following objective

$$J(\alpha) = \int_0^T [x(t) - d(t)]^2 dt \tag{5.15}$$

where $d(t) \in \mathbb{R}^n$ is a desired reference trajectory in state space. When one applies one of the previously discussed optimization algorithms the gradient of the cost function is needed, which is given by

$$\frac{\partial J}{\partial \alpha} = \int_0^T 2[x(t) - d(t)] \frac{\partial x}{\partial \alpha} dt. \tag{5.16}$$

Clearly we need to find then an expression for $\frac{\partial x}{\partial \alpha}$. This follows from a so-called **sensitivity model**

$$\frac{\partial \dot{x}(t)}{\partial \alpha} = f_x(t) \frac{\partial x(t)}{\partial \alpha} + f_\alpha(t), \quad \frac{\partial x(t_0)}{\partial \alpha} = 0 \tag{5.17}$$

where the Jacobian $f_x(t) = \frac{\partial f}{\partial x}$ and $f_\alpha(t)$ are evaluated around the nominal values. This sensitivity model is obtained by deriving the left hand side and right hand side of the state space model with respect to $\alpha$. This right hand side contains both an implicit and explicit dependency on $\alpha$ and the sensitivity model follows then from application of the chain rule. Finally, one can see that one has to simulate the augmented system consisting of the state space model and the sensitivity model with state vectors $x$ and $\frac{\partial x(t)}{\partial \alpha}$, respectively.

A similar reasoning holds for input/output representations of systems. Consider for example a second order scalar nonlinear differential equation

$$\ddot{y} + F(\alpha, \dot{y}) + y = u \tag{5.18}$$

where $F(\cdot)$ is some nonlinear function depending on $\dot{y}$ and $\alpha$. The initial condition $y(0) = y_{1_0}$, $\dot{y}(0) = y_{2_0}$ and $\alpha$ some scalar parameter to be adjusted. Let us take a cost function

$$J(\alpha) = \int_0^T [y(t) - d(t)]^2 dt \tag{5.19}$$

where $d(t)$ is a reference trajectory. The gradient of the cost function is given by

$$\frac{\partial J(\alpha)}{\partial \alpha} = \int_0^T 2[y(t) - d(t)] \frac{\partial y(t)}{\partial \alpha} \, dt. \tag{5.20}$$

The variable $\partial y(t)/\partial \alpha$ follows from the sensitivity model

$$\ddot{z} + \frac{\partial F}{\partial \dot{y}} \dot{z} + z = -\frac{\partial F}{\partial \alpha} \tag{5.21}$$

with $z = \partial y/\partial \alpha$ and initial state $z(0) = \dot{z}(0) = 0$.

This method of computing sensitivity models is applicable both to discrete time and continuous time systems and to models that depend on a vector of unknown interconnection weights instead of a single adjustable parameter $\alpha$.

The procedure can be applied now to the training of recurrent neural networks either in input-output form (like the previously discussed NOE model) or in state space form. Other procedures instead of sensitivity models also exist such as backpropagation through time [51]. The use of a sensitivity model in combination with a gradient based local optimization algorithm is called *dynamic backpropagation* [40]. We illustrate this for a nonlinear state space model that is parameterized by MLPs (neural state space models)

$$\left\{ \begin{array}{rcl} \hat{x}_{k+1} & = & W_{AB} \tanh(V_A \hat{x}_k + V_B u_k + \beta_{AB}) \; ; \; \hat{x}_0 = x_0 \\ \hat{y}_k & = & W_{CD} \tanh(V_C \hat{x}_k + V_D u_k + \beta_{CD}) \end{array} \right. \tag{5.22}$$

where $W_*, V_*, \beta_*$ are interconnection matrices and bias vectors with dimensions compatible with the input vector $u_k$, estimated state vector $\hat{x}_k$ and estimated output vector $\hat{y}_k$. Given a training set of $N$ input/output data the cost function is given by

$$J(\theta) = \frac{1}{2N} \sum_{k=1}^{N} \epsilon_k(\theta)^T \epsilon_k(\theta) \tag{5.23}$$

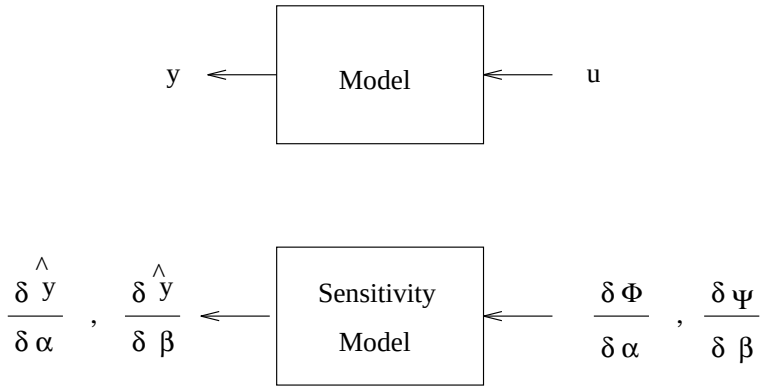



Figure 5.5: *In dynamic backpropagation of recurrent networks a sensitivity model is simulated simultaneously with the model in order to compute the gradient of the cost function.*

with $\epsilon_k = y_k - \hat{y}_k(\theta)$ and unknown parameter vector $\theta = [W_{AB}(:); V_A(:); V_B(:); \beta_{AB}; W_{CD}(:); V_C(:); V_D(:); \beta_{CD}]$. where the matlab notation ':' denotes columnwise scanning of a matrix to a vector. The gradient of the cost function is

$$\frac{\partial J}{\partial \theta} = \frac{1}{N} \sum_{k=1}^{N} \frac{\partial \epsilon_k(\theta)}{\partial \theta}^T \epsilon_k(\theta) \tag{5.24}$$

where

$$\frac{\partial \epsilon_k(\theta)}{\partial \theta} = \frac{\partial [y_k - \hat{y}_k(\theta)]}{\partial \theta} = -\frac{\partial \hat{y}_k(\theta)}{\partial \theta} \tag{5.25}$$

and $\frac{\partial \hat{y}_k(\theta)}{\partial \theta}$ follows from the sensitivity model. The neural state space model is of the form

$$\left\{ \begin{array}{rcl} \hat{x}_{k+1} & = & \Phi(\hat{x}_k, u_k, \epsilon_k; \alpha) \ ; \ \hat{x}_0 = x_0 \text{ given} \\ \hat{y}_k & = & \Psi(\hat{x}_k, u_k; \beta) \end{array} \right. \tag{5.26}$$

with $\alpha, \beta$ elements of parameter vector $\theta$. The sensitivity model becomes (Fig.5.5)

$$\left\{ \begin{array}{rcl} \frac{\partial \hat{x}_{k+1}}{\partial \alpha} & = & \frac{\partial \Phi}{\partial \hat{x}_k} \cdot \frac{\partial \hat{x}_k}{\partial \alpha} + \frac{\partial \Phi}{\partial \alpha} \\ \frac{\partial \hat{y}_k}{\partial \alpha} & = & \frac{\partial \Psi}{\partial \hat{x}_k} \cdot \frac{\partial \hat{x}_k}{\partial \alpha} \\ \frac{\partial \hat{y}_k}{\partial \beta} & = & \frac{\partial \Psi}{\partial \beta}. \end{array} \right. \tag{5.27}$$

