

Neural Networks Approach

3.1 Introduction

Neural networks are massively parallel, distributed processing systems representing a new computational technology built on the analogy to the human information processing system. That is how we know the neural networks today, but the evolution of artificial neural networks, from the early idea of neuro-physiologist Heb (1949) about the structure and the behaviour of a biological neural system up to the recent model of artificial neural system, was very long. The first cornerstones here were laid down by the neurologists McCulloch and Pitts (1943) who, using formal logic, modelled neural networks using the neurons as binary devices with fixed thresholds interconnected by synapses. Nevertheless, the list of pioneer contributors in this field of work is long. It certainly includes the names of distinguished researchers like Rosenblatt (1958), who extended the idea of the *computing neuron* to the *perceptron* as an element of a self-organizing computational network capable of learning by feedback and by structural adaptation. Further pioneer work was also done by Widrow and Hoff (1960), who created and implemented the analogue electronic devices known as ADALINE (Adaptive Linear Element) and MADALINE (Multiple ADALINE) to mimic the neurons, or perceptrons. They used the least mean squares algorithm, simply called the *delta rule*, to train the devices to learn the pattern vectors presented to their inputs. In 1969, Minsky and Papert (1969) portrayed perceptron history in an excellent way but their view, that the multilayer perceptron (MLP) systems had limited learning capabilities similar to the one-layer perceptron system, was later disproved by Rumelhart and McClelland (1986). Rumelhart and McClelland in fact showed that multilayer neural networks have outstanding nonlinear discriminating capabilities and are capable of learning more complex patterns by *backpropagation learning*. This essentially terminates the most fundamental development phase of perceptron-based neural networks.

After a period of stagnation, the research interest was turned to the possible alternative network variants that have been found in *self-organizing networks*

(Amari and Maginu, 1988), *resonating neural networks* (Grossberg, 1988), *feedforward networks* (Werbos, 1974), *associative memory networks* (Kohonen, 1989), *counterpropagation networks* (Hecht-Nielsen, 1987a), *recurrent networks* (Elman, 1990), *radial basis function networks* (Broomhead and Lowe, 1988), *probabilistic networks* (Specht, 1988), *etc.* Nevertheless, up to now, the most comprehensively studied and, in engineering practice, most frequently used neural networks are the multilayer perceptron networks (MLPN) and radial basis function networks (RBFN), which are frequently the subject of further research and applications.

Neural networks have, since the very beginning of their practical application, proven to be a powerful tool for signal analysis, features extraction, data classification, pattern recognition, *etc.* Owing to their capabilities of learning and generalization from observation data, the networks have been widely accepted by engineers and researchers as a tool for processing of experimental data. This is mainly because neural networks reduce enormously the computational efforts needed for problem solving and, owing to their massive parallelity, considerably accelerate the computational process. This was reason enough for intelligent network technology to leave soon the research laboratories and to migrate to industry, business, financial engineering, *etc.* For instance, the neural-network-based approaches developed and the methodologies used have efficiently solved the fundamental problems of time series analysis, forecasting, and prediction using collected observation data and the problems of on-line modelling and control of dynamic systems using sensor data.

Generally speaking, the practical use of neural networks has been recognized mainly because of such distinguished features as

- general nonlinear mapping between a subset of the past time series values and the future time series values
- the capability of capturing essential functional relationships among the data, which is valuable when such relationships are not *a priori* known or are very difficult to describe mathematically and/or when the collected observation data are corrupted by noise
- universal function approximation capability that enables modelling of arbitrary nonlinear continuous functions to any degree of accuracy
- capability of learning and generalization from examples using the data-driven self-adaptive approach.

3.2 Basic Network Architectures

The model of the basic element of a neural network *i.e.* the neuron, as still used today was originally worked out by Widrow and Hoff (1960). They considered the perceptron as an adaptive element bearing a resemblance to the neuron (Figure 3.1). A neuron, as the fundamental building block of a neural information processing system, is made up of (see Figure 3.1)

- a *cell body* with an inherent *nucleus*

- **dendrites** that feed the external signals to the cell body
- **axons** that carry the signals out of the cell to other cell bodies

This configuration was translated in terms of analogue computational technology as shown in Figure 3.1, where

- the core part of the element, called a perceptron, contains a summing element Σ and a nonlinear element NL
- the multiple signal inputs x_i are connected via adjustable weighting elements w_i with the core part of the element
- the signal output(s) y_d

An additional perceptron input w_0 , called the *bias*, is understood as a threshold (switching) element.

