Neural Networks in Quantitative Finance

Master Thesis submitted to

Prof. Dr. Wolfgang Härdle

Institute for Statistics and Econometrics

CASE - Center for Applied Statistics and Economics

Humboldt-Universität zu Berlin

by

Enzo Giacomini

(176366)

in partial fulfillment of the requirements

for the degree of **Master of Arts**

Berlin, December 23, 2003

Declaration of Authorship

I hereby confirm that I have authored this master thesis independently and without use of others than the indicated sources. All passages which are literally or in general matter taken out of publications or other sources are marked as such.

Berlin, 23rd december 2003.

Enzo Giacomini

Contents

1 Introduction **4**

2 Neural Networks **5**

2.1 Perceptron 5

2.1.1 Perceptron as a linear discriminant 6

2.1.2 A simple example: OR 8

2.1.3 Error function 10

2.1.4 Delta Rule: learning a classification task 11

2.1.5 Non linear separable sets 12

2.1.6 XOR 12

2.1.7 The step function 14

2.2 Neuron 14

2.2.1 Propagation function 15

2.2.2 Activation function 15

2.3 Neural Networks 17

2.3.1 Neural network as a graph 17

2.3.2 Multi Layer Perceptron networks - MLP 19

2.3.3 Radial Basis Function networks - RBF 19

2.3.4 Representation power of neural networks 20

3 Statistical Learning and Neural Networks **22**

3.1 The learning problem 22

3.1.1 The loss function 22

3.1.2 The risk function 22

3.1.3 Setting the loss function 23

3.2 Empirical Risk Minimization 23

3.3 Neural Networks Learning 24

3.3.1 Error surfaces 25
3.3.2 Descending gradients 26
3.3.3 Descending gradients on neural networks 26
3.3.4 Example in XploRe 28
3.3.5 Backpropagation 28

4 Applications **30**
4.1 Neural Networks in Time Series Forecasting 30
4.1.1 Autoregressive structure 30
4.1.2 Financial time series 31
4.1.3 Exogenous indicators 31
4.1.4 Examples in XploRe 32
4.2 Neural Networks in Volatility Estimation 34
4.2.1 Estimation from Conditional Volatilities 34
4.2.2 Example in XploRe 35
4.2.3 Estimation from Implied Volatilities 36
4.2.4 Examples in XploRe 39

5 Experiment **42**
5.1 Model 42
5.2 Time Series 42
5.2.1 Transformation 42
5.2.2 Time dependency 43
5.3 Networks 43
5.4 Performance Measures 43
5.5 Results and Comments 44

6 Summary and Conclusion **47**

7 References **49**

A Appendix **50**

A.1 descgrad.xpl 50

A.1.1 Example 52

A.2 TSNN.xpl 52

A.2.1 Example 56

A.3 condvolRBF.xpl 56

A.3.1 Example 58

A.4 volsurfRBF.xpl 58

A.4.1 Example 60

A.5 volsurfMLP.xpl 60

A.5.1 Example 62

List of Figures

1 Perceptron with weights w_1, w_2 and threshold θ 6

2 Hyperplane separating sets $\mathcal{C}_1$ from $\mathcal{C}_2$ 7

3 Boolean function OR yields 1 (black) or 0 (white) 8

4 Separating hyperplane $w^\top x - 0.5 = 0$, $w = (1,1)^\top$ 9

5 Separating hyperplane $w^\top x - 0.5 = 0$, $w = (1, \frac{1}{4})^\top$ 10

6 Boolean function XOR yields 1 (black) or 0 (white) 12

7 XOR reproduced by a perceptron acting on the outputs of 2 perceptrons 13

8 Neuron 15

9 Sigmoid function $g(x) = \frac{1}{1+e^{(-cx)}}$ with $c = 1.5$ (red), $c = 3$ (dotted red) and $c = 10$ (black). The step function is the limiting case when $c \to \infty$. 17

10 Feedforward neural network Φ_{NN} 18

11 (2-5-3-1) MLP 20

12 Descending Gradients 28

13 Forecast (red) of exchange rate Japanese Yen / US Dollar (blue), RBF network, 25 hidden units, lag 5 33

14 Forecast (red) of exchange rate German Mark / US Dollar (blue), MLP network, 15 hidden units, lag 5 33

15 Log returns and conditional volatilities from the exchange rate British Pound / US Dollar from 29/04/1997 to 29/04/2002. Estimated with RBF network, 25 hidden units, lag 5. 37

16 Log returns and conditional volatilities from Commerzbank stock from 29/04/1997 to 29/04/2002. Estimated with RBF network, 25 hidden units, lag 2. 38

17 Implied volatility surface estimated using RBF network with 25 hidden units. Parameters: moneyness and maturities. Data: German and Swiss Futures Exchange (EUREX) from 04/01/1999. 40

18 Implied volatility surface estimated using a (2-15-1) MLP. Parameters: strike prices and maturities. Data: German and Swiss Futures Exchange (EUREX) from 04/01/1999. 41

List of Tables

1 OR function 8

2 Output $y(x)$ from a perceptron with $w = (1, 1)^\top$ and $\theta = 0.5$, output $t(x)$ from the OR function. 9

3 Output $y(x)$ from a perceptron with $w = (1, \frac{1}{4})^\top$ and $\theta = 0.5$, output $t(x)$ from the OR function. 10

4 Error function $Q(w)$ 11

5 XOR function 12

6 Output of hidden layer: linear separable 14

7 Time series and sample size 43

8 Performance of networks on JPYUSD 46

9 Performance of networks on DEMUSD 46

10 Performance of networks on BPUSD 47

1 Introduction

Neural networks are a very powerful tool in modern quantitative finance. They are present in the fields related to semi- and non- parametric regression and pattern classification, like time series prediction, risk estimation and credit scoring. However, the mathematical concepts involved on neural networks are considerably complex and some of its theoretical aspects are not yet completely developed. As a consequence, in implementing networks the choice of some parameters and architectures may be made without enough theoretical support. Thus, much of the knowledge on neural networks comes from heuristical approaches from practitioners, developed while applying them on real problems.

The aim of this essay is to overview the neural networks and some of its quantitative finance applications. This wide approach should facilitate the comprehension of basic concepts and structures related to its implementation in the financial environment.

The work is divided into three parts. The first section introduces the basic elements that constitutes the neural networks, the neuron, and its characteristics. The second part presents some applications from neural networks in finance related to semi- and non- parametric regression: time series forecasting, estimation from conditional volatilities and estimation from implied volatility surfaces. The third part is a small experiment where MLP and RBF networks with different architectures are used to forecast three exchange rate time series. The results of each prediction are evaluated and compared according to the networks and architectures used.

All sections contain graphics and practical examples from neural networks are used on real data. All results and illustrations are generated using the libraries and codes from XploRe, which are gathered at the appendix. The bibliography contains classical references on neural networks, where deeper computational and mathematical treatment can be found.

2 Neural Networks

The perceptron is a model of a simple learning machine that stays at the basis of the works in neural networks. It was developed by Rosenblatt in 1962 to solve pattern recognition problems (separate data into two different categories), in an attempt to model mathematically a neurophysiologic element.

2.1 Perceptron

The biological neurons emit an electrical pulse if the concentration of some chemical substance on its environment reaches a certain level. If this situation happens, the neurons are said to be in *excited* state and a *synapse* (the emission of an electrical sign) takes place. Such behaviour of biological neurons is mathematically simulated through a *perceptron*.

The perceptron is an element that weights and sums up inputs (the mathematical analogy to capturing the features of the chemical environment) and compares the result with a predefined threshold value θ (the concentration level of the chemical substance at which the biological neuron is excited). If the weighted sum of inputs is greater than the threshold value, the perceptron releases 1 as output, otherwise 0 (the analogous to emitting a synapsis if the concentration level is reached).

Formulating more precisely, a perceptron y is a function transforming a n-dimensional input into a binary output, $y : \mathbb{R}^n \longrightarrow \{0, 1\}$. The components of the input $x = (x_1, ..., x_n)^\top \in \mathbb{R}^n$ are weighted with $w = (w_1, ..., w_n)^\top \in \mathbb{R}^n$ and summed up

$$f(x) = \sum_{i=1}^{n} w_i x_i = w^\top x \tag{1}$$

The sum is compared with the threshold value θ. If it is greater than it, the perceptron is excited, emitting a pulse. This is represented through the step function

$$g(x) = \begin{cases} 1 & \text{if } a \geq \theta, \\ 0 & \text{if } a < \theta \end{cases}$$

Writting it with indicator function we get

$$g(a) = 1_{[\theta,\infty)}(a) \tag{2}$$

Thus, a perceptron is a composed function $y(x) = g(f(x))$, whose output is expressed by

$$y(x) = 1_{[\theta,\infty)}(w^\top x) \tag{3}$$

Alternatively, one can incorporate the threshold value θ on the notation, defining new vectors $x = (x_0, ..., x_n)^\top \in \mathbb{R}^{n+1}$, with $x_0 \equiv -1$ and $w = (w_0, ..., w_n)^\top \in \mathbb{R}^{n+1}$, with $w_0 \equiv \theta$ and write the output of a perceptron as

$$y(x) = 1_{[0,\infty)}(w^\top x) \tag{4}$$

A perceptron with input $x = (-1, x_1, x_2)^\top$ and weights $w = (\theta, w_1, w_2)^\top$ is graphically represented on figure 1.

Perceptron

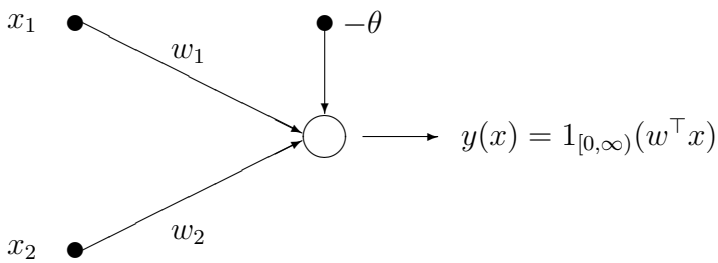


Figure 1: Perceptron with weights w_1, w_2 and threshold θ

2.1.1 Perceptron as a linear discriminant

Beyond reproducing the behavior of its neurological counterparts, a perceptron may work as a linear discriminant (see Bishop (1995)). That means, perceptrons can separate (classify) elements that belong to different sets, given that these sets are *linear separable.*

Two sets $\mathcal{C}_1$ and $\mathcal{C}_2$ on a n-dimensional space are *linearly separable* if there exist $n+1$ real numbers $w_1, ..., w_{n+1}$ such that, for any element $(x_1, ..., x_n) \in \mathcal{C}_1$

$$(\sum_{i=1}^{n} w_i x_i \geq w_{n+1})$$

and for any element $(x_1, ..., x_n) \in \mathcal{C}_2$

$$(\sum_{i=1}^{n} w_i x_i < w_{n+1})$$

These sets are separated by a hyperplane given by

$$(\sum_{i=1}^{n} w_i x_i - w_{n+1}) = 0 \tag{5}$$

A function $f : \mathbb{R}^n \longrightarrow \{0, 1\}$ is linearly separable if the set of elements for which $f(x_1, ..., x_n) = 1$ is linearly separable from the set of elements for which $f(x_1, ..., x_n) = 0$.

For $w_{n+1} = \theta$, the hyperplane separating both sets becomes

$$\sum_{i=1}^{n} w_i x_i - \theta = 0$$

or in vector notation

$$w^\top x - \theta = 0 \tag{6}$$

Separating Hyperplane

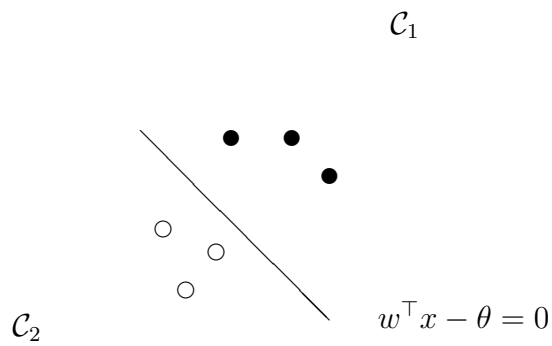


Figure 2: Hyperplane separating sets $\mathcal{C}_1$ from $\mathcal{C}_2$

Given two linear separable sets $\mathcal{C}_1$ and $\mathcal{C}_2$, $(\mathcal{C}_1, \mathcal{C}_2 \in \mathbb{R}^n)$, the hyperplane described on equation (6) is determined by a perceptron with weights $w = (w_1, ..., w_n)^\top$ and bias θ. Thus, a perceptron works as a *classifier*, dividing

x	$t(x)$
(0,0)	0
(0,1)	1
(1,0)	1
(1,1)	1

Table 1: OR function

the input space into two sub-spaces and classifying the input variables as belonging to either one or another of the sub-spaces. For sets in $\mathbb{R}^2$ follows (see figure 2) that $\{\forall x \in \mathbb{R}^2 \mid w^\top x - \theta \geq 0\}$,

$$y(x) = g(f(x; w, \theta)) = 1 \Rightarrow x \in \mathcal{C}_1$$

and $\{\forall x \in \mathbb{R}^2 \mid w^\top x - \theta < 0\}$,

$$y(x) = g(f(x; w, \theta)) = 0 \Rightarrow x \in \mathcal{C}_2$$

Note that the vector of weights w and the bias θ controls the position of the separating hyperplane.