From an elementwise notation of the model

$$\left\{\begin{array}{lcl} \hat{x}^i & := & \sum_j {w_{AB}}_j^i \, \tanh(\sum_r {v_A}_r^j \, \hat{x}^r + \sum_s {v_B}_s^j \, u^s + {\beta_{AB}}^j) \\ \hat{y}^i & = & \sum_j {w_{CD}}_j^i \, \tanh(\sum_r {v_C}_r^j \, \hat{x}^r + \sum_s {v_D}_s^j \, u^s + {\beta_{CD}}^j), \end{array}\right. \tag{5.28}$$

and after defining $\phi^l = \sum_r {v_A}_r^l \, \hat{x}^r + \sum_s {v_B}_s^l \; u^s + \beta_{AB}^l$ $\rho^l = \sum_r {v_C}_r^l \, \hat{x}^r + \sum_s {v_D}_s^l \; u^s + \beta_{CD}^l$, one obtains the derivatives

$$\frac{\partial \Phi}{\partial \alpha} : \quad \left\{\begin{array}{lcl} \frac{\partial \Phi^i}{\partial {w_{AB}}_l^j} & = & \delta_j^i \, \tanh(\phi^l) \\ \frac{\partial \Phi^i}{\partial {v_A}_l^j} & = & {w_{AB}}_j^i \, (1 - \tanh^2(\phi^j)) \, \hat{x}^l \\ \frac{\partial \Phi^i}{\partial {v_B}_l^j} & = & {w_{AB}}_j^i \, (1 - \tanh^2(\phi^j)) \, u^l \\ \frac{\partial \Phi^i}{\partial {\beta_{AB}}^j} & = & {w_{AB}}_j^i \, (1 - \tanh^2(\phi^j)) \end{array}\right. \tag{5.29}$$

$$\frac{\partial \Psi}{\partial \beta} : \quad \left\{\begin{array}{lcl} \frac{\partial \Psi^i}{\partial {w_{CD}}_l^j} & = & \delta_j^i \, \tanh(\rho^l) \\ \frac{\partial \Psi^i}{\partial {v_C}_l^j} & = & {w_{CD}}_j^i \, (1 - \tanh^2(\rho^j)) \, \hat{x}^l \\ \frac{\partial \Psi^i}{\partial {v_D}_l^j} & = & {w_{CD}}_j^i \, (1 - \tanh^2(\rho^j)) \, u^l \\ \frac{\partial \Psi^i}{\partial {\beta_{CD}}^j} & = & {w_{CD}}_j^i \, (1 - \tanh^2(\rho^j)) \end{array}\right.$$

$$\frac{\partial \Phi}{\partial \hat{x}_k} : \quad \frac{\partial \Phi^i}{\partial \hat{x}^r} = \sum_j {w_{AB}}_j^i \, (1 - \tanh^2(\phi^j)) \, {v_A}_r^j$$

$$\frac{\partial \Psi}{\partial \hat{x}_k} : \quad \frac{\partial \Psi^i}{\partial \hat{x}^r} = \sum_j {w_{CD}}_j^i \, (1 - \tanh^2(\rho^j)) \, {v_C}_r^j.$$

where $\delta_i^j$ denotes the Kronecker delta.

# Chapter 6

# Support vector machines

In this Chapter we discuss Support Vector Machines (SVM)[1] for linear and nonlinear classification and function estimation. The chapter is mainly based on [3, 14, 18, 19, 24, 46, 48].

## 6.1 Motivation

Despite the fact that classical neural networks (MLPs, RBF networks) have nice properties such as universal approximation and reliable algorithms presently exist for this class of techniques, they still have a number of persistent drawbacks. A first problem is the existence of many local minima solutions. Although many of these local solutions actually can be good solutions, it is often inconvenient, e.g. from a statistical perspective. Another problem is how to choose the number of hidden units (Fig.6.1).

The theory of Support Vector Machines (SVMs) sheds a new light on these problems. Support vector machines have been introduced by Vapnik. In fact the original idea of linear SVMs dates back already from the sixties but it became more important and popular in recent years when extensions to general nonlinear SVMs have been made [18, 19]. In SVMs one works with kernel based representations of the network allowing linear, polynomial, splines, RBF and other kernels. Several operations on kernels are allowed and for specific applications such as textmining string kernels can be used. The so-

[1] At http://www.kernel-machines.org/ many material about tutorials, papers and software is available.

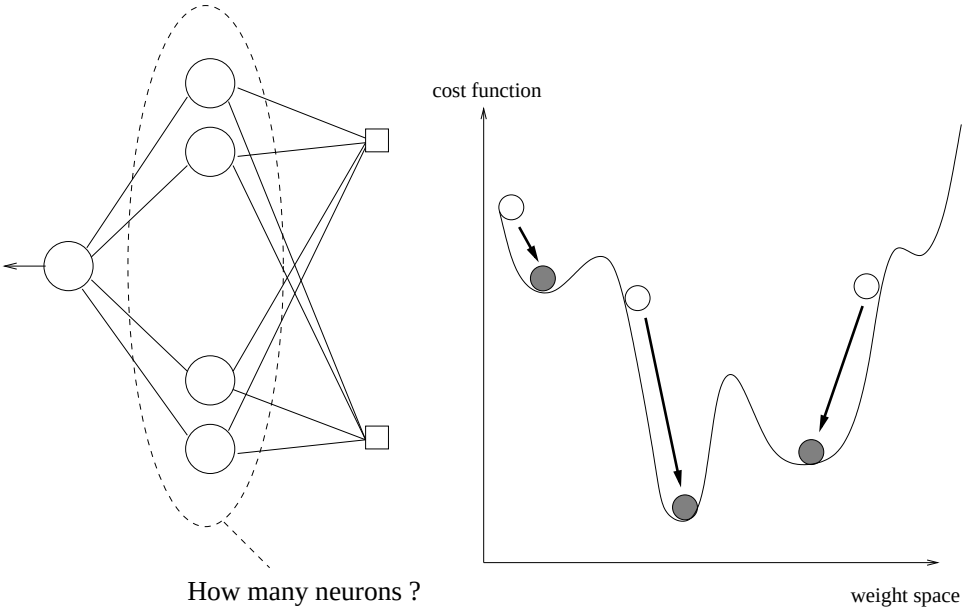



Figure 6.1: *Drawbacks of classical neural networks: (Top) problem of number of hidden units; (Bottom) existence of many local minima solutions.*

lution is characterized by a convex optimization problem (typically quadratic programming) which has a unique solution in contrast with MLPs. Moreover also the model complexity (e.g. number of hidden neurons) follows from the solution to this convex optimization problem. The support vectors can be interpreted as informative points. SVM models also work well in huge dimensional input spaces. SVMs have been originally proposed within the context of statistical learning theory, where also probabilistic bounds on the generalization error of models have been derived. These bounds are expressed in terms of the VC (Vapnik-Chervonenkis) dimension, which can be considered as a combinatorial measure for model complexity. A drawback, however, is that SVMs have been mainly developed for static problems as classification, function and density estimation. Also a kernel version of PCA and kernel versions of cluster algorithms exist which will not be treated here in this course. SVMs have been successfully applied to many real-life problems including text categorisation, image recognition, handwritten digit recognition, bioinformatics, protein homology detection and financial engineering.