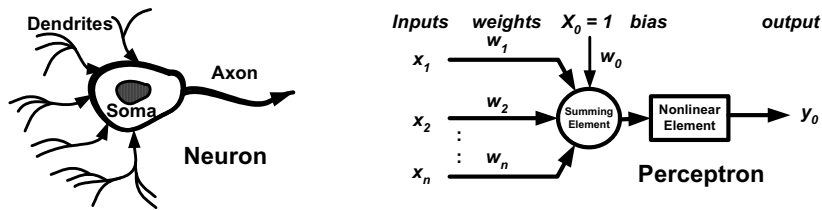


Figure 3.1. Symbolic representation of neuron and perceptron

The output signal is defined as

$$y_0 = f\left(\sum_{i=1}^n w_i x_i + w_0\right)$$

and the bias follows the relationship

$$w^T x + w_0 \geq 0$$

meaning that the perceptron *fires*, i.e. it is activated and produces an output signal when this condition is met, otherwise not.

Our attention should now be shifted to the question of what nonlinear function should be implemented in the core part of the perceptron as its **activation function**. The early attempt of Block (1962) to select the **binary step function** for this purpose was later modified in favour of a **sigmoid activation function** (Figure 3.2).

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

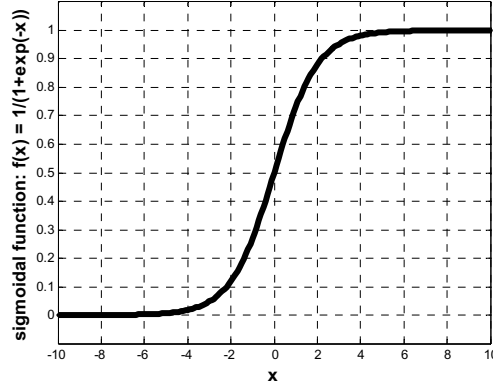


Figure 3.2. Sigmoid activation function

The perceptron basically learns through a training process, based on a set of collected data. During the training, the perceptron adjusts its interconnection weights according to the data presented at its input. For adjusting the perceptron weights, Widrow and Hoff (1960) originally proposed using the *delta rule*, i.e. the recursive gradient-type of learning algorithm (the so-called *α -LMC Algorithm*) that adds to the current weight value $w(k)$ a compensation term $\eta \varepsilon(k)x(k)$, to build the next weight value

$$w(k+1) = w(k) + \eta \varepsilon(k)x(k),$$

where η is a proportionality term, $\varepsilon(k)$ is the error at the adjusting step k , and $x(k)$ the value of the input signal at the current step k .

Although rather simple, the delta learning rule has, in the majority of cases, demonstrated a high efficiency and a high convergence speed in perceptron training. Even so, a single perceptron alone cannot learn enough to be capable of solving more complex problems because its radius of computational action is rather restricted by the simplicity of its structure. This was demonstrated in an example of a perceptron as a pattern classifier. Owing to its restricted structural capabilities the perceptron can only solve the *linearly separable problems*. It is thus far away from being a general-purpose processing device. But, the fundamental erroneous belief of Minsky was that even multiple perceptron layer devices cannot build a universal general-purpose processing machine. This was disproved by building the *multilayer perceptrons* (MLPs) that, in addition to the perceptron *input layer* and *output layer*, also include so-called *hidden layers* inserted between the input and the output layer to form a cascaded network structure with extended connectionist capabilities (see Section 3.3.1). The term hidden layer was selected for the intermediate layer because this layer is only accessible through the input and/or the output layer but not directly. In practice, one hidden layer is usually sufficient to build the network with the extended

computational capabilities for solving the majority of practical problems. Only in some rare cases some additional hidden layers could be needed. This also holds in time series analysis and forecasting applications.

Accidentally, the concept of the perceptron emerged at that time when the difficulties in solving complex intelligent problems using classical computing automata of John von Neumann had grown to be insurmountable. It was realized that, for solving such problems, massive, highly parallel, distributed data processing systems are required. Building of such highly sophisticated computational systems was already put on the agenda of some leading research institutions. However, discovery of the perceptron as a simple computing element that can easily be mutually interconnected with other perceptrons to build huge computing networks was viewed as a more promising way for development of the massive parallel computational systems needed at that time. Minsky and Papert (1969) expected that the use of more complex, MLP configurations could help in building the future intelligent, general-purpose computers with learning and cognition capability. This was very soon proven using perceptrons as the basic elements of ADALINE (A) in single-layer perceptrons to build a multi-layer MADALINE architecture (see Figure 3.3).