2.1.2 A simple example: OR

The boolean function OR is a mapping $t : \{0, 1\}^2 \to \{0, 1\}$ and can take the values as in table 1. Thus, the function OR can be considered as classifying the inputs into 0 or 1 (see figure 3). We note that the function OR is a linearly separable function.

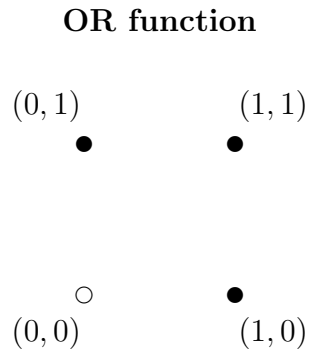


Figure 3: Boolean function OR yields 1 (black) or 0 (white)

x	$w^\top x$	$y(x)$	$t(x)$
$(0,0)$	0	0	0
$(0,1)$	1	1	1
$(1,0)$	1	1	1
$(1,1)$	2	1	1

Table 2: Output $y(x)$ from a perceptron with $w = (1,1)^\top$ and $\theta = 0.5$, output $t(x)$ from the OR function.

A perceptron can mimic such function as it correctly sets one possible separating hyperplane and classifies the inputs into the corresponding outputs. As mentioned before, this depends on the parameters (weights and biases) used, given that they determine the position of the separating hyperplane. With parameters $\theta = 0.5$ and $w = (1,1)^\top$, a perceptron sets the separating hyperplane

$$(1,1)^\top x - 0.5 = 0$$

and yields as output

$$y(x) = 1_{[0.5,\infty)}(w^\top x)$$

i.e., the same as $t(x)$, the output of the OR function, as seen on the table 2.

Separating Hyperplane

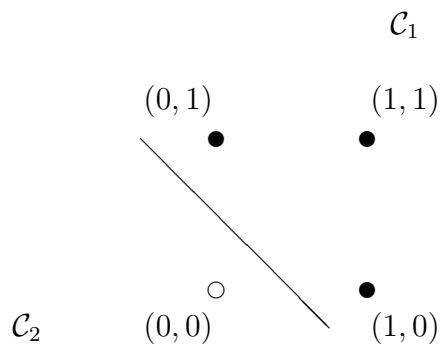


Figure 4: Separating hyperplane $w^\top x - 0.5 = 0$, $w = (1,1)^\top$

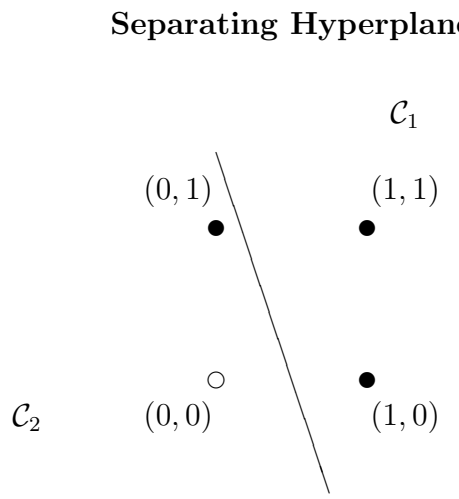


Figure 5: Separating hyperplane $w^\top x - 0.5 = 0$, $w = (1, \frac{1}{4})^\top$

x	$w^\top x$	$y(x)$	$t(x)$
$(0,0)$	0	0	0
$(0,1)$	0.25	**0**	**1**
$(1,0)$	1	1	1
$(1,1)$	1.25	1	1

Table 3: Output $y(x)$ from a perceptron with $w = (1, \frac{1}{4})^\top$ and $\theta = 0.5$, output $t(x)$ from the OR function.

2.1.3 Error function

In order to evaluate how good the classification obtained by a perceptron is against the correct classification, the outputs from the perceptron and from the function $t(x)$ must be compared. For this purpose, an error function counting the number of inputs misclassified by the perceptron is defined as

$$Q(w) = \sum_{i=1}^{k} e(x_i) \tag{7}$$

where

$$e(x_i) = \begin{cases} 0 & \text{if } y(x_i) - t_i = 0, \\ 1 & \text{if } y(x_i) - t_i \neq 0 \end{cases} \tag{8}$$

The set $T = \{(x_i, t_i)\}, i = 1, ..., k$ contains k elements called *patterns*, each

w	$Q(w)$
$(1,1)$	0
$(1,\frac{1}{4})$	0
$(0,1)$	1
$(-1,-1)$	3

Table 4: Error function $Q(w)$

pattern corresponding to an input and its respective output. The set T is called the *training set*. Thus, the error function is the sum of misclassifications produced by a perceptron with parameters w over all k patterns from T.

Moreover, the error function depends on the parameters of the perceptron, as $y(x) = y(x; w, \theta)$. That is a natural result when one reminds that the position of the separating hyperplane is determined by these parameters.

The change of the weights to $w = (1, \frac{1}{4})^\top$ yields another separating hyperplane, as shown on figure 5, and one misclassification, that means, $Q(w) = 1$, as shown on table 3.

2.1.4 Delta Rule: learning a classification task

The best classification is the one in which $Q(w) = 0$, that means, the perceptron classifies all inputs correctly. To obtain the best classification, the weights w that set $Q(w) = 0$ have to found. There is a procedure, called *delta rule*, under which the correct weights can be iteratively obtained. We can say that through the delta rule a perceptron perfectly *learns* a classification task. The delta rule consists of the following steps:

Starting with any randomly chosen weight values w_0, a training set $T = \{(x_i, t_i)\}, i = 1, ...k$, a learning rate η and defining iteration counter $t = 0$, repeat

- if the element i is correctly classified, do not change weights

$$x_i \in \mathcal{C}_1, w^\top x_i \geq 0 \Rightarrow w_{(t+1)} = w_{(t)}$$

$$x_i \in \mathcal{C}_2, w^\top x_i < 0 \Rightarrow w_{(t+1)} = w_{(t)}$$

- if the element i is misclassified, change weights according to

$$x_i \in \mathcal{C}_1, w^\top x_i < 0 \Rightarrow w_{(t+1)} = w_{(t)} + \eta_{(t)} x_i$$

x	$t(x)$
0,0	0
0,1	1
1,0	1
1,1	0

Table 5: XOR function

$$x_i \in \mathcal{C}_2, w^\top x_i \geq 0 \Rightarrow w_{(t+1)} = x_{i(t)} - \eta_{(t)} x_i$$

until all i elements from the training set are correct classified. (The convergence of the delta rule is proved in Haykin (1999)).

2.1.5 Non linear separable sets

There are sets that are not linearly separable, a single perceptron is not able to perfectly classify its elements. To obtain the best possible classification, one approach is to introduce more perceptrons in another layers, called *hidden layers*.

2.1.6 XOR

A basic example of a set that is not linear separable is the set formed by the (not linearly separable) function XOR, a mapping $t : \{0,1\}^2 \to \{0,1\}$ (see table 5 and figure 6).

XOR function

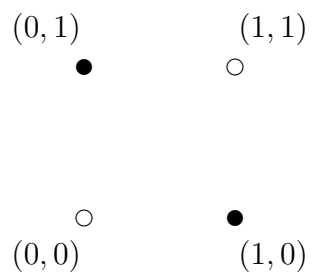


Figure 6: Boolean function XOR yields 1 (black) or 0 (white)

From the figure 6 one can realize that there is no separating hyperplane dividing the space into sets containing the elements as outputs from the XOR. Thus, a single perceptron cannot correctly classify all inputs. A better classification is attained through the use of more perceptrons, introduced in a hidden layer. The new perceptrons *transform* the input space in a linear separable one and releasing to the perceptron in the subsequent layerlinearly separable inputs. One of the architectures that reproducethe XOR function is on figure 7.

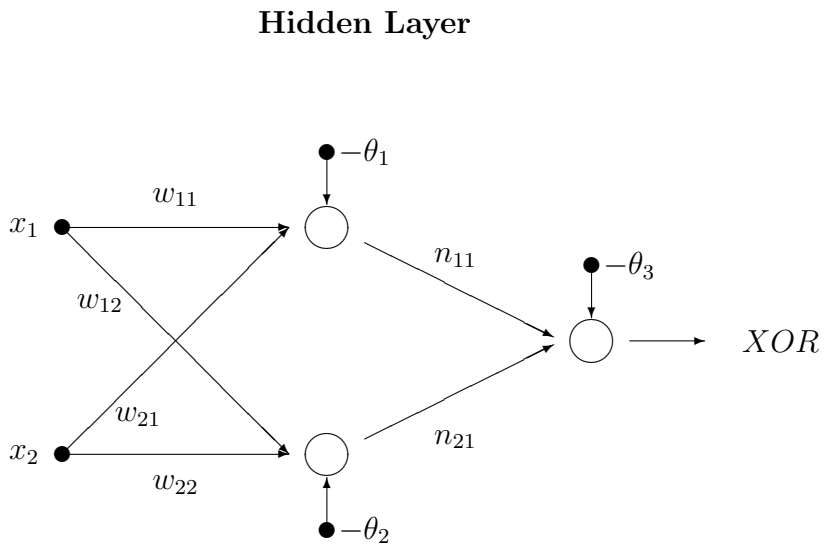


Figure 7: XOR reproduced by a perceptron acting on the outputs of 2 perceptrons

The values of the parameters for the hidden layer perceptrons p_1 and p_2 are:

$$w_{1,1} = 1, w_{1,2} = -1, \theta_1 = 0.5$$

$$w_{2,1} = -1, w_{2,2} = 1, \theta_2 = 0.5$$

The values of the parameters for the output perceptron p_3 are:

$$n_{1,1} = 1, n_{2,1} = 1, \theta_3 = 1$$

The inputs and outputs yielded with the parameters above are on the table 6.

x	p_1	p_2	p_3
$(0,0)$	$(0$	$,\ 0)$	0
$(0,1)$	$(0$	$,\ 1)$	1
$(1,0)$	$(1$	$,\ 0)$	1
$(1,1)$	$(0$	$,\ 0)$	0

Table 6: Output of hidden layer: linear separable

2.1.7 The step function

The step function is not continuous and consequently not differentiable over its whole domain. This fact represents a disadvantage in practical implementations, where calculations that require the derivatives of the outputs related to the inputs of a perceptron are needed. To overcome this hurdle, another functions are used, specially the *sigmoid function*. The sigmoid function is continuous, approximates the step function very well and is differentiable on the whole domain. Generally, any function can be used on the place of the step function and in this situation one do not talk about perceptrons anymore but about *neurons*.

2.2 Neuron

The basic element of a neural network is the *neuron*, its building block. A neuron is represented by a mapping $y : \mathbb{R}^k \longrightarrow \mathbb{R}$ transforming a k dimensional input into a real number. The neuron y is a function composed of

- a propagation function $f : \mathbb{R}^k \to \mathbb{R}$
- an activation function $g : \mathbb{R} \to [0,1]$ where $g(x)$ has the output of $f(x)$ as argument.

Thus, a neuron has the general form

$$y(x) = g(f(x; w)) \tag{9}$$

The output of $g(x) \in [0,1]$ is bounded and can be considered as a standardization of any bounded output in $\mathbb{R}$. If $f(x)$ is a polynomial, its degree is called the order of the neuron and its coefficients can be considered the parameters of the neuron.

Neuron

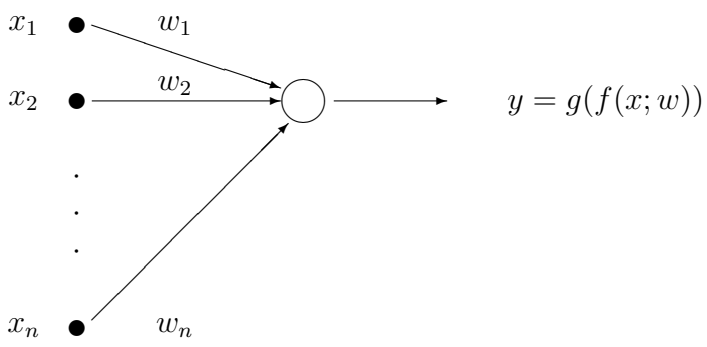


Figure 8: Neuron

2.2.1 Propagation function

A common neuron has as propagation function the weighted sum of inputs as in equation (1)

$$f(x) = \sum_{i=1}^{k} w_i x_i - \theta$$

2.2.2 Activation function

The activation function g of a neuron may assume many forms. For example, g can be

- a linear function:
$$g(x) = ax + b \tag{10}$$
- the identity function:
$$g(x) = x \tag{11}$$

If g is a binary function, i.e., has the from $g : \mathbb{R}^n \to \{a, b\}, a, b \in \mathbb{R}$, the neuron is called a *binary* neuron. Below two examples of activation functions from binary neurons:

- the binary step function

$$g(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0 \end{cases} \tag{12}$$

- the bipolar step function

$$g(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0 \end{cases} \tag{13}$$

Generally, g can be non linear from the form $g : \mathbb{R}^n \to \mathbb{R}$. Specially interesting are the *sigmoidal* functions from which the step functions described above are a limiting case:

- the standard sigmoid function

$$g(x) = \frac{1}{1 + e^{(-cx)}} \tag{14}$$

  as $c \to \infty$, the sigmoid function converges to the binary step function.