## 6.2 Maximal margin classifiers and linear SVMs

### 6.2.1 Margin

In Fig.6.2 an illustrative example is given of a separable problem in a two-dimensional feature space. One can see that there exist several separating hyperplanes that separate the data of the two classes (data depicted by 'x' and '+'). Towards the development of SVM theory it is important to define a unique separating hyperplane. This is done by maximizing the distance to the nearest points of the two classes. According to Vapnik, one can do then a rescaling of the problem such that $\min_i |w^T x_i + b| = 1$, i.e. the scaling is done such that the point closest to the hyperplane has a distance $1/\|w\|_2$. The *margin* between the classes is then equal to $2/\|w\|_2$. Maximizing the margin corresponds then to minimizing $\|w\|_2$.

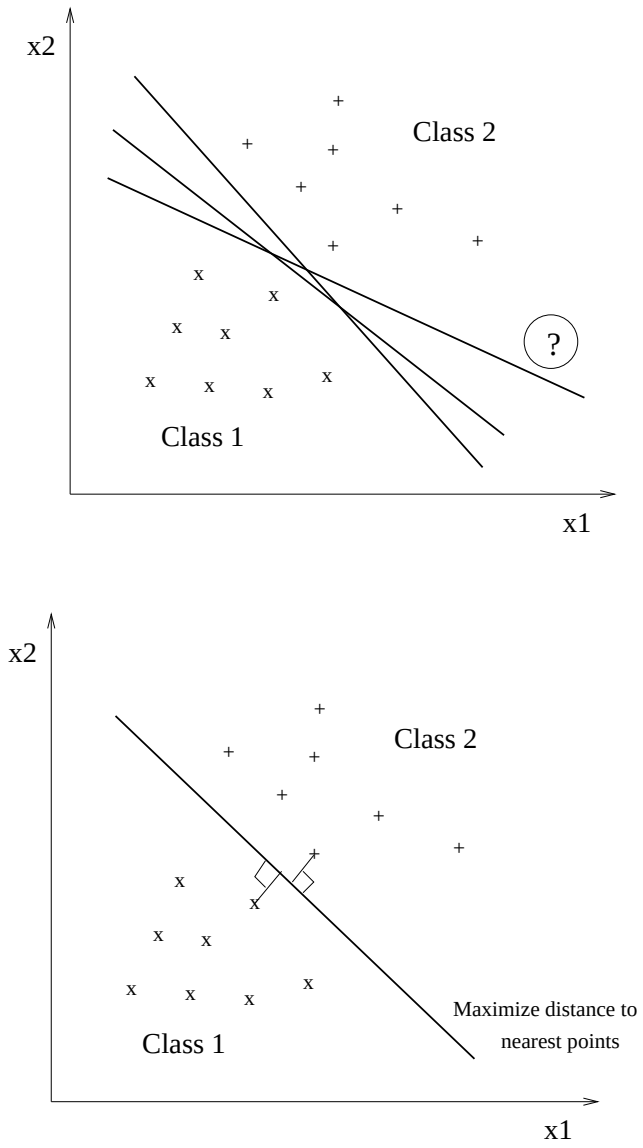



Figure 6.2: *Linear classification: (Top) separable problem where the separating hyperplane is not unique; (Bottom) definition of a unique hyperplane which is maximizing the distance to the nearest points.*

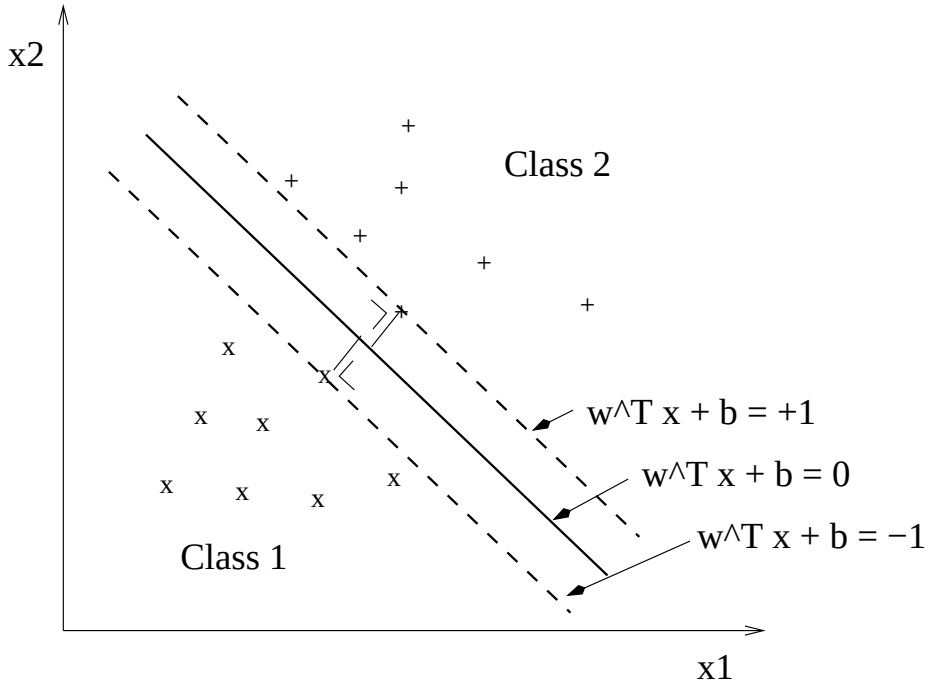



Figure 6.3: *Linear classification: definition of unique separating hyperplane. The margin is the distance between the dashed lines.*

### 6.2.2 Linear SVM classifier: separable case

After introducing the margin concept, we are in a position now to formulate the linear SVM classifier, which was originally proposed by Vapnik (1964).