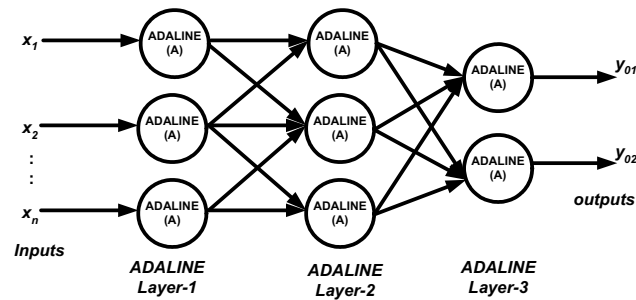


Figure 3.3. ADALINE-based MADALINE

In 1950, Rosenblatt used a single perceptron layer for optical character recognition. It was a multiple input structure fully connected to the perceptron layer with adjustable multiplicative constants w_i called weights. The input signals, before being forwarded to the *processing elements* (i.e. perceptrons) of the single network layer, are multiplied by the corresponding values of the weighting elements. The outputs of the processing units build a set of signals that determine the number of pattern classes that can be distinguished in the input data sets by the linear separation capability of perceptron layer. For weight adjustment Rosenblatt used the delta rule.

3.3 Networks Used for Forecasting

Hu (1964) was the first to demonstrate - on a practical weather forecasting example - the general forecasting capability of neural networks. Werbos (1974) later experimented with the neural networks as tools for time series forecasting, based on observational data. However, apart from some isolated attempts to solve the forecasting problems using the then still poorly developed neural networks technology, the research work in practical application of neural networks had generally undergone a long period of stagnation. The stagnation was broken and the work on neural network applications enthusiastically resumed after the backpropagation training algorithm was formulated by Rumelhart *et al.* (1986). Experimenting with the backpropagation-trained neural networks, Werbos (1989, 1990) also concluded that the networks even outperform the statistical forecasting methods, such as *regression analysis* and the *Box-Jenkins forecasting approach*. Lapedes and Farber (1988) also successfully used neural networks for modelling and prediction of nonlinear time series.

In the following, typical neural networks used for forecasting and prediction purposes will be described.

3.3.1 Multilayer Perceptron Networks

Although in the meantime the variety of proposed neural network structures has grown, the multilayered perceptron has remained the prevailing one and also the most widespread network structure. This particularly holds for the three-layer network structure in which the *input layer* and the *output layer* are directly interconnected with the intermediate single *hidden layer*. The inherent capability of the three-layer network structure to carry out any arbitrary input-output mapping highly qualifies the multilayer perceptron networks for efficient time series forecasting. When trained on examples of observation data, the networks can learn the characteristic features “hidden” in the examples of the collected data and even generalize the knowledge learnt, which will be discussed later in detail.

The multilayer perceptron, because of its cascaded structure, performs the input-output mapping of nonlinearities. For instance, the input-output mapping of a one hidden layer perceptron network can generally be written as

$$y = f_o \left(\sum w_h f_h \left(\sum f_i (w_i^T x) \right) \right).$$

Relying on the Stone-Weierstrass theorem, which states that any arbitrary function can be approximated with a given accuracy by a sufficiently large-order polynomial, Cybenko (1989) and Hornik *et al.* (1989) proved that a single hidden layer neural network is a *universal approximator* because it can approximate an arbitrary continuous function with the desired accuracy provided that the number of perceptrons in it is high enough. This network capability is general, *i.e.* it does not depend on the shape of the perceptron activation function if it is nonlinear.

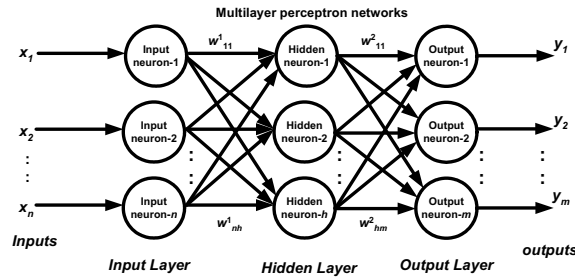


Figure 3.4 Multilayer perceptron architecture

Rumelhart and McClelland (1986, MIT book) suggested for multilayer neural networks the **backpropagation learning rule**. This has also widely been accepted. Later, various accelerated versions of the rule have been elaborated that speed up the learning process. In the meantime, the multilayer perceptron networks trained to learn using backpropagation algorithm are simply called **backpropagation networks**.