- the hyperbolic tangent function:

$$\begin{aligned} g(x) &= \tanh\left(\frac{cx}{2}\right) \\ &= \frac{1 - e^{(-cx)}}{1 + e^{(-cx)}} \end{aligned} \tag{15}$$

  as $c \to \infty$, the hyperbolic tangent function converges to the bipolar step function.

Those functions are, in opposite to the step function, continuous and have the important characteristic that

$$g'(x) = g(x) - g(x)^2 \tag{16}$$

This property is very useful for practical calculation of neural networks, as will be seen in subsequent sections.

Neurons are the basic units that organized in a certain way give origin to the *neural networks*. Neural networks representation power is much higher than of the single neurons and they are the subject of the next section.

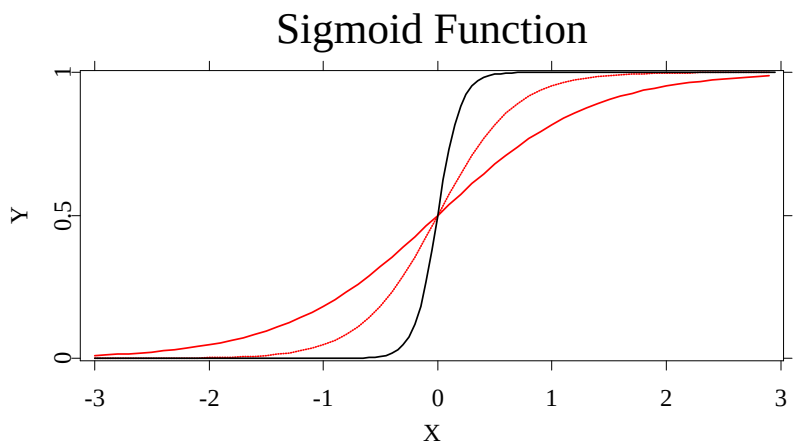


Figure 9: Sigmoid function $g(x) = \frac{1}{1+e^{(-cx)}}$ with $c = 1.5$ (red), $c = 3$ (dotted red) and $c = 10$ (black). The step function is the limiting case when $c \to \infty$.

2.3 Neural Networks

Artificial neural networks are mathematical representations of biological neural structures. They produce the mapping $\Phi_{NN} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and can be written as

$$(y_1, ..., y_m) = \Phi_{NN}(x_1, ...x_n) \tag{17}$$

2.3.1 Neural network as a graph

A neural network is composed of interconnected neurons disposed according to a certain architecture. It can be suitable represented as a graph, where the neurons are represented as nodes. Associated to each arc connecting the nodes, there is a

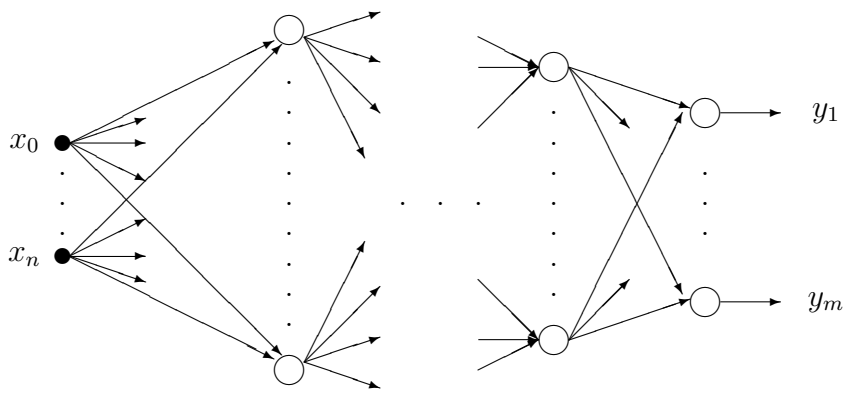
value representing the weighting parameters (polynomial coefficients of the propagation function) of each neuron. Besides that,

- to each node are associated two values: the dimension of the inputs, *in* and the dimension of the outputs, *out*.
- the set of nodes with $in = 0$ is called input layer. This set is non empty, i.e., there exists at least one node with $in = 0$

- the set of nodes with $out = 0$ is called output layer. This set is non empty, i.e., there exists at least on node with $out = 0$
- the level l of a node is the longest length from an input node to that node.
- the depth d of a neural network is the length of the longest path from the input nodes to the output nodes.
- the set of nodes with level l that do not belong to input or output layers (i.e., $0 < l < d$) is called l-hidden layer
- the graph is an acyclic directed graph, i.e, outputs of nodes do not serve as input to nodes with lower level. In this case the network is called *feedforward network*.

A network Φ_{NN} where the threshold values are incorporated in the input vector $x = (x_0, ..., x_n)^\top \in \mathbb{R}^{n+1}, x_0 \equiv -1$ and the output vector is $y = (y_1, ..., y_m)^\top \in \mathbb{R}^m$ is represented on figure 10.

Neural Network



input layer 1.hidden layer d-1.hidden layer output layer

Figure 10: Feedforward neural network Φ_{NN}

2.3.2 Multi Layer Perceptron networks - MLP

Neural networks where the hidden neurons have sigmoidal activation function and the output neurons the sigmoidal or identity function are called *Multi Layer Perceptrons* (MLP). Networks $\Phi_{MLP} : \mathbb{R}^n \to \mathbb{R}^m$

- composed only of input layer and output layer (without hidden layer) are called single layer perceptrons,
- with j_l units in each l-hidden layer, $l = 1, ..., d-1$ are denominated $(n - j_1 - ... - j_l - ... - m)$ MLP.

Each p-component of $y = (y_1, ..., y_m)$ is released by the m-neuron at the output layer as a function of the input $x = (x_1, ..., x_n)$ and of the parameters w. Writing in compact from, with weights on the input vectors

$$y_p = g_p \left(\sum_{i=0}^{j_{d-1}} w_i \left(g_i \left(\sum_{u=0}^{j_{d-2}} w_u ... \sum_{t=0}^{j_1} w_t \left(g_t \left(\sum_{k=0}^{n} w_k x_k \right) \right) ... \right) \right) \right) \tag{18}$$

Figure 11 shows the graph of a neural network Φ_{MLP}, where $d = 3$, $n = 2$, $j_1 = 5$, $j_2 = 3$ and $m = 1$ or $(2 - 5 - 3 - 1)$ MLP.

2.3.3 Radial Basis Function networks - RBF

Radial Basis Function (RBF) neurons are neurons where

- the propagation function has the form

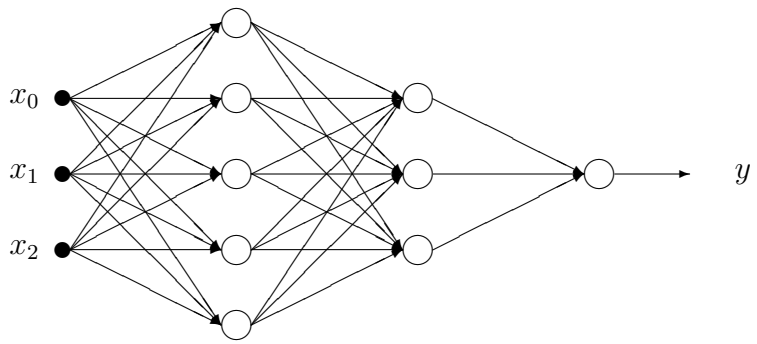
$$f(x) = \| x - w \| \tag{19}$$

  where $x = (x_1, ..., x_n)^\top, w_r = (w_1, ..., w_n)^\top \in \mathbb{R}^n$ are the inputs and weights.

- the activation function $h(x)$ has the form of a radial symmetric function, commonly the gaussian function (but also the generalized inverse multi-quadric function, the thin-plate spline function, the generalized multi-quadric function)

Networks with

Multi Layer Perceptron



$$y = g\left(\sum_{j=1}^{3} w_j g_j \left(\sum_{l=1}^{5} w_{lj} g_l \left(\sum_{i=0}^{2} w_{il} x_i\right) + w_{0l}\right) + w_{0j}\right)$$

Figure 11: (2-5-3-1) MLP

- one hidden layer containing r RBF neurons
- output neurons with propagation function $f(x) = \sum_{i=1}^{r} w_{ij} x_i$ and identity activation function $g(x) = x$

are called RBF networks $\Phi_{RBF} : \mathbb{R}^n \to \mathbb{R}^m$ with r RBF neurons on the hidden layer. Each j-component of the output $y = (y_1, ..., y_m)$ is given by

$$y_j(x) = \sum_{i=1}^{r} w_{ij} h_i(\| x - w_i \|) \tag{20}$$

In each neuron the propagation function calculates how close (using in most cases the euclidian distance, as above) the input vector x is to the vector w_r. The gaussian activation function releases then higher values for input vectors that are close to the vector w_r and small values for inputs that are far away from it. Thus the weights form *clusters* in the input space.

2.3.4 Representation power of neural networks

MLP and RBF networks have a huge representation power. The higher the number of neurons in a single hidden layer, the higher the complexity of the

represented function. It is proved that a neural network with one hidden layer and enough number of neurons can replicate *any* function. For practical purposes, however, networks with neurons distributed in more than one hidden layer are preferred. When the function to be replicated is unknown and just its realizations are available, the neural networks can be used to extract such relation form the realizations. The next section is dedicated to find out the best network that replicates or at least approximate an unknown function based on its realizations.

3 Statistical Learning and Neural Networks

The main purpose of the statistical learning is to build a *model* representing an unknown *process* based on a data set containing empirical realizations of the process. The model should extract as much information as possible from the data and have good generalization properties, i.e., make good predictions of future realizations of the process. This section follows the ideas developed in Vapnik (1995) and Vidyasagar (1997).

3.1 The learning problem

Given a set of independent variables x following an identical unknown probability distribution function $F(x)$, an unknown process H over the input data x produces the output $y = H(x)$ distributed as $F(y|x)$. A model M, also called machine, has to approximate the process using parameters θ through $M = f(x;\theta)$.

3.1.1 The loss function

In order to find out the best approximation of H, a measure of its quality is necessary, i.e, a function that expresses how god the performance of the model is at each x. Consequently, this function has as arguments the true output of the process $y = H(x)$, the output of the approximation function $f(x;\theta)$ and measures how distant both values are from each other. This function is called the *loss function* $L(y, f(x;\theta))$.

3.1.2 The risk function

To get a measure of performance over all x, the loss function is summed over all possible values of x and y from the unknown distribution $F(x, y)$ in order to express the *expected loss*, also called *risk function* $R(\theta)$

$$R(\theta) = \int_{\mathcal{S}} L(y, f(x;\theta))dF(x, y) \tag{21}$$

To get a good approximation of H, the risk function must be minimized. Thus, the problem of statistical learning is a *risk minimization* problem, where $f(x;\theta)$ is to be determined and the distribution $F(x, y)$ is unknown.

3.1.3 Setting the loss function

The choice of an appropriate loss function $L(y, f(x;\theta))$ determines which kind of problem is represented.

- if the loss function is the indicator function, as

$$L(y, f(x;\theta)) = \begin{cases} 0 & \text{if } y = f(x;\theta), \\ 1 & \text{if } y \neq f(x;\theta) \end{cases} \tag{22}$$

  it indicates whether the model M correctly classifies the inputs x into outputs y. Then the model M can be interpreted as a classifier. The risk minimization implies that minimum probability of misclassification is achieved.

  Note that this was the error function used to evaluate the performance of a single perceptron over a data set.

- if the loss function takes the form

$$L(y, f(x;\theta)) = (y - f(x;\theta))^2 \tag{23}$$

  the minimization of the risk function becomes a least square regression estimation.

- if the loss function is a density function and

$$L(y, p(x;\theta)) = -ln(p(x,\theta)) \tag{24}$$

  the minimization of the risk function becomes a density estimation.