Consider a given training set $\{x_k, y_k\}_{k=1}^N$, input patterns $x_k \in \mathbb{R}^n$ and output patterns $y_k \in \mathbb{R}$ with class labels $y_k \in \{-1, +1\}$. Assume

$$\begin{cases} w^T x_k + b \geq +1 \quad , & \text{if} \quad y_k = +1 \\ w^T x_k + b \leq -1 \quad , & \text{if} \quad y_k = -1 \end{cases} \tag{6.1}$$

which is equivalent to

$$y_k[w^T x_k + b] \geq 1, \quad k = 1, ..., N. \tag{6.2}$$

One formulates then the optimization problem:

$$\min_{w,b} \frac{1}{2} w^T w \quad \text{s.t.} \quad y_k[w^T x_k + b] \geq 1, \quad k = 1, ..., N. \tag{6.3}$$

The Lagrangian for this problem is

$$\mathcal{L}(w, b; \alpha) = \frac{1}{2} w^T w - \sum_{k=1}^{N} \alpha_k \{y_k[w^T x_k + b] - 1\} \tag{6.4}$$

with Lagrange multipliers $\alpha_k \geq 0$ for $k = 1, ..., N$ (later called the support values). The solution is given by the saddle point of the Lagrangian

$$\max_{\alpha} \min_{w,b} \mathcal{L}(w, b; \alpha). \tag{6.5}$$

One obtains

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w} = 0 \quad \rightarrow \quad w = \sum_{k=1}^{N} \alpha_k y_k x_k \\ \frac{\partial \mathcal{L}}{\partial b} = 0 \quad \rightarrow \quad \sum_{k=1}^{N} \alpha_k y_k = 0 \end{cases} \tag{6.6}$$

with resulting classifier

$$y(x) = \text{sign}[\sum_{k=1}^{N} \alpha_k y_k x_k^T x + b]. \tag{6.7}$$

By replacing the expression for $w$ in the Lagrangian one obtains the following Quadratic Programming (QP) problem (Dual Problem) which solves the problem in the Lagrange multipliers

$$\max_{\alpha} \mathcal{Q}(\alpha) = -\frac{1}{2} \sum_{k,l=1}^{N} y_k y_l x_k^T x_l \ \alpha_k \alpha_l + \sum_{k=1}^{N} \alpha_k \tag{6.8}$$

such that

$$\sum_{k=1}^{N} \alpha_k y_k = 0, \ \ \alpha_k \geq 0 \tag{6.9}$$

Note that this problem is solved in $\alpha$, not in $w$. One can prove that the solution to the QP problem is global and unique. The data related to nonzero $\alpha_k$ are called support vectors, in other words these data points contribute to the sum in the classifier model. A drawback is, however, that the QP problem matrix size grows with number of data $N$, e.g. when one has 1,000,000 data points the size of the matrix involved in the QP problem will be 1,000,000 $\times$ 1,000,000 which is too huge for computer memory storage.

### 6.2.3 Linear SVM classifier: non-separable case

For most real-life problems, when taking a linear classifier, not all the data points of the training set will be correctly classified (Fig.6.4) unless the true underlying problem is perfectly linearly separable. However, often the distributions of the two classes will have a large overlap such that misclassifications have to be tolerated.

Therefore one modifies the inequalities into

$$y_k[w^T x_k + b] \geq 1 - \xi_k, \quad k = 1, ..., N \tag{6.10}$$

with slack variables $\xi_k \geq 0$ such that the original inequalities can be violated for certain points if needed (for $\xi_k > 1$ the original inequality is violated for that data point). The optimization problem becomes

$$\min_{w,b,\xi} \mathcal{J}(w, \xi) = \frac{1}{2} w^T w + c \sum_{k=1}^{N} \xi_k \tag{6.11}$$

subject to

$$\begin{cases} y_k[w^T x_k + b] \geq 1 - \xi_k, \quad k = 1, ..., N \\ \xi_k \geq 0, \quad k = 1, ..., N. \end{cases} \tag{6.12}$$

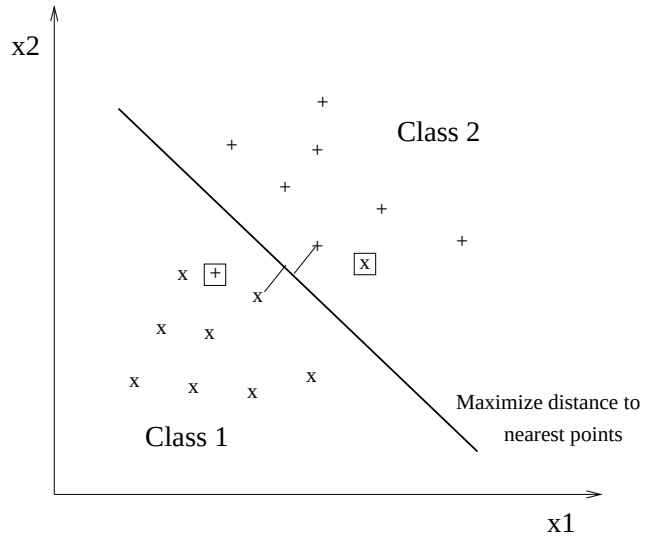



Figure 6.4: *Problem of non-separable data, due to overlapping distributions.*

with Lagrangian

$$\mathcal{L}(w,b,\xi;\alpha,\nu) = \mathcal{J}(w,\xi) - \sum_{k=1}^{N} \alpha_k \{y_k[w^T x_k + b] - 1 + \xi_k\} - \sum_{k=1}^{N} \nu_k \xi_k \tag{6.13}$$

and Lagrange multipliers $\alpha_k \geq 0$, $\nu_k \geq 0$ for $k = 1, ..., N$. The solution is given by saddle point of Lagrangian:

$$\max_{\alpha,\nu} \min_{w,b,\xi} \mathcal{L}(w,b,\xi;\alpha,\nu). \tag{6.14}$$

One obtains

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w} = 0 \;\; \rightarrow \;\; w = \sum_{k=1}^{N} \alpha_k y_k x_k \\ \frac{\partial \mathcal{L}}{\partial b} = 0 \;\; \rightarrow \;\; \sum_{k=1}^{N} \alpha_k y_k = 0 \\ \frac{\partial \mathcal{L}}{\partial \xi_k} = 0 \;\; \rightarrow \;\; 0 \leq \alpha_k \leq c, \;\; k = 1, ..., N \end{cases} \tag{6.15}$$

which gives the quadratic programming problem (dual problem):

$$\max_{\alpha_k} \mathcal{Q}(\alpha) = -\frac{1}{2} \sum_{k,l=1}^{N} y_k y_l \, x_k^T x_l \, \alpha_k \alpha_l + \sum_{k=1}^{N} \alpha_k \tag{6.16}$$

such that

$$\begin{cases} \sum_{k=1}^{N} \alpha_k y_k = 0 \\ 0 \leq \alpha_k \leq c, \ k = 1, ..., N. \end{cases} \tag{6.17}$$

This problem has additional box constraints now. The computation of $b$ follows from the KKT (Karush-Kuhn-Tucker) conditions.

## 6.3 Kernel trick and Mercer condition

Important progress in SVM theory has been made thanks to the fact that the linear theory has been extended to nonlinear models (Vapnik, 1995) [18, 19]. In order to achieve this, one maps the input data into a high dimensional feature space which can be potentially infinite dimensional. A construction of the linear separating hyperplane is done then in this high dimensional feature space[2], after a nonlinear mapping $\varphi(x)$ of the input data to the feature space (Fig.6.5).

Surprisingly, no explicit construction of the nonlinear mapping $\varphi(x)$ is needed. One makes use of the so-called **Mercer theorem** (often called the **kernel trick**). This states that there exists a mapping $\varphi$ and an expansion

$$K(x, y) = \sum_{i=1}^{M} \varphi_i(x)\varphi_i(y), \qquad x, y \in \mathbb{R}^n, \tag{6.18}$$

if and only if, for any $g(x)$ such that

$$\int g(x)^2 dx \qquad \text{is finite} \tag{6.19}$$

one has

$$\int K(x, y)g(x)g(y)dxdy \geq 0. \tag{6.20}$$

Hence the kernel should be positive definite. By applying this theorem one can avoid computations in the huge dimensional feature space. Instead one chooses a kernel function. For RBF kernels $M \to \infty$ while for linear and polynomial kernels $M$ is finite.