3.2 Empirical Risk Minimization

In practical applications, the distribution $F(x, y)$ is unknown and the risk function cannot be evaluated. The risk function is then substituted through an empirical function based on a given data set, the set of inputs and outputs $T = ((x_1, y_1), ..., (x_k, y_k))$ called, as seen before, *training set*. The empirical risk is defined as

$$R(\theta)_{emp} = \frac{1}{k} \sum_{i=1}^{k} L(y_i, f(x_i;\theta)) \tag{25}$$

and the learning process is thus based on *empirical risk minimization*, ERM, for a set of functions $L \in \mathcal{L}$:

$$\min_{\theta} \frac{1}{k} \sum_{i=1}^{k} L(y_i, f(x_i; \theta)) \tag{26}$$

Using the loss functions $L(y, f(x; \theta)) = (y_i - f(x_i; \theta))^2$, the risk function becomes

$$R(\theta)_{emp} = \frac{1}{k} \sum_{i=1}^{k} (y_i - f(x_i; \theta))^2 \tag{27}$$

and the ERM is then equivalent to the least-squares method.

Using the loss functions $L(y, p(x; \theta)) = -ln(p(x_i, \theta))$, the risk function becomes

$$R(\theta)_{emp} = \frac{1}{k} \sum_{i=1}^{k} -ln(p(x_i, \theta)) \tag{28}$$

and the ERM is then equivalent to the maximum likelihood method.

In this general framework, *neural networks* represent machines that are able to learn a process based on empirical data through empirical risk minimization. There are many possible neural networks structures, each structure representing functions $M = f(x; \theta)$ used in the learning process. In addition, the defined loss function determines the purpose of the neural network.

3.3 Neural Networks Learning

Neural networks represent the mapping $\Phi_{NN} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, transforming inputs $x \in \mathbb{R}^n$ into outputs $y \in \mathbb{R}^m$. Furthermore, a set of parameters w determined by the architecture of the network influences the mapping. Thus, a general neural network can be written as

$$(y_1, ..., y_m) = \Phi_{NN}(x_1, ...x_n; w) \tag{29}$$

One of the main purposes of a neural network is to reproduce or approximate an *unknown* functional relation $\alpha : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ between the variables $x \in \mathbb{R}^n$ and $t \in \mathbb{R}^m$ where a training set $T = \{(x_1, t_1), ..., (x_k, t_k)\}$ with k empirical realizations of such relation is known. Thus, for $i = 1, ..., k$ the target t_i is supposed to have the form

$$t_i = \alpha(x_i) \tag{30}$$

A neural network Φ_{NN} with inputs x_i and parameters w yields

$$y_i = \Phi_{NN}(x_i; w) \tag{31}$$

and the output of the network differs from the target output as

$$\Phi_{NN}(x_i; w) = t_i + \varepsilon_i \tag{32}$$

The framework is the same as in the preceding section: as a statistical learning machine, the neural network is supposed to extract as much information as possible from the training set and generalize well for future realizations of the unknown process α.

The extraction of information is made through empirical risk minimization, i.e, the parameters w of the network have to be chosen in order to minimize the sum of the losses over all *patterns* of the training set

$$\min_w R(w)_{emp} = \frac{1}{k} \sum_{i=1}^{k} L(t_i, y_i) \tag{33}$$

Once the loss function $L(t_i, y_i)$ is defined, the minimizing parameters can be found. That is equivalent to find a global minimum of $R(w)$ on the w-space.

3.3.1 Error surfaces

Using as loss function the squared difference between the target and the output, $L(t_i, y_i) = (t_i - y_i)^2$, the sum of the loss functions over all patterns becomes the sum of squared errors (its characteristics remains the same if multiplied by a constant)

$$SSE = \frac{1}{2} \sum_{i=1}^{k} \sum_{j} (t_{j,i} - y_{j,i})^2 = \frac{1}{2} \sum_{i=1}^{k} (\mathbf{t}_i - \mathbf{y}_i)^2 \tag{34}$$

and the empirical risk function becomes the mean squared error (MSE), a function of the parameters w of the network:

$$R(w) = MSE = \frac{1}{k} SSE \tag{35}$$

The function $R(w)$ can be viewed as a surface on the w-space and is called *error surface.*

3.3.2 Descending gradients

The minimization problem $\min_w R(w)$ consists in finding the weights of the neural network under which the error surface reaches a global minimum.

Among the numerical methods used to solve minimization problems, the method of descendent gradients shows suitable properties. Starting from a initial w_{in} with corresponding $R(w_{in})$, the parameters w are corrected at each iteration in discrete steps, in the opposite direction of the gradient of $R(w)$. When the gradient of $R(w)$ is zero, the parameters w are not modified anymore and a minimum of $R(w)$ is reached.

The gradient of the risk function, $\nabla R(w)$ must then be determined at each iteration. The change Δw in the parameter w is proportional to the negative of the gradient and to the learning rate $0 < \eta < 1$ - a factor that gives the magnitude of the change on the parameters and influences the convergence of the method. Thus, at each step the parameter w is modified according to

$$w_{n+1} = w_n + \Delta w \tag{36}$$

where

$$\Delta w = -\eta \nabla (R(w)) \tag{37}$$

The iteration stops when the gradient is zero or when the cost function has a smaller value than a desired precision. After enough iterations, the parameter w may reach a value where the gradient does not change anymore and a minimum of the function $R(w)$ is attained.

However, this minimum can be a local one, the global minimum is not achieved but the weights do not change anymore. In this case, the descending gradient method do not solve the learning problem.

3.3.3 Descending gradients on neural networks

To evaluate the gradient on a neuron one should remind its structure based on the propagation and activation functions. Using as loss function the SSE, the elements of the gradient in a single j-neuron $y_j = g_j(f_j(x))$ are calculated as

$$\frac{\partial R}{\partial w_{ij}} = \frac{\partial R}{\partial y_j} \frac{\partial y_j}{\partial f_j} \frac{\partial f_j}{\partial w_{ij}} \tag{38}$$

where the partial derivative from the cost function with respect to the output

of the j-neuron on the output layer is given by

$$\frac{\partial R}{\partial y_j} = \frac{\partial(\frac{1}{2}\sum(t_j - y_j)^2)}{\partial y_j} = -(t_j - y_j) \tag{39}$$

the derivative with respect to the propagation function is given by

$$\frac{\partial y_j}{\partial f_j} = \frac{\partial g_j(f_j)}{\partial f_j} = {g_j}'(f_j) \tag{40}$$

and finally the derivative with respect to the weights is given by

$$\frac{\partial f_j}{\partial w_{ij}} = y_i \tag{41}$$

All derivatives together yield

$$\frac{\partial R}{\partial w_{ij}} = -(t_j - y_j)g_j'(f_j)y_i \tag{42}$$

Calling

$$\delta_j = -(t_j - y_j){g_j}'(f_j)$$

we can write

$$\frac{\partial R}{\partial w_{ij}} = \delta_j y_i \tag{43}$$

Note: From the derivations we can verify the delta rule for standard perceptrons. As seen above, it is necessary to take the derivatives of the function g to implement the descending gradients method. Therefore, the propagation function must be differentiable, what is not verified with the step or sign functions, as was the case for perceptrons. Substituting those functions by the sigmoid or hyperbolic tangent functions allows the implementation of the learning process can be implemented.

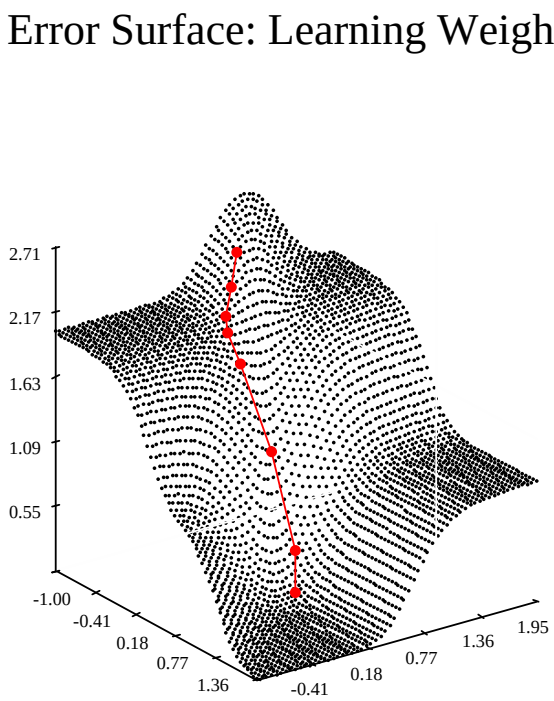


Figure 12: Descending Gradients

3.3.4 Example in XploRe

In the figure 12 a example of error surface and descending gradients method. The example was generated in XploRe with SFMdescgrad.xpl (listed on the appendix).

A single neuron with sigmoid activation function is trained on a set of values. After the error for all patterns (batch) is calculated, the weights are corrected in the opposite direction of the gradient vector. The method stops after a fixed number of iterations.

3.3.5 Backpropagation

A version of the descendent gradients method applied to neural networks is called backpropagation and is based on recursive derivations over the network, where the derivation above is generalized for the weights on the hidden

layers and input layers (i.e., for the whole network).

It is proved that the error from the j-neuron is composed of the sum of errors from the k-neurons that have as input the output of the j-neuron:

$$\delta_j = \sum_k \delta_k w_{jk} \tag{44}$$

Therefore, the change on the weights w_{ij} is

$$\Delta w_{ij} = \eta \delta_j y_i \tag{45}$$

where

$$\delta_j = \begin{cases} -(t_j - y_j) g_j^{'}(f_j) & \text{if } j \text{ is an output neuron,} \\ g_j{}'(f_j) \sum_k \delta_k w_{jk} & \text{if } j \text{ is a hidden neuron} \end{cases} \tag{46}$$

In the backpropagation, the network is feeded with a pattern k from the training set. At each neuron the derivative of the propagation function and the output of the neuron are saved. After the error of the j-neuron is calculated using the equations above the change in the parameters is computed.

The parameters can be corrected according to the sum of errors of all patterns (*batch* backpropagation) or at each step, according to the error from each pattern (*online* backpropagation). In both methods the convergence is strongly dependent on the choice of the initial weights. In most cases, however, the parameters that guarantee convergence are not known and its choice is random.

To see more on learning process, descending gradients and backpropagation, refer to Duda, Hart and Stork (2000) and Haykin (1999).

4 Applications

4.1 Neural Networks in Time Series Forecasting

Many time series contain an unknown autoregressive structure, where its future realizations depend on its past realizations. A neural network can be used to extract an estimation of the autoregressive structure hidden in realizations of a time series. In such procedure the network learns the characteristics of the dependency through optimization of its parameters (weights), as described in the section 3. Afterwards, the network may be used to forecast future values of the series.

4.1.1 Autoregressive structure

For a time series $\{s_t\}_{t=1}^{N}$, the vector

$$x_t = (s_t, s_{t-1}, ..., s_{t-p+1}) \tag{47}$$

with $t > p - 1$, represents a past window of size p. The vector

$$t_t = (s_{t+k+1}, ..., s_{t+k+f}) \tag{48}$$

with $k = h - 1$, $t \leq N - k - f$ represents a future window of size f displaced h steps ahead of t.

A non linear relationship supposed to exist between both windows of the time series $\{s_t\}_{t=1}^{N}$ has the form $\phi : \mathbb{R}^p \rightarrow \mathbb{R}^f$ and is expressed as

$$(s_{t+k+1}, ..., s_{t+k+f}) = \phi(s_t, s_{t-1}, ..., s_{t-p+1}) \tag{49}$$

Such function can be estimated through a neural network Φ_{NN} with p input neurons and f output neurons. The network must be trained using past windows of the series of size p as input vectors and future windows of size f as target vectors. The training set T can be defined as $T = \{(x_t, t_t)\}$ where $t = p, ..., N - k - f$. After training, the neural network yields

$$(s_{t+k+1}, ..., s_{t+k+f}) = \Phi_{NN}(s_t, s_{t-1}, ..., s_{t-p+1}) + \varepsilon_t \tag{50}$$

As an example, for $k = 1$, $f = 1$ and $p = 3$, the model represents an Non Linear Autoregressive process of order 3

$$s_{t+1} = \Phi_{NN}(s_t, s_{t-1}, s_{t-2}) + \varepsilon_t$$

4.1.2 Financial time series

In forecasting financial time series with high frequency data (daily, hourly), the realizations at time $t+1$ are considered as non linear function of p past realizations, the process is then called a non linear autoregressive process of order p or NLAR(p), see Franke, Härdle and Hafner (2001).

A non linear relationship $\phi : \mathbb{R}^p \to \mathbb{R}$ supposed to exist between both set of values has the form

$$s_{t+1} = \phi(s_t, s_{t-1}, ..., s_{t-p+1}) \tag{51}$$

The autoregressive structure ϕ can be approximated through a neural network $\Phi_{NN} : \mathbb{R}^p \to \mathbb{R}$ with parameters w as

$$s_{t+1} = \Phi_{NN}(s_t, s_{t-1}, ..., s_{t-p+1}; w) + \varepsilon_t \tag{52}$$

If the SSE is used as cost function the network with parameters

$$\hat{w} = \arg\min \frac{1}{N-p} \sum_{t=p}^{N-1} (s_{t+1} - \Phi_{NN}(s_t, s_{t-1}, ..., s_{t-p+1}; w))^2 \tag{53}$$

will best approximate the series in a least square sense.

The prediction of the realization at $t+1$ is then

$$\hat{s}_{t+1} = \Phi_{NN}(s_t, s_{t-1}, ..., s_{t-p+1}; \hat{w}) \tag{54}$$

4.1.3 Exogenous indicators

The realizations of a time series may be explained not only by the dependency on its own past realizations but also by the realizations of another time series representing economic or financial indicators. The process is then called a non linear autoregressive process with exogenous components of order p or NLARX(p) process.

The realizations at time t from the h exogenous time series are represented as $X_t^1, ..., X_t^h$. Defining x_t as

$$x_t = (s_t, s_{t-1}, ..., s_{t-p+1}, X_t^1, ..., X_t^h) \tag{55}$$

a non linear relationship ϕ supposed to exist between x_t and s_{t+1} with the form $\phi : \mathbb{R}^{p+h} \rightarrow \mathbb{R}$ and is expressed as

$$s_{t+1} = \phi(s_t, s_{t-1}, ..., s_{t-p+1}, X_t^1, ..., X_t^h) \tag{56}$$

The autoregressive structure ϕ can be approximated through a neural network $\Phi_{NN} : \mathbb{R}^{p+h} \rightarrow \mathbb{R}$ with parameters w as

$$s_{t+1} = \Phi_{NN}(s_t, s_{t-1}, ..., s_{t-p+1}, X_t^1, ..., X_t^h; w) + \varepsilon_t \tag{57}$$

A training set is then constructed, using as x_t as input vectors and as output realizations s_{t+1}.

If the SSE is used as cost function the network with parameters

$$\hat{w} = \arg\min \frac{1}{N-p} \sum_{t=p}^{N-1} (s_{t+1} - \Phi_{NN}(s_t, s_{t-1}, ..., s_{t-p+1}, X_t^1, ..., X_t^h; w))^2 \tag{58}$$

will best approximate the series in a least squared sense.

The prediction of the realization at $t+1$ is then

$$\hat{s}_{t+1} = \Phi_{NN}(s_t, s_{t-1}, ..., s_{t-p+1}, X_t^1, ..., X_t^h; \hat{w}) \tag{59}$$

4.1.4 Examples in XploRe

Using TSNN.xpl (see appendix), the time series containing the exchange rates Japanese Yen to US Dollar and German Mark to US Dollar are used, after log-differences transformation, to train a RBF network with 25 hidden units and a MLP network with 15 hidden units respectively. The input units are the 5 lagged values $(t, t-1, ..., t-4))$ of the series and the output unit is the value of the series at $t+1$. Thus, as in the representation described in the section 4.1.1, a NLAR(5) process.

After training, the networks are applied on a test set and the forecasting results as well as the original series are plotted on figure 13 (Japanese Yen to US Dollar) and 14 (German Mark to US Dollar).

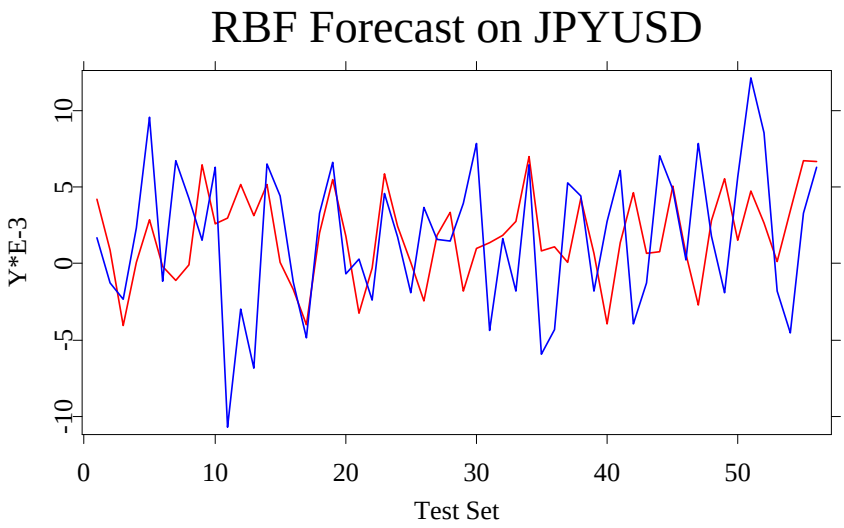


Figure 13: Forecast (red) of exchange rate Japanese Yen / US Dollar (blue), RBF network, 25 hidden units, lag 5

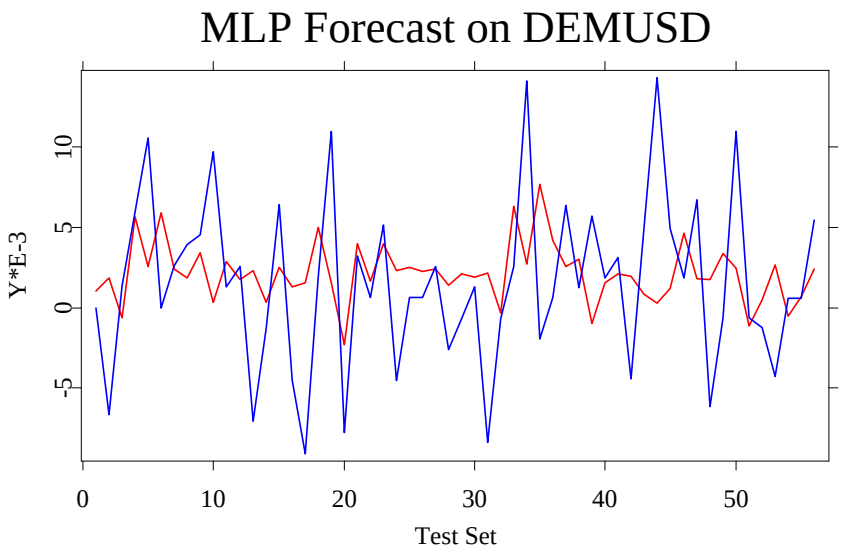


Figure 14: Forecast (red) of exchange rate German Mark / US Dollar (blue), MLP network, 15 hidden units, lag 5

4.2 Neural Networks in Volatility Estimation

4.2.1 Estimation from Conditional Volatilities

Neural networks can be used the estimate the conditional volatility of financial time series. Considering a time series with stochastic volatility following an AR(p)-ARCH(p) process from the form

$$s_{t+1} = \phi(s_t, ...s_{t-p+1}, X_t^1, ..., X_t^h) + \psi(s_t, ...s_{t-p+1}, X_t^1, ..., X_t^h)\varepsilon_{t+1} \tag{60}$$

where ε_t is i.i.d. with $E(\varepsilon_t) = 0$, $E(\varepsilon_t^2) = 1$.

Defining $z_t = (s_t, ...s_{t-p+1}, X_t^1, ..., X_t^h)^\top \in \mathbb{R}^{p+h}, z \in \mathbb{R}^{p+h}$

we can write the AR(p)-ARCH(p) process as

$$s_{t+1} = \phi(z_t) + \psi(z_t)\varepsilon_{t+1} \tag{61}$$

It follows that

$$E[s_{t+1} \mid z_t = z] = \phi(z) \tag{62}$$

$$E[s_{t+1}^2 \mid z_t = z] = \theta(z) \tag{63}$$

$$Var[s_{t+1} \mid z_t = z] = \psi^2(z) \tag{64}$$

$$\psi^2(z) = \theta(z) - \phi^2(z) \tag{65}$$

Using a neural network Φ_{NN} to approximate $\phi(z)$ we obtain

$$\hat{\phi}(z) = \Phi_{NN}(z; \hat{w}) \tag{66}$$

where

$$\hat{w} = \arg\min \frac{1}{N-p} \sum_{t=p}^{N-1} (s_{t+1} - \Phi_{NN}(z_t; w))^2 \tag{67}$$

Using a neural network Θ_{NN} to approximate $\theta(z)$ we obtain

$$\hat{\theta}(z) = \Theta_{NN}(z; \hat{w}) \tag{68}$$

where

$$\hat{w} = \arg\min \frac{1}{N-p} \sum_{t=p}^{N-1} (s_{t+1}^2 - \Theta_{NN}(z_t; w))^2 \tag{69}$$

An estimator of $\psi^2(z)$ can be obtained as

$$\hat{\psi}^2(z) = \hat{\theta}(z) - \hat{\phi}^2(z) \tag{70}$$

The non negativity from $\hat{\psi}^2$ is guaranteed only for special conditions on the architecture of the networks Φ_{NN} and Θ_{NN}. To avoid this constraints there is the approach where the residuals are substituted by the sample residuals (see Franke, Härdle and Hafner (2001)).

The residuals can be written as

$$\epsilon_{t+1} = s_{t+1} - \phi(z_t) = \psi(z_t)\varepsilon_{t+1} \tag{71}$$

As $E(\varepsilon_t^2) = 0$ and $E(\varepsilon_t^2) = 1$, it follows that

$$E[\epsilon_{t+1}^2 \mid z_t = z] = \psi^2(z) \tag{72}$$

Approximating the residuals through the sample residuals

$$\hat{\epsilon}_{t+1} = s_{t+1} - \hat{\phi}(z_t) \tag{73}$$

and the squared sample residuals with a neural network Ψ_{NN} with parameters

$$\hat{w} = \arg\min \frac{1}{N-p} \sum_{t=p}^{N-1} (\hat{\epsilon}_{t+1}^2 - \Psi_{NN}(z_t; w))^2 \tag{74}$$

the estimation of the conditional volatility can be written as

$$\hat{\psi}^2(z) = \Psi_{NN}(z; \hat{w}) \tag{75}$$

4.2.2 Example in XploRe

Using condvolRBF.xpl (see appendix), the time series containing the exchange rates British Pound to US Dollar is used (after log-differences transformation) to train a RBF network with 25 hidden units. The input units are

the 5 lagged values $(t, t-1, ..., t-4)$ of the series and the output unit is the series value at $t+1$. Thus, the series is supposed to follow an AR(5)-ARCH(5) process.

As in the steps described on section 4.2.2, the conditional volatilities are estimated using a RBF network with 25 hidden units. The log returns of the original series and the estimated conditional volatilities are plotted on figure 15.

The same procedure was applied on the series of the Commerzbank stock values. Again a RBF with 25 hidden units was used but with time dependency of lag 2. The log returns of the original series and the estimated conditional volatilities are plotted on figure 16.

4.2.3 Estimation from Implied Volatilities

From the Black Scholes model, the price of a call option C_t at time t is given through the formula

$$C_t = S_t\Phi(d_1) - Ke^{-r\tau}\Phi(d_2) \tag{76}$$

$$d_1 = \frac{\ln \frac{S_t}{K} + (r + \frac{1}{2}\sigma^2)\tau}{\sigma\sqrt{\tau}} \tag{77}$$

$$d_2 = d_1 - \sigma\sqrt{\tau} \tag{78}$$

where S_t is the spot price of the underlying asset, σ the volatility of the underlying asset price process, r the risk free interest rate, τ the time to maturity, K the strike price of the option and Φ the cumulative distribution function of the normal distribution. The Black Scholes model assumes that σ is constant over the price process of a given underlying asset.

In real situations, the option prices and the underlying prices on time t are observable and the strike price K and time to maturity τ are settled on contract. The volatility σ is, however, *not* observable. It is only possible to obtain the volatility *implied* on the option prices, i.e, the value $\hat{\sigma}$ that solves the Black Scholes equation with the another parameters. One can obtain the implied volatility $\hat{\sigma}$ inverting the Black Scholes formula. In XploRe, this can be done with two different numerical methods (bisection and Newton-Raphson) in the quantlet ImplVola.xpl.

In opposite to the theoretical formulation, the implied volatilities are not constant. They form a smile when plotted against the strike prices K at time t, see Härdle, Kleinow and Stahl (2002) and change also according to the time to maturity τ.