[2]In fact it would be better to call this a high-dimensional *hidden layer*, because in pattern recognition one frequently uses the term feature space in another meaning of input space. Nevertheless we will use the term feature space in the sequel.

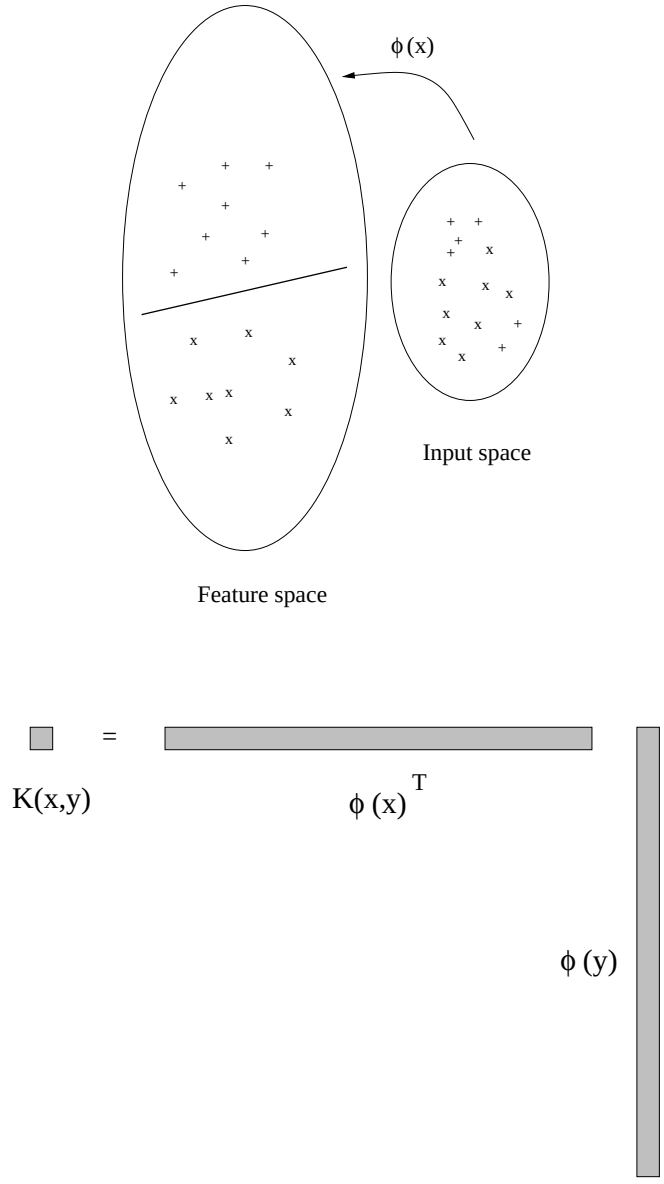



Figure 6.5: *(Top) Mapping of the input space to a high dimensional feature space where a linear separation is made, which corresponds to a nonlinear separation in the original input space; (Bottom) Illustration of using a positive definite kernel function $K$.*

## 6.4 Nonlinear SVM classifiers

The extension from linear SVM classifiers to nonlinear SVM classifiers is straightforward. One starts formulating the problem in the primal space i.e. in the $w$ vector (which could be infinite dimensional). Assume

$$\begin{cases} w^T\varphi(x_k)+b \geq +1 \quad , & \text{if} \quad y_k=+1 \\ w^T\varphi(x_k)+b \leq -1 \quad , & \text{if} \quad y_k=-1 \end{cases} \tag{6.21}$$

which is equivalent now to

$$y_k[w^T\varphi(x_k)+b] \geq 1, \quad k=1,...,N. \tag{6.22}$$

No explicit construction of $\varphi(\cdot) : \mathbb{R}^n \rightarrow \mathbb{R}^{n_h}$ ($n_h$ not specified) is needed at this point. In principle $n_h$ can also be infinite dimensional. The optimization problem becomes

$$\min_{w,b,\xi} \mathcal{J}(w,\xi) = \frac{1}{2}w^Tw + c\sum_{k=1}^{N}\xi_k \tag{6.23}$$

subject to

$$\begin{cases} y_k[w^T\varphi(x_k)+b] \geq 1-\xi_k, \quad k=1,...,N \\ \xi_k \geq 0, \quad k=1,...,N. \end{cases} \tag{6.24}$$

One constructs the Lagrangian:

$$\mathcal{L}(w,b,\xi;\alpha,\nu) = \mathcal{J}(w,\xi_k) - \sum_{k=1}^{N}\alpha_k\{y_k[w^T\varphi(x_k)+b]-1+\xi_k\} - \sum_{k=1}^{N}\nu_k\xi_k \tag{6.25}$$

with Lagrange multipliers $\alpha_k \geq 0$, $\nu_k \geq 0$ ($k=1,...,N$). The solution is given by the saddle point of the Lagrangian:

$$\max_{\alpha,\nu} \min_{w,b,\xi} \mathcal{L}(w,b,\xi;\alpha,\nu). \tag{6.26}$$

One obtains

$$\begin{cases} \frac{\partial\mathcal{L}}{\partial w}=0 \quad \rightarrow \quad w=\sum_{k=1}^{N}\alpha_k y_k\varphi(x_k) \\ \frac{\partial\mathcal{L}}{\partial b}=0 \quad \rightarrow \quad \sum_{k=1}^{N}\alpha_k y_k=0 \\ \frac{\partial\mathcal{L}}{\partial \xi_k}=0 \quad \rightarrow \quad 0\leq\alpha_k\leq c, \ k=1,...,N. \end{cases} \tag{6.27}$$

The quadratic programming problem (dual problem) becomes

$$\max_{\alpha_k} \mathcal{Q}(\alpha) = -\frac{1}{2} \sum_{k,l=1}^{N} y_k y_l \, K(x_k, x_l) \, \alpha_k \alpha_l + \sum_{k=1}^{N} \alpha_k \tag{6.28}$$

such that

$$\begin{cases} \sum_{k=1}^{N} \alpha_k y_k = 0 \\ 0 \leq \alpha_k \leq c, \ k = 1, ..., N. \end{cases} \tag{6.29}$$

Note that $w$ and $\varphi(x_k)$ are never calculated but all calculations are done in the dual space. We make use of the Mercer condition by choosing a kernel

$$K(x_k, x_l) = \varphi(x_k)^T \, \varphi(x_l). \tag{6.30}$$

Finally, the nonlinear SVM classifier takes the form

$$y(x) = \text{sign}[\sum_{k=1}^{N} \alpha_k \, y_k \, K(x, x_k) + b] \tag{6.31}$$

with $\alpha_k$ positive real constants and $b$ a real constant, which follow as solution to the QP problem. Non-zero $\alpha_k$ are called support values and the corresponding data points are called support vectors. The bias term $b$ follows from the KKT conditions.