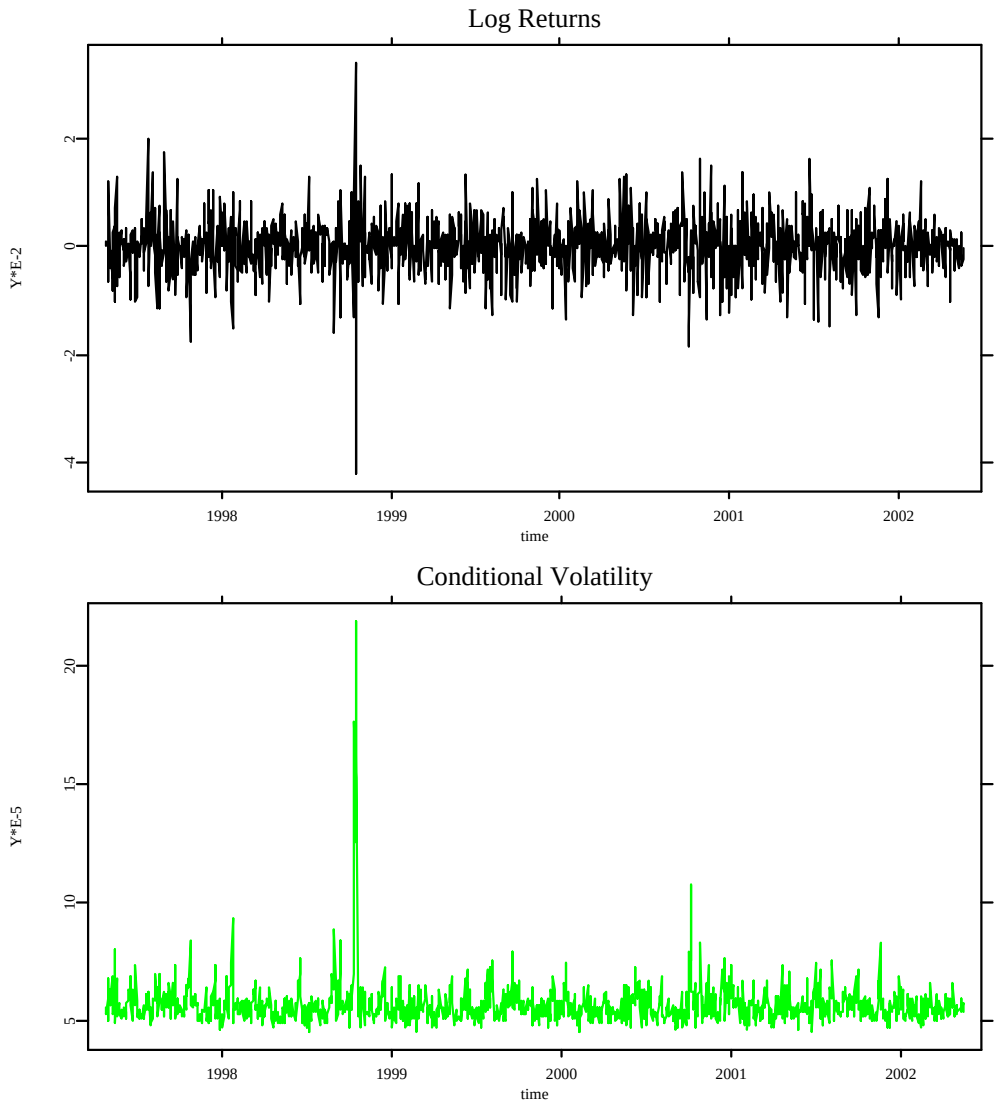


Figure 15: Log returns and conditional volatilities from the exchange rate British Pound / US Dollar from 29/04/1997 to 29/04/2002. Estimated with RBF network, 25 hidden units, lag 5.

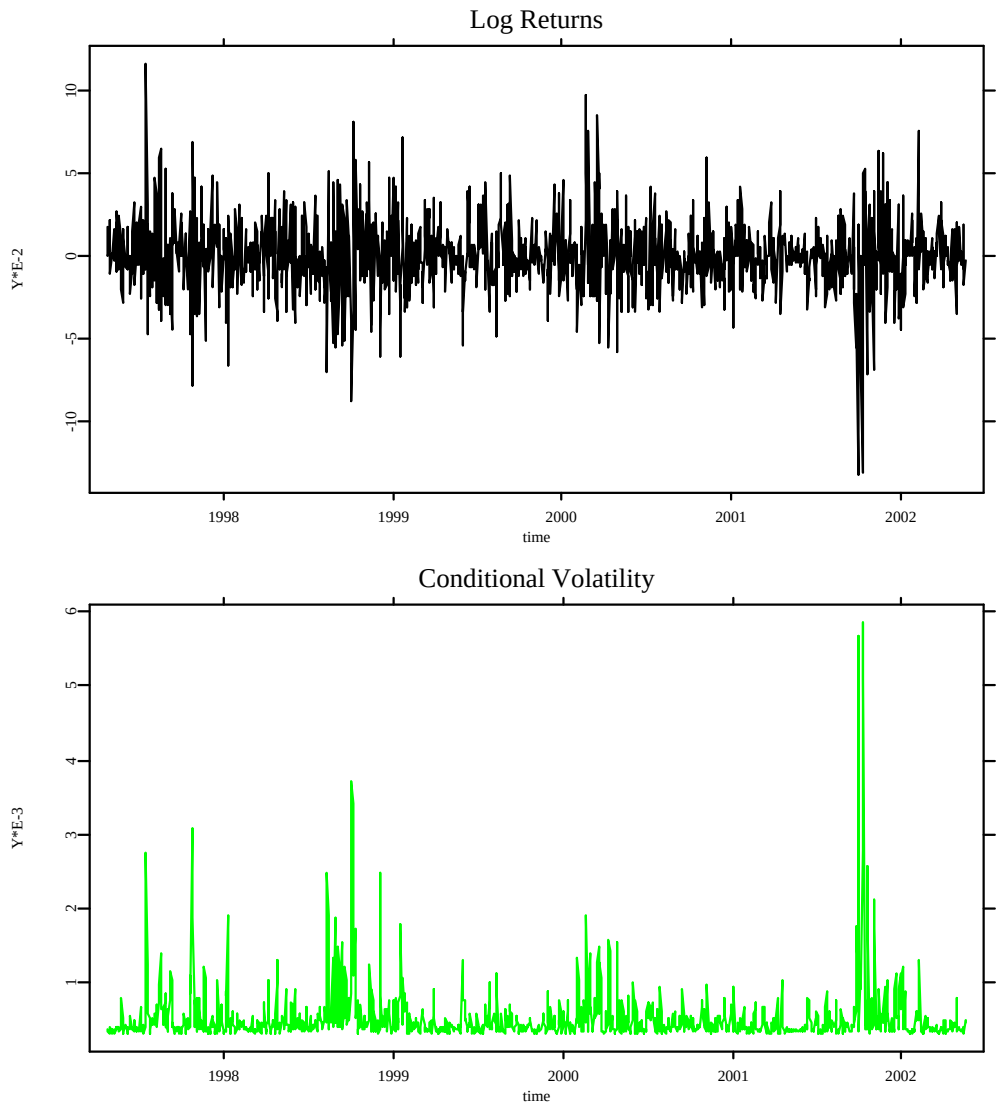


Figure 16: Log returns and conditional volatilities from Commerzbank stock from 29/04/1997 to 29/04/2002. Estimated with RBF network, 25 hidden units, lag 2.

It is possible to estimate the dependency of the implied volatility on another parameters, like the strike price K, the *moneyness* $\frac{K}{S}$ and time to maturity τ. From special practical interest are the estimations

$$\hat{\sigma} = f(K, \tau) \tag{79}$$

or

$$\hat{\sigma} = f\left(\frac{K}{S}, \tau\right) \tag{80}$$

This relation may have a non linear form and can be estimated with neural networks, given that implied volatilities for a strike price or moneyness and for different maturities are available to constitute the training set. The network Ψ_{NN}

$$\hat{\sigma} = \Psi_{NN}\left(\frac{K}{S}, \tau; \hat{w}\right) \tag{81}$$

where

$$\hat{w} = \arg\min \frac{1}{n-1} \sum_{i=1}^{n} \left(\hat{\sigma}_i - \Psi_{NN}\left(\frac{K_i}{S_i}, \tau_i; w\right)\right)^2 \tag{82}$$

is used to estimate volatilities over a grid from maturities and strike prices (or moneyness), producing an *implied volatility surface.*

4.2.4 Examples in XploRe

In XploRe, volsurfRBF.xpl and volsurfMLP.xpl (both listed on appendix) estimate the implied volatility surface from the data set volsurfdata2.dat using neural networks. The data set contains settlement price of the DAX (underlying asset), strike prices, interest rates, times to maturity and prices from puts and calls traded at the German and Swiss Futures Exchange on 04/01/1999.

The implied volatility surface estimated through a RBF network with 25 hidden units is shown on figure 17. The implied volatility surface estimated through a MLP network with 25 hidden units is shown on figure 18. Both pictures also show the implied volatility curves (red), used on the estimation of the surface.

Volatility Surface - RBF network

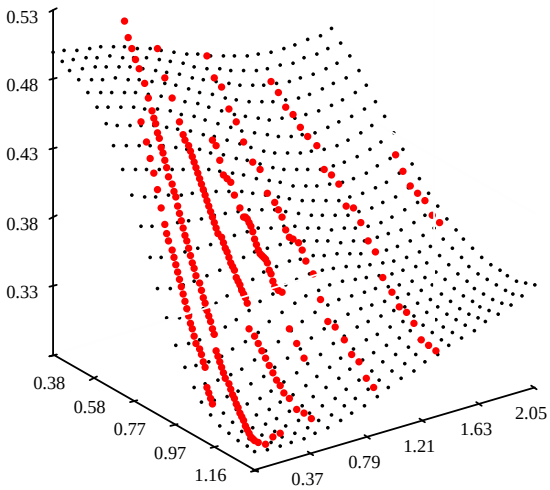


Figure 17: Implied volatility surface estimated using RBF network with 25 hidden units. Parameters: moneyness and maturities. Data: German and Swiss Futures Exchange (EUREX) from 04/01/1999.

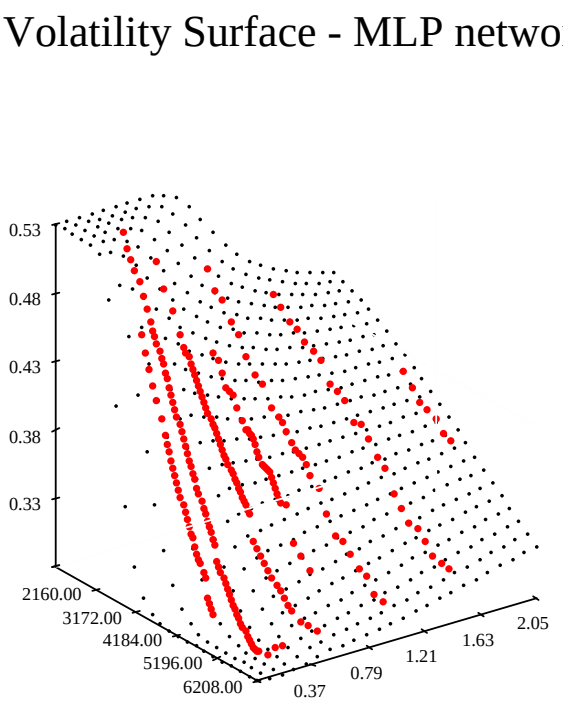


Figure 18: Implied volatility surface estimated using a (2-15-1) MLP. Parameters: strike prices and maturities. Data: German and Swiss Futures Exchange (EUREX) from 04/01/1999.

5 Experiment

5.1 Model

The forecast of time series with neural networks involves the (sometimes heuristical) choice of parameters related to the architecture of the network. The experiment described on this section aims to compare one step ahead forecasts of times series produced by MLP and RBF networks with different architectures.

Three different time series and eight different architectures are used. As in section 4.1.2, a non linear time dependency of size (lag) p, is considered for the series. The experiment uses a network Φ_{NN} with one hidden layer containing h neurons to forecast the realizations of the time series at $t+1$

$$\hat{s}_{t+1} = \Phi_{NN}(s_t, ..., s_{t-p+1})$$

Afterwards, the performance of the forecasts are evaluated.

5.2 Time Series

The time series used contain $t = (1, ..., N)$ daily observations of

- the exchange rate Japanese Yen to US-Dollar (JPYUSD)
- the exchange rate German Mark to US-Dollar (DEMUSD)
- the exchange rate British Pound to US-Dollar (BPUSD)

5.2.1 Transformation

To eliminate trend and seasonality the time series are transformed by first differences of logarithms. After this operation, the time series elements s_t represent the logarithm of the financial return of holding a unit of the currency or stock for one period:

$$s_t = \log(p_t) - \log(p_{t-1}) = \log\left(\frac{p_t}{p_{t-1}}\right) \tag{83}$$

The time series $\{s_t\}_{t=1}^{N}$ are split into two sets, the training set and the test set:

Time Series	from	to	t_0	N
JPYUSD	02/01/1992	12/02/1997	1267	1334
DEMUSD	02/01/1992	12/02/1997	1267	1334
BPUSD	29/04/1997	29/04/2002	1239	1305

Table 7: Time series and sample size

- the training set contains roughly 95% of the observations, i.e., $t = (1, ..., t_0)$, $t_0 = mod(0.95N)$,
- the test set contains roughly 5% of the observations, i.e., $t = (t_0 + 1, ..., N)$.

The table 7 shows the information about the time series and size of subsets used.

5.2.2 Time dependency

The process is modelled with lag 5, the realization at $t + 1$ is dependent on the realizations of the last 5-trading days.

5.3 Networks

There are, as in most neural networks applications, many parameters to choose (number of units, number of hidden layers, type of neurons, learning rates for supervised and unsupervised training, initial weights). The RBF and MLP networks are built with *one* hidden layer of h neurons forming the architecture $5-h-1$. The number h of units on the hidden layer is increased from 10 to 45 in steps from 5 units. For each architecture, the networks are trained on the training sets until a MSE from 5.10^{-5} or less is reached. The another parameters are the defaults for RBF and MLP training quantlets from the XploRe neural networks library.

5.4 Performance Measures

The forecasts are made on the test set where $t = (t_0 + 1, ..., N)$. There are $k = N - (t_0 + 1 + lag)$ forecasts to be compared with the true realizations. Moreover, we define $t_f = t_0 + 1 + lag$. As performance measures are used:

- the normalized mean squared error (NMSE)

$$NMSE = \frac{1}{k}\sum_{t=t_f}^{N}\frac{(s_t - \hat{s}_t)^2}{\hat{\sigma}^2} \tag{84}$$

  where $\hat{\sigma}^2$ is the variance of the training set (in sample unconditional volatility)

- the mean absolute error (MAE)

$$MAE = \frac{1}{k}\sum_{t=t_f}^{N} \mid s_t - \hat{s}_t \mid \tag{85}$$

- the function (SIGN)

$$SIGN = \frac{1}{k}\sum_{t=t_f}^{N} \delta_t \tag{86}$$

  where

$$\delta_t = \begin{cases} 1 & \text{if } s_{t+1}\hat{s}_{t+1} \geq 0, \\ 0 & \text{otherwise} \end{cases} \tag{87}$$

Considering financial returns, it is useful to check if the result of the network can be considered as a trading strategy. For this purpose, the sign of the predicted return and the sign of the real return are compared (in the ideal case they should be the same). The fraction of predictions with same sign as the true realizations is calculated by the function SIGN described above.

5.5 Results and Comments

The results are shown on the tables 8,9, and 10. For 8 different architectures (5 to 45 hidden units in steps of 5)

- JPYUSD: the RBF networks performed better than the MLP for all architectures, concerning NMSE and MAE. The best network is a RBF with 30 hidden units.
- DEMUSD: the number of RBF and MLP networks with better performance concerning NMSE and MAE is the same (4). The best network is a RBF with 45 hidden units, the second best a MLP with 15 hidden units.

- BPUSD: the number of RBF with better performance is 7 concerning NMSE and MAE. The best network is a RBF with 25 hidden units.

We can see from the results that RBF networks do considerably better than MLP networks in extracting the information necessary to perform a good generalization from the training set. This can be a consequence of the unsupervised learning periods that take place on the RBF training algorithm (where the clusters centers and deviances are learnt) but not on the MLP training periods. Therefore, the training of the RBF networks may be faster and more efficient. On the another side, the worse generalization from the MLP can be caused by overfitting the training data. In this case, the MLP may learn information specific to the training set that has no use for generalization, see Anders (1997). Besides that, one have to consider the possibility that MLPs with more than one hidden layer may generalize better, maybe better than RBFs.

The number of hidden units used does not seem to have a straight relation with the forecast performance. Networks with few hidden units performed better than networks with many hidden units and the way around. This was verified for both MLPs and RBFs.

The amount of data on the training set and how the original series is split to form it may also influence the results. The use, for example, of only the first half of data for training may exclude some periods of special economic instability that are not specially useful for generalizing purposes. That may improve the performance on test sets.

The initial weights and learning rates were the default from the quantlets on the XploRe library for neural networks. Another initial weights and learning rates might have an effect on the generalization characteristics of the respective networks.

RBFs perform better on the specific data set with the specific configuration from this experiment. Considering the number of parameters available for adjusting and the number of variables that simultaneously influence the result, it is not possible to conclude in a general sense which network type or architecture forecasts better.

Network	Lag	Hidden neurons	NMSE	$MAE.10^2$	SIGN
RBF	5	10	0.66664	0.45441	0.5
		15	0.51549	0.40042	0.57143
		20	0.6066	0.45364	0.60714
		25	0.50691	0.41894	0.57143
		30	0.4877	0.37785	0.625
		35	0.52913	0.39829	0.58929
		40	0.52948	0.39702	0.64286
		45	0.63094	0.43072	0.53571
MLP	5	10	0.73479	0.47821	0.58929
		15	0.72187	0.46555	0.64286
		20	1.1409	0.58655	0.57143
		25	0.87558	0.51119	0.58929
		30	0.83735	0.49527	0.64286
		35	0.83735	0.64515	0.5
		40	1.2757	0.64169	0.60714
		45	1.2391	0.62611	0.58929

Table 8: Performance of networks on JPYUSD

Network	Lag	Hidden neurons	NMSE	$MAE.10^2$	SIGN
RBF	5	10	0.66274	0.40932	0.69355
		15	0.66799	0.44215	0.64516
		20	0.65755	0.44772	0.54839
		25	1.0278	0.55917	0.67742
		30	0.85476	0.52001	0.35484
		35	0.87699	0.53147	0.37097
		40	0.6434	0.42877	0.66129
		45	0.63786	0.41272	0.62903
MLP	5	10	0.66647	0.43303	0.64516
		15	0.64716	0.41818	0.69355
		20	0.73808	0.43648	0.72581
		25	0.66557	0.42872	0.69355
		30	0.65714	0.4286	0.62903
		35	0.64651	0.41912	0.67742
		40	0.74276	0.4598	0.64516
		45	0.64531	0.42885	0.74194

Table 9: Performance of networks on DEMUSD

Network	Lag	Hidden neurons	NMSE	$MAE.10^2$	SIGN
RBF	5	10	0.37297	0.24671	0.59016
		15	0.39721	0.25636	0.59016
		20	0.42154	0.26729	0.55738
		25	0.33938	0.23938	0.60656
		30	0.51186	0.30311	0.55738
		35	0.38701	0.25412	0.45902
		40	0.49345	0.29482	0.52459
		45	0.42195	0.26568	0.47541
MLP	5	10	0.55015	0.29609	0.55738
		15	0.55072	0.29272	0.55738
		20	0.69091	0.31388	0.55738
		25	0.64251	0.31937	0.54098
		30	0.47864	0.28435	0.54098
		35	0.51331	0.28541	0.54098
		40	0.49738	0.29364	0.4918
		45	0.50398	0.29107	0.58929

Table 10: Performance of networks on BPUSD

6 Summary and Conclusion

This work presented an overview on neural networks and its basic building block, the neuron. Besides that, it covered a general approach to networks statistical learning process, followed by technical exposition on descending gradients and backpropagation methods. The applications described were concentrated on time series prediction, estimation of conditional volatilities from historical time series and estimation of surfaces of underlying asset volatilities implied on option prices. A small experiment compared forecast performances of RBF and MLP networks on different exchange rate series.

As seen from the summary above, neural networks provide quantitative finance with strong support in problems related to non-parametric regression. Also remarkable are the heuristic considerations involved on the set up of neural networks: sometimes parameters and architectures are chosen only by trial and error. A deeper comprehension of the mechanisms and techniques used on the development of neural networks is necessary and decisive to its successful implementation. Thus, the realization of further work in this direction (for example, comparison between performance from networks with different architectures and initial parameters, evaluation of overfitting, opti-

mization of the stopping time in training periods) would be useful, probably resulting in more effective use of neural networks.

Concerning networks applications, new studies may be directed toward another practical financial problems: estimation of conditional Value at Risk, development of automatization of strategies for trading and portfolio hedging, development of credit scoring and default prediction tools. Finally, beyond neural networks but still in the framework of statistical learning, support vector machines remain as an interesting and challenging field.

7 References

Anders, U. (1997). *Statistische neuronale Netze*, Verlag Vahlen, München.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford.

Duda, R., Hart, P. and Stork, D. (2000). *Pattern Classification*, Wiley, New York.

Franke, J., Härdle, W. and Hafner, C. (2003). *Einführung in die Statistik der Finanzmärkte*, Springer Verlag, Heidelberg.

Haykin, S. (1999). *Neural Networks*, Prentice Hall, Upper Saddle River.

Härdle, W., Kleinow, T. and Stahl, G. (2002). *Applied Quantitative Finance*, Springer Verlag, Heildelberg.

Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*, Springer Verlag, New York.

Vidyasagar, M., (1997). *A Theory of Learning and Generalization*, Springer Verlag, London.

A Appendix

A.1 descgrad.xpl

```
proc()=SFMdescgrad(x,y,w,bias,p,epoch) ;
----------------------------------------------------------------------
; EBook     SFM ;
----------------------------------------------------------------------
; See_also   SFMerrsurf ;
----------------------------------------------------------------------
; Macro       SFMdescgrad ;
----------------------------------------------------------------------
; Description plots minimization of Q(w), ; using sigmoid
activation y=1./(1+exp(-p*x)), ; following method of descending
gradients ;
----------------------------------------------------------------------
; Usage       SFMdescgrad(x,y,w,bias,p,epoch) ;
----------------------------------------------------------------------
; Keywords   Neural Network ;
----------------------------------------------------------------------

; Author      E. Giacomini, 20030625 ;
----------------------------------------------------------------------

  res=errsurf(x,y,w,bias,1,p)     ; creates plot over a grid
  w=#(-0.7,0.7 )                           ; initial weights
  e=1                              ; initializes epochs
  aa=0~0~0
  while(e<=epoch)
    sumerro=0
    sumgrad=0
    i=1
    while(i<=rows(x))                   ; calculates for each weight
      inp=(sum(w.*x[i,]')-bias)         ; the error function Q(w)
      activ=1/(1+exp(-p*inp))           ; activation function
      deriv=activ*(1-activ)             ; derivative of activation function
      erro=(y[i,]-activ)
      sqerr=erro^2                      ; squared error
      sumerro=sumerro+sqerr
      grad=2*erro*deriv*x[i,]'            ; gradient
```

```
      sumgrad=sumgrad+grad              ; sum up gradients
      i=i+1
    endo
;sumerro=sumerro/2         ; Q(w) for given weights
    bb=(w'~sumerro)
    w=w+sumgrad               ; corrects weights
    aa=aa|bb
    dd=aa[2:rows(aa),]
    cc=setmask(aa[2:rows(aa),],"line","red","thin")
    setmaskp(dd,4,8,6)
    res=setmask(res,"point","black","size","tiny")

    ; PLOT WITH PATH
    plot3d(1,res,dd,cc)       ; plots Q(w) and calculated weights

    ; PLOT WITHOUT PATH
    ;setmaskp(bb,4,8,6)       ; plots Q(w) and calculated weights
    ;plot3d(1,res,bb)

    e=e+1                     ; next training period
  endo
endp

proc(aa)=errsurf(x,y,w,bias,activ,p)
  w4=0*matrix(rows(w),1)
  i=1
  while(i<=rows(x))
    w1=x[i,1]*w[,1]+x[i,2]*w[,2]-bias
    if (activ==1)
      w2=1./(1+exp(-p*w1))
    endif
    if (activ==0)
      w2=w1/abs(w1)
      w2=replace(w2,#(NaN,-1),0)
    endif
    w3=(y[i]-w2)^2
    w4=w4+w3
    i=i+1
  endo
  aa=(w~w4)
endp
```

A.1.1 Example

```
; Input, target, grid
; bias, parameter of sigmoid function and
periods of training
library("plot")
x=#(0,0)'|#(1,0)'|#(0,1)'|#(1,1)' y=#(0,1,0,1) ; Boolean x1
;y=#(0,0,0,1)                                  ; Boolean x1 AND x2
w=grid(#(-1,-1),#(0.05,0.05),#(60,60))
bias=0.5
p=5
epoch=8
SFMdescgrad(x,y,w,bias,p,epoch)
setgopt(plot3disp,1,1,"title","Error Surface: Learning
weights","border",0)
```