Several choices are possible for the kernel $K(\cdot, \cdot)$:

$K(x, x_k) = x_k^T x$ (linear SVM)
$K(x, x_k) = (x_k^T x + \eta)^d$ (polynomial SVM of degree $d$), $\eta \geq 0$
$K(x, x_k) = \exp\{-\|x - x_k\|_2^2 / \sigma^2\}$ (RBF kernel)
$K(x, x_k) = \tanh(\kappa \, x_k^T x + \theta)$ (MLP kernel) .

The Mercer condition holds for all $\sigma$ values in the RBF case, but not for all possible choices of $\kappa, \theta$ in the MLP case (therefore the use of an MLP kernel is not popular in SVM methods). In the case of an RBF and MLP kernel, the number of hidden units corresponds to the number of support vectors, e.g. for the RBF kernel one has

$$\begin{aligned} y(x) &= \text{sign}[\sum_{k=1}^{N} \alpha_k y_k \exp\{-\|x - x_k\|_2^2 / \sigma^2\} + b] \\ &= \text{sign}[\sum_{k \in \mathcal{S}_{SV}} \alpha_k y_k \exp\{-\|x - x_k\|_2^2 / \sigma^2\} + b] \end{aligned} \tag{6.32}$$

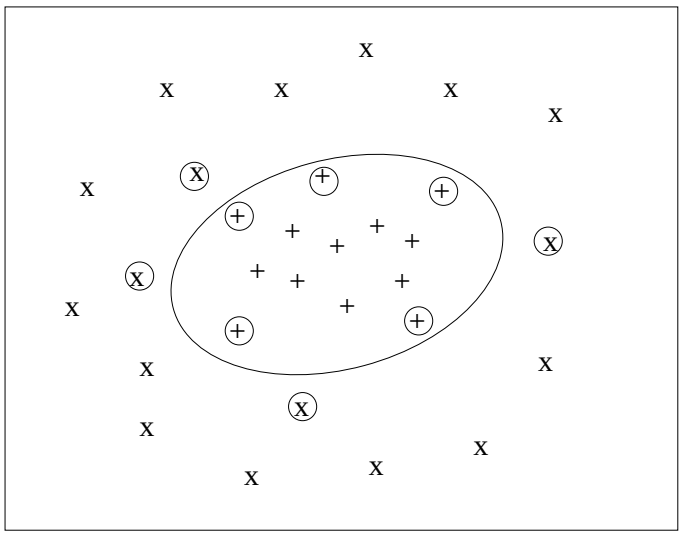


Figure 6.6: *In the abstract figure the encircled points are support vectors. These points have a non-zero support value $\alpha_k$. The decision boundary can be expressed in terms of these support vectors (which explains the terminology). In standard QP type support vector machines all support vectors are located close to the decision boundary.*

where $\mathcal{S}_{SV}$ denotes the set of support vectors. It means that each hidden unit corresponds to a support vector (non-zero support values $\alpha_k$) and the number of hidden units equals the number of support values. The support vectors also have a nice geometrical meaning (Fig.6.6). They are located close to the decision boundary and the decision boundary can be expressed in terms of these support vectors (which explains the terminology).

## 6.5 SVMs for function estimation

### 6.5.1 SVM for linear function estimation

Consider regression in the set of linear functions

$$f(x) = w^T x + b \tag{6.33}$$

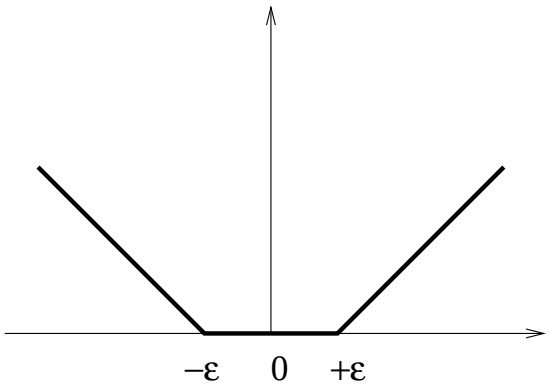



Figure 6.7: *Vapnik $\epsilon$-insensitive loss function for function estimation.*

with $N$ training data $x_k \in \mathbb{R}^m$ and output values $y_k \in \mathbb{R}$. The empirical risk minimization is defined as

$$R_{emp} = \frac{1}{N} \sum_{k=1}^{N} |y_k - w^T x_k - b|_\epsilon. \tag{6.34}$$

For standard SVM function estimation one employs the so-called Vapnik's $\epsilon$-insensitive loss function[3]

$$|y - f(x)|_\epsilon = \begin{cases} 0 & , \text{ if } |y - f(x)| \leq \epsilon \\ |y - f(x)| - \epsilon & , \text{ otherwise} \end{cases} \tag{6.35}$$

shown in Fig.6.7.

By taking such a cost function one can formulate the following optimization problem

$$\min \ \frac{1}{2} w^T w \tag{6.36}$$

subject to $|y_k - w^T x_k - b| \leq \epsilon$ or

$$\begin{cases} y_k - w^T x_k - b & \leq \epsilon \\ w^T x_k + b - y_k & \leq \epsilon. \end{cases} \tag{6.37}$$

[3]SVM theory can be extended to any convex cost function. Historically, the SVM results were first derived for Vapnik's $\epsilon$-insensitive loss function. In general, the choice of a 1-norm in the cost function is more robust than a 2-norm, e.g. with respect to outliers and non-Gaussian noise on the data.

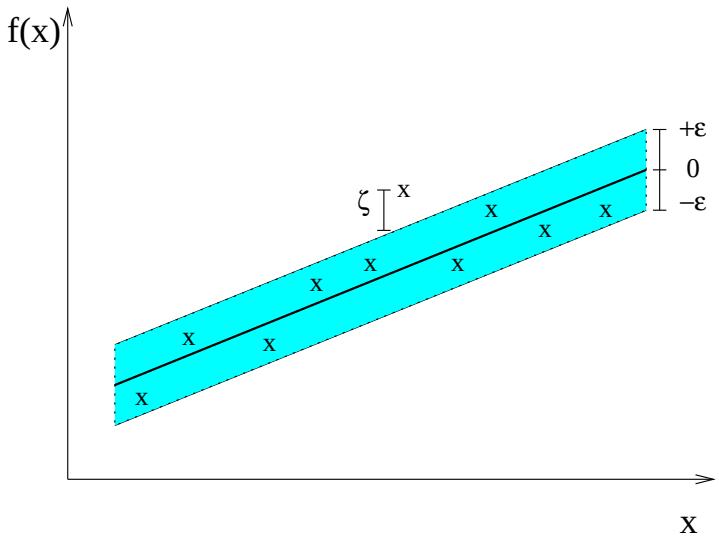



Figure 6.8: *Tube of $\epsilon$-accuracy and points which cannot meet this accuracy, motivating the use of slack variables.*

Here $\epsilon$ denotes the required accuracy as demanded by the user. However, a priori not all points will be able to meet this requirement. Therefore one introduces slack variables

$$\min \ \frac{1}{2}w^T w + c\sum_{k=1}^{N}(\xi_k + \xi_k^*) \tag{6.38}$$

subject to

$$\begin{cases} y_k - w^T x_k - b & \leq & \epsilon + \xi_k \\ w^T x_k + b - y_k & \leq & \epsilon + \xi_k^* \\ \xi_k, \xi_k^* & \geq & 0. \end{cases} \tag{6.39}$$

The constant $c > 0$ determines the trade-off between flatness of $f$ and the amount up to which deviations larger than $\epsilon$ are tolerated, which is illustrated in Fig.6.8.