A.2 TSNN.xpl

```
proc()=TSNN(ts)
  headline = "Please select neural network"
  items = "MLP" | "RBF"
  network = selectitem(headline, items)

  ValueNames = "Please select % of data used for training"
  |"Please select lag "
  defaults = 0.95|5
  values= readvalue(ValueNames, defaults)
  tr=values[1]
  lag=values[2]

;log difference
  x=diff(log(ts))

; tr% of data as training set
  h=floor(tr*rows(x))
  xtrain=x[1:h]
```

```
  hh=h+1

; (1-tr)% as test set
;  xtes=x[hh:rows(x)] ; 60 periods as test set
  xtes=x[hh:(hh+60)]

  if (network[1])
    ValueNames = "Number of neurons in the hidden layer "
    |"Number of training periods "
    defaults = 25 | 100
    parmlp = readvalue(ValueNames, defaults)
    par = 0|0|0|0|0.7|0|parmlp[2]|0


; prepares input matrix, output target
    standtrain =(xtrain-min(xtrain))./(max(xtrain)-min(xtrain))
    standtes =(xtes-min(xtes))./(max(xtes)-min(xtes))
    train=lagged(standtrain,lag)
    tes=lagged(standtes,lag)

;trains MLP network

    net = nnrnet2(train.yt, train.tt, matrix(rows(train.yt)), parmlp[1],par)

;predicts training set

    nntra  = nnrpredict2(train.yt, net)

; predicts test set

    nntes  = nnrpredict2(tes.yt, net)

; rescale data

    nntra =min(xtrain)+(nntra.*(max(xtrain)-min(xtrain)))
    nntes =min(xtes)+(nntes.*(max(xtes)-min(xtes)))
    train.tt =min(xtrain)+(train.tt.*(max(xtrain)-min(xtrain)))
    tes.tt =min(xtes)+(tes.tt.*(max(xtes)-min(xtes)))
    vardata=var(train.tt)
; evaluate prediction for MLP
```

```
    MAE=sum(abs(nntes-tes.tt))/rows(tes.tt)
    MSE=(sum((nntes-tes.tt)^2))/rows(tes.tt)
    NMSE=MSE/vardata
    SIGN=sum(sign(sign(tes.tt.*nntes)+1))/rows(tes.tt)
    MSE
    MAE
    NMSE
    SIGN
    summarize(tes.tt)

  endif

  if(network[2])

    ValueNames = "Number of neurons in the hidden layer "
    |"Periods unsupervised learning"|"Periods supervised learning"|"minimum MSE"
    defaults = 25 | 100 | 1000 | 0.00005
    parrbf = readvalue(ValueNames, defaults)



; prepares input matrix, output target
    train=lagged(xtrain,lag)
    tes=lagged(xtes,lag)
    vardata=var(train.tt)

    clusters = parrbf[1]
    learn    = 0.1|0.2|0.1
    epochs   = parrbf[2]|parrbf[3]
    mMSE     = parrbf[4]
    activ    = 0

;trains MLP network
    net = rbftrain2(train.yt,train.tt,clusters,learn,epochs,mMSE,activ)

;predicts training set
    nntra=rbfpredict(train.yt,net,min(train.tt),max(train.tt))

; predicts test set
    nntes=rbfpredict(tes.yt,net,min(tes.tt),max(tes.tt))  ; prediction
```

```
; evaluate prediction for RBF

    testtest=rbftest(tes.yt,tes.tt,net)
    MAE=sum(testtest.AED)/rows(tes.tt)
    MSE=testtest.MSE
    NMSE=MSE/vardata
    SIGN=sum(sign(sign(tes.tt.*nntes)+1))/rows(tes.tt)
    MSE
    MAE
    NMSE
    SIGN
    summarize(tes.tt)

  endif

;TRAINING plots real data (blue)  versus result from network (red)
  t=(1:rows(train.tt))
  tr=setmask(t~nntra,"line","red","thin")
  ytr=setmask(t~train.tt,"line","blue","thin")

;TEST plots real data (blue)  versus result from network (red)
  t=(1:rows(tes.tt))
  tttt=setmask(t~nntes,"line","red","thin")
  yyyy=setmask(t~tes.tt,"line","blue","thin")

; EVALUATION plots
  fmt="%10.10f"
  text=#("MSE ","NMSE","MAE ","SIGN")
  str=string(fmt,#(MSE,NMSE,MAE,SIGN))
  outtext=text+"  "+str

  disp=createdisplay(1,1)
; shows graphic ; show(disp,1,1,tr,ytr)
  show(disp,1,1,tttt,yyyy)
;  show(disp,2,1,outtext)
  setgopt(disp,1,1,"title","Neural Network Forecast",
  "xlabel","Test Set","border",0)
endp

proc(tt,yt)=lagged(xt,lag) ; rearranges data vector, given a lag,
```

```
; to form input(yt) matrix and output(tt) vector
  r=rows(xt)
  tt=xt[(lag + 1):(r),]
  yt=xt
  i=1
  while(i<=(lag-1))
    yt=reshape(yt[2:r]|yt[1],#(r,1))

    xt=xt~yt
    i=i+1
  endo
  yt=xt[1:r-lag]
endp
```

A.2.1 Example

```
axeson()
library("stats")
library("nn")
library("plot")
XFGTHB=read("XFGTHBbasket.dat")
r=rows(XFGTHB)
JPYUSD=XFGTHB[4:r,11]
DEMUSD=XFGTHB[4:r,10]
SFM02=read("SFM02.dat")
BPUSD=SFM02[,2]
GOLD=SFM02[,4]
Commerzbank=SFM02[,5]

TSNN(JPYUSD)
```

A.3 condvolRBF.xpl

```
proc()=condvolRBF(x,lag,clusters,learn,epochs,mMSE,activ) ; takes
log diff
  x=diff(log(x))
  tt=1:rows(x)
```

```
; prepares input matrix, output target
  train=lagged(x,lag)
  t=(1:rows(train.tt))

; trains rbf - neural network
  rbfnet=rbftrain2(train.yt,train.tt,clusters,learn,epochs,mMSE,activ)
  rbftra=rbfpredict(train.yt,rbfnet,min(train.tt),max(train.tt))

; squared sample residuals
  eps2=(train.tt-rbftra)^2
  rbfvol=rbftrain2(train.yt,eps2,clusters,learn,epochs,mMSE,activ)

; prediction
  vol=rbfpredict(train.yt,rbfvol,min(eps2),max(eps2))

; plots results
  disp=createdisplay(2,1)
  t=(1:rows(eps2))./257 + (1997+80/257)
  tt=(1:rows(x))./257 + (1997+80/257)
  vol=setmask(t~vol,"line","green","thin")
  series=setmask(tt~x,"line","black","thin")
  show(disp,1,1,series)
  show(disp,2,1,vol)
  setgopt(disp,1,1,"title","Log Returns","xlabel","time","border",0)
  setgopt(disp,2,1,"title","Conditional Volatility"
  ,"xlabel","time","border",0)

endp

proc(tt,yt)=lagged(xt,lag) ; rearranges data vector, given a lag,
; to form input(yt) matrix and output(tt) vector
  r=rows(xt)
  tt=xt[(lag + 1):(r),]
  yt=xt
  i=1
  while(i<=(lag-1))
    yt=reshape(yt[2:r]|yt[1],#(r,1))
    xt=xt~yt
    i=i+1
  endo
  yt=xt[1:r-lag]
```

```
endp
```

A.3.1 Example

```
axeson()
 library("nn")
 library("plot")
XFGTHB=read("XFGTHBbasket.dat")
r=rows(XFGTHB)
JPYUSD=XFGTHB[4:r,11]
DEMUSD=XFGTHB[4:r,10]
SFM02=read("SFM02.dat")
BPUSD=SFM02[,2]
GOLD=SFM02[,4]
Commerzbank=SFM02[,5]
clusters = 25
learn    = 0.1|0.2|0.1
epochs=
100|1500
mMSE     = 0.00005
activ    = 0
lag=5 ;sets lag
condvolRBF(BPUSD,lag,clusters,learn,epochs,mMSE,activ)
```

A.4 volsurfRBF.xpl

```
proc()=volsurfRBF(x,metric,clusters,learn,epochs,mMSE,activ,IVmethod)

  ;check errors, messages
  error((sum(sum(x<0)')>=1),
  "volsurfRBF: Watch out! Some of your data entries are negative!!")
  error((cols(x)!=6),
  "volsurfRBF: data matrix must contain 6 columns")
  error((sum((x[,6]!=1)&&(x[,6]!=0))>=1),
  "volsurfRBF: type needs to be either 1 or zero")
  error(((metric!=0)&&(metric!=1)),
  "volsurfRBF: metric needs to be either 1 or 0")
```

```
  ;numerical method and calculation of implied volatility
  if (exist("IVmethod")==0)
    y=ImplVola(x)
  else
    if (IVmethod=="bisect")
      y=ImplVola(x,"bisect")
    else
      error(1,"volsurfRBF: Your choice of method is unclear")
    endif
  endif

  if ((sum(y<0)>=1))
    "The following lines violate no arbitrage conditions"
    "These are the wrong data sets. "
    "First colums is the row number of your orginal data set."
    paf (((y<0)>=1).*(1:rows(y))~x, ((y<0)>=1))
    error(1,"volsurfRBF:
    Watch out! Your data violate no arbitrage conditions.
     See output window")
  endif

;metric moneyness or strike
  m=matrix(rows(x))
  if(metric==0)
    m=(x[,2]./(x[,1].*exp(x[,3].*x[,4])))
  else
    m=x[,2]
  endif

; prepares training set for network
  rbfx=m~x[,4]
  rbfy=y
; training
  rbfnet = rbftrain2(rbfx,rbfy,clusters,learn,epochs,mMSE,activ)
; prediction on grid of values
  origin=(min(rbfx))'
  step=#((max(rbfx)-min(rbfx))./20)
  newx=grid((origin-2*(step)),step, #(24,24))
  result=rbfpredict(newx,rbfnet,min(rbfy),max(rbfy))

; surface and smiles: graphical object, plot
```

```
  rbfsurf=newx~result   ; surface
  smile=m~x[,4]~y       ; smile
  smileplotdot  = setmask(smile, "fillcircle","small", "red")
  rbfplotdot  = setmask(rbfsurf, "fillcircle","tiny", "black")
  plot3d(1,rbfplotdot,smileplotdot)

endp
```

A.4.1 Example

```
library("plot")
library("nn")
library ("finance")
x=read("volsurfdata2.dat")      ; read the data
clusters = 25
learn    = 0.1|0.2|0.1
epochs   = 100|500
mMSE     = 0.0000005
activ    = 0
volsurfRBF(x,0,clusters,learn,epochs,mMSE,activ)
setgopt(plot3disp,1,1,"title"," Volatility Surface - RBF network")
```

A.5 volsurfMLP.xpl

```
proc()=volsurfMLP(x,metric,neuron,IVmethod)

;check errors, messages
  error((sum(sum(x<0)')>=1),"volsurfRBF: Watch out!
  Some of your data entries are negative!!")
  error((cols(x)!=6),"volsurfRBF: data matrix must contain 6 columns")
  error((sum((x[,6]!=1)&&(x[,6]!=0))>=1),"volsurfRBF:
   type needs to be either 1 or zero")
  error(((metric!=0)&&(metric!=1)),
  "volsurfRBF: metric needs to be either 1 or 0")

;numerical method and calculation of implied volatility
  if (exist("IVmethod")==0)
```

```
    y=ImplVola(x)
  else
    if (IVmethod=="bisect")
      y=ImplVola(x,"bisect")
    else
      error(1,"volsurfMLP: Your choice of method is unclear")
    endif
  endif

  if ((sum(y<0)>=1))
    "The following lines violate no arbitrage conditions"
    "These are the wrong data sets. "
    "First colums is the row number of your orginal data set."
    paf (((y<0)>=1).*(1:rows(y))~x, ((y<0)>=1))
    error(1,"volsurfMLP: Watch out!
     Your data violate no arbitrage conditions. See output window")
  endif

;metric moneyness or strike
  m=matrix(rows(x))
  if(metric==0)
    m=(x[,2]./(x[,1].*exp(x[,3].*x[,4])))
  else
    m=x[,2]
  endif

; prepares training set for network
  xx=m~x[,4]
  yy=y
; uniformize data for training
  mlpx =(xx-min(xx))./(max(xx)-min(xx))
  mlpy =(yy-min(yy))./(max(yy)-min(yy))
; training
  mlpnet = nnrnet2(mlpx,mlpy,matrix(rows(mlpx)),neuron)
; prediction on grid of values
  origin=(min(mlpx))'
  step=#((max(mlpx)-min(mlpx))./20)
  newx=grid((origin-2*(step)),step, #(24,24))
  result=nnrpredict2(newx,mlpnet)

; scale prediction back, create grid to plot
```

```
  result=(result).*(max(yy)-min(yy))+min(yy)
  origin=(min(xx))'
  step=#((max(xx)-min(xx))./20)
  newxx=grid((origin-2*(step)),step, #(24,24))

; surface and smiles: graphical object, plot
  mlpsurf=newxx~result  ; surface
  smile=m~x[,4]~y       ; smile
  smileplotdot  = setmask(smile, "fillcircle","small", "red")
  mlpplotdot  = setmask(mlpsurf, "fillcircle","tiny", "black")
  plot3d(1,mlpplotdot,smileplotdot)

endp
```

A.5.1 Example

```
library("plot")
library("nn")
library ("finance")
x=read("volsurfdata2.dat")      ; read the data
neuron=15
volsurfMLP(x,1,neuron)
setgopt(plot3disp,1,1,"title"," Volatility
Surface - MLP network")
```