The Lagrangian is

$$\begin{gathered} \mathcal{L}(w, b, \xi, \xi^*; \alpha, \alpha^*, \eta, \eta^*) = \\ \frac{1}{2}w^T w + c\sum_{k=1}^{N}(\xi_k + \xi_k^*) - \sum_{k=1}^{N}\alpha_k(\epsilon + \xi_k - y_k + w^T x_k + b) \\ -\sum_{k=1}^{N}\alpha_k^*(\epsilon + \xi_k^* + y_k - w^T x_k - b) - \sum_{k=1}^{N}(\eta_k\xi_k + \eta_k^*\xi_k^*) \end{gathered} \tag{6.40}$$

with positive Lagrange multipliers $\alpha_k, \alpha_k^*, \eta_k, \eta_k^* \geq 0$. The saddle point of the Lagrangian is characterized by

$$\max_{\alpha,\alpha^*,\eta,\eta^*} \min_{w,b,\xi,\xi^*} \mathcal{L}(w, b, \xi, \xi^*; \alpha, \alpha^*, \eta, \eta^*) \tag{6.41}$$

with conditions for optimality:

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w} = 0 & \rightarrow & w = \sum_{k=1}^{N} (\alpha_k - \alpha_k^*) x_k \\ \frac{\partial \mathcal{L}}{\partial b} = 0 & \rightarrow & \sum_{k=1}^{N} (\alpha_k - \alpha_k^*) = 0 \\ \frac{\partial \mathcal{L}}{\partial \xi_k} = 0 & \rightarrow & c - \alpha_k - \eta_k = 0 \\ \frac{\partial \mathcal{L}}{\partial \xi_k^*} = 0 & \rightarrow & c - \alpha_k^* - \eta_k^* = 0. \end{cases} \tag{6.42}$$

The dual problem becomes

$$\begin{aligned} \max_{\alpha,\alpha_*} \mathcal{Q}(\alpha, \alpha_*) = & -\frac{1}{2} \sum_{k,l=1}^{N} (\alpha_k - \alpha_k^*)(\alpha_l - \alpha_l^*)\, x_k^T x_l \\ & -\epsilon \sum_{k=1}^{N} (\alpha_k + \alpha_k^*) + \sum_{k=1}^{N} y_k (\alpha_k - \alpha_k^*) \end{aligned} \tag{6.43}$$

subject to

$$\begin{cases} \sum_{k=1}^{N} (\alpha_k - \alpha_k^*) & = & 0 \\ \alpha_k, \alpha_k^* & \in & [0, c]. \end{cases} \tag{6.44}$$

The resulting SVM for linear function estimation is

$$f(x) = w^T x + b \tag{6.45}$$

with $w = \sum_{k=1}^{N} (\alpha_k - \alpha_k^*) x_k$ such that

$$f(x) = \sum_{k=1}^{N} (\alpha_k - \alpha_k^*) x_k^T x + b. \tag{6.46}$$

The support vector expansion is sparse in the sense that many support values will be zero.

### 6.5.2 SVM for nonlinear function estimation

Consider again a nonlinear mapping to the feature space:

$$f(x) = w^T \varphi(x) + b \tag{6.47}$$

with given data $\{x_k, y_k\}_{k=1}^N$. The optimization problem in the primal weight space, which could be infinite dimensional, becomes

$$\min \ \frac{1}{2} w^T w + c \sum_{k=1}^{N} (\xi_k + \xi_k^*) \tag{6.48}$$

subject to

$$\begin{cases} y_k - w^T \varphi(x_k) - b & \leq & \epsilon + \xi_k \\ w^T \varphi(x_k) + b - y_k & \leq & \epsilon + \xi_k^* \\ \xi_k, \xi_k^* & \geq & 0 \end{cases} \tag{6.49}$$

with as a resulting dual problem

$$\begin{array}{c} \max_{\alpha, \alpha_*} \mathcal{Q}(\alpha, \alpha_*) = -\frac{1}{2} \sum_{k,l=1}^{N} (\alpha_k - \alpha_k^*)(\alpha_l - \alpha_l^*) \, K(x_k, x_l) \\ -\epsilon \sum_{k=1}^{N} (\alpha_k + \alpha_k^*) + \sum_{k=1}^{N} y_k (\alpha_k - \alpha_k^*) \end{array} \tag{6.50}$$

subject to

$$\begin{cases} \sum_{k=1}^{N} (\alpha_k - \alpha_k^*) & = & 0 \\ \alpha_k, \alpha_k^* & \in & [0, c]. \end{cases} \tag{6.51}$$

One applies the Mercer condition $K(x_k, x_l) = \varphi(x_k)^T \varphi(x_l)$ which gives

$$f(x) = \sum_{k=1}^{N} (\alpha_k - \alpha_k^*) K(x, x_k) + b. \tag{6.52}$$

# Bibliography


[1] *Bishop C.M., Neural networks for pattern recognition, Oxford University Press, 1995.*

[2] *Cherkassky V., Mulier F., Learning from data: concepts, theory and methods, John Wiley and Sons, 1998.*

[3] *Cristianini N., Shawe-Taylor J., An introduction to support vector machines, Cambridge University Press, 2000*

[4] *Devroye L., Györfi L., Lugosi G., A Probabilistic Theory of Pattern Recognition, NY: Springer, 1996.*

[5] *Duda R.O., Hart P.E., Stork D. G., Pattern Classification (2ed.), Wiley, 2001.*

[6] *Fayyad U.M., Piatetsky-Shapiro G., Smyth P., Uthurasamy R. (Ed.), Advances in Knowledge Discovery and Data Mining, MIT Press, 1996.*

[7] *Fletcher R., Practical methods of optimization, Chichester and New York: John Wiley and Sons, 1987.*

[8] *Hastie T., Tibshirani R., Friedman J., The elements of statistical learning, Springer-Verlag, 2001.*

[9] *Haykin S., Neural Networks: a Comprehensive Foundation, Macmillan College Publishing Company: Englewood Cliffs, 1994.*

[10] *Kohonen T., Self-Organizing Maps, Springer Series in Information Sciences, Vol. 30, 1997.*

[11] *MacKay D.J.C., Information Theory, Inference and Learning Algorithms, book in preparation available at http://wol.ra.phy.cam.ac.uk/mackay/*

[12] *Ripley B.D., Pattern Recognition and Neural Networks, Cambridge: Cambridge University Press, 1996.*

[13] *Ritter H., Martinetz T., Schulten K., Neural Computation and Self-Organizing Maps: An Introduction, Addison-Wesley, Reading, MA, 1992.*

[14] *Schölkopf B., Burges C., Smola A., Advances in Kernel Methods: Support Vector Learning, MIT Press, Cambridge, MA, December 1998.*

[15] *Suykens J.A.K., Vandewalle J., De Moor B., Artificial Neural Networks for Modelling and Control of Non-Linear systems, Kluwer Academic Publishers, Boston, 1996.*

[16] *Suykens J.A.K., Vandewalle J. (Eds.) Nonlinear Modeling: advanced black-box techniques, Kluwer Academic Publishers, Boston, 1998.*

[17] *Suykens J.A.K., Van Gestel T., De Brabanter J., De Moor B., Vandewalle J., Least Squares Support Vector Machines, World Scientific, Singapore, 2002.*

[18] *Vapnik V., The Nature of Statistical Learning Theory, Springer-Verlag, 1995.*

[19] *Vapnik V., Statistical learning theory, John Wiley, New-York, 1998.*

[20] *Weigend A.S., Gershenfeld N.A. (Eds.), Time Series Prediction: Forecasting the Future and Understanding the Past, Addison-Wesley, 1994.*

[21] Barron A.R., "Universal approximation bounds for superposition of a sigmoidal function," *IEEE Transactions on Information Theory*, Vol.39, No.3, pp.930-945, 1993.

[22] Bassett D.E., Eisen M.B., Boguski M.S., "Gene expression informatics - it's all in your mine," *Nature Genetics*, supplement, Vol.21, pp.51-55, Jan 1999.

[23] Brown P., Botstein D., "Exploring the new world of the genome with DNA microarrays," *Nature Genetics*, supplement, Vol.21, pp.33-37, Jan 1999.

[24] Burges C., "A Tutorial on Support Vector Machines for Pattern Recognition," *Knowledge Discovery and Data Mining*, 2(2), 1998.

[25] Chen S., Billings S., Grant P., "Nonlinear system identification using neural networks," *International Journal of Control*, Vol.51, No.6, pp.1191-1214, 1990.

[26] Chen S., Billings S., "Neural networks for nonlinear dynamic system modelling and identification," *International Journal of Control*, Vol.56, No.2, pp.319-346, 1992.

[27] Espinoza M., Suykens J.A.K., Belmans R., De Moor B., "Electric Load Forecasting," *IEEE Control Systems Magazine*, Vol. 27, No. 5, pp. 43-57, Oct. 2007.

[28] Freund Y., Schapire R.E,. "A short introduction to boosting," *Journal of Japanese Society for Artificial Intelligence,* 14(5):771-780, 1999.

[29] Glymour C., Madigan D., Pregibon D., Smyth P., "Statistical inference and data mining," *Communications of the ACM*, Vol.39, No.11, pp.35-41, 1996.

[30] Guyon I., Matic N., Vapnik V., "Discovering informative patterns and data cleaning," in U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds., *Advances in Knowledge Discovery and Data Mining*, pp. 181-203, MIT Press, 1996.

[31] Hornik K., Stinchcombe M., White H., "Multilayer feedforward networks are universal approximators," *Neural Networks*, Vol.2, pp.359-366, 1989.

[32] Jain A., Mao J., Mohiuddin K., "Artificial neural networks: a tutorial," *IEEE Computer*, Vol.29, No.3, pp.31-44, 1996.

[33] Kohonen T., "The self-organizing map," *Proc. IEEE*, Vol.78, No.9, pp.1464-1480, 1990.

[34] Kohonen T., Kaski S., Lagus K., Salojärvi J., Paatero V., Saarela A., "Organization of a massive document collection," *IEEE Transactions on Neural Networks* (special issue on neural networks for data mining and knowledge discovery), Vol.11, No.3, pp. 574-586, 2000.

[35] Lerouge E., Moreau Y., Verrelst H., Vandewalle J., Stoermann C., Gosset P., Burge P., "Detection and management of fraud in UMTS networks", in Proc. of the Third International Conference on The Practical Application of Knowledge Discovery and Data Mining (PADD99), London, UK, Apr. 1999, pp. 127-148.

[36] MacKay D.J.C, "Bayesian interpolation," *Neural Computation*, 4(3): 415-447, 1992.

[37] MacKay D.J.C, "A practical Bayesian framework for backpropagation networks," *Neural Computation*, 4(3): 448-472, 1992.

[38] Møller M.F., "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, Vol.6, pp.525-533, 1993.

[39] Morgan N., Bourlard H., "Continuous speech recognition: an introduction to the hybrid HMM/connectionist approach," *IEEE Signal Processing Magazine*, pp.25-42, May 1995.

[40] Narendra K.S., Parthasarathy K., "Gradient methods for the optimization of dynamical systems containing neural networks," *IEEE Transactions on Neural Networks*, Vol.2, No.2, pp.252-262, 1991.

[41] Poggio T., Girosi F., "Networks for approximation and learning," *Proceedings of the IEEE*, Vol.78, No.9, pp.1481-1497, 1990.

[42] Reed R., "Pruning algorithms - a survey," *IEEE Transactions on Neural Networks*, Vol.4, No.5, pp.740-747, 1993.

[43] Rumelhart D.E., Hinton G.E., Williams R.J., "Learning representations by back-propagating errors," *Nature*, Vol.323, pp.533-536, 1986.

[44] Schölkopf B., Sung K.-K., Burges C., Girosi F., Niyogi P., Poggio T., Vapnik V., "Comparing support vector machines with Gaussian kernels to radial basis function classifiers," *IEEE Transactions on Signal Processing,* Vol.45, No.11, pp.2758-2765, 1997.

[45] Sjöberg J., Zhang Q., Ljung L., Benveniste A., Delyon B., Glorennec P., Hjalmarsson H., Juditsky A., "Nonlinear black-box modeling in system identification: a unified overview," *Automatica*, Vol.31, No.12, pp.1691-1724, 1995.

[46] Smola A., Schölkopf B., "A Tutorial on Support Vector Regression," NeuroCOLT Technical Report NC-TR-98-030, Royal Holloway College, University of London, UK, 1998.

[47] Suykens J.A.K., Vandewalle J., De Moor B., "$NL_q$ theory: checking and imposing stability of recurrent neural networks for nonlinear modelling," *IEEE Transactions on Signal Processing (special issue on neural networks for signal processing)*, Vol.45, No.11, pp. 2682-2691, Nov. 1997.

[48] Suykens J.A.K., Vandewalle J. "Least squares support vector machine classifiers," *Neural Processing Letters*, Vol.9, No.3, pp.293-300, June 1999.

[49] Van Calster B., Timmerman D., Lu C., Suykens J.A.K., Valentin L., Van Holsbeke C., Amant F., Vergote I., Van Huffel S., "Preoperative diagnosis of ovarian tumors using Bayesian kernel-based methods", *Ultrasound in Obstetrics and Gynecology,* vol. 29, no. 5, May 2007, pp. 496-504.

[50] van der Smagt P.P., "Minimisation methods for training feedforward neural networks," *Neural Networks*, Vol.7, No.1, pp.1-11, 1994.

[51] Werbos P., "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, 78 (10), pp.1150-1560, 1990.