The learning capability of backpropagation networks is mainly due to the internal mapping of the characteristic signal features in the process of network training onto the hidden layer. The mappings stored in this layer during the training phase of the network can be automatically retrieved during its application phase for further processing. Although the features-capturing capability of the network can be extended enormously when a second hidden layer is added, the additional training and computational time required in this case, however, advises the network user not to do this, if it is not absolutely required by the complexity of the problem to be solved.

Training of backpropagation networks (without internal feedback) is a process of **supervised learning**, relying on the **error-correction learning** method in which the desired, *i.e.* a given, output pattern is expected to be matched by the final output pattern of the network within a specified accuracy. This is to be achieved by adjusting the network weights according to a parameter tuning algorithm, traditionally performed by a backpropagation algorithm that is considered as a generalization of the delta rule.

3.3.2 Radial Basis Function Networks

The idea of function approximation using **localized basis functions** is the result of the research work done by Bashkirov *et al.* (1964) and by Aizerman, Braverman and Rozenoer (1964) on the **potential function approach** to pattern recognition. Moody and Darken (1989) used this idea to implement a fast learning neural network structure with locally tuned processing units. Similarly, Broomhead and Lowe (1988) have described an approach to local functional approximation based on adaptive function interpolation. This has found a remarkable resonance within the researchers working on function approximation using **radial basis functions**,

that is considered to be the birth of a new category of neural networks, named **radial basis function networks**.

The new category of networks was enthusiastically welcomed by the neural network society because the new networks have demonstrated the improved capability of solving pattern separation and classification problems. Backpropagation networks, in spite of their universal approximation capability, fail to be reliable pattern classifiers. This is because during the training phase multilayer perceptron networks build strictly **separating hyperplanes** that exactly classify the given examples, so that the new, unknown examples are randomly classified. This is a consequence of using the sigmoidal function as the network activation function with its resemblance to the unit step function, which is a global function. Also, the sigmoidal function, since it belongs to the set of **monotonic basis functions**, has a slowly decaying behaviour in a large area of its arguments. Therefore, the networks using this kind of activation function can reach a very good overall approximation quality in the large area of arguments; however, they cannot exactly reproduce the function values at the given points. For this one needs **locally restricted basis functions**, such as a **Gaussian function**, **bell-shaped function**, **wavelets** or the **B-spline functions**.

The locally restricted functions can be centred with the exact values at some selected argument values. The function values around these selected argument positions can decay relatively fast, controlled by the approximation algorithm. Powel (1988) suggested that the locally restricted basis functions should generally have the form

$$F(x) = \sum_{i=1}^n w_i \varphi(\|x - x_i\|),$$

where $\varphi(\|x - x_i\|)$ is a set of nonlinear functions relying on the Euclidean distance $\|x - x_i\|$. Moody and Darken (1989) selected for their radial basis function networks the exponential activation function

$$F_i = \exp\left(-\frac{\|x_i - c_i\|^2}{\sigma_i^2}\right),$$

which is similar to the Gaussian density function centred at c_i . The function spread σ_i around the centre determines the ratio of the function decay with its distance from the centre.

The common configuration of an RBF network firmly consists of three layers (Figure 3.5): the input layer, the hidden layer, and the output layer. In the neurons of hidden layer the activation functions are placed. The input layer of the network is directly connected with the hidden layer of the network, so that only the connections between the hidden layer and the output layer are weighted. As a consequence, the training procedure here is entirely different from that in the backpropagation networks. The most important issue here is the selection for each

neuron in the hidden layer the centre c_i and the spread around the centre σ_i ; this is mostly done using the ***k-means clustering algorithm***, which is capable of determining the optimal position of centres. In addition, the value of the ***spread parameter*** σ_i should be selected small enough in order to restrict the basis function spreading, but also large enough to enable a smooth network output through the joint effect with the neighbouring functions.

The network training process mainly includes two training phases:

- ***initialization*** of RBF centres, for instance using ***unsupervised clustering*** methods (Moody and Darken, 1989), ***linear vector quantization*** (Schwenker *et al*, 1994), or ***decision trees*** (Kubat, 1998)
- ***output weight training*** of the RBF using an adaptive algorithm to estimate its appropriate values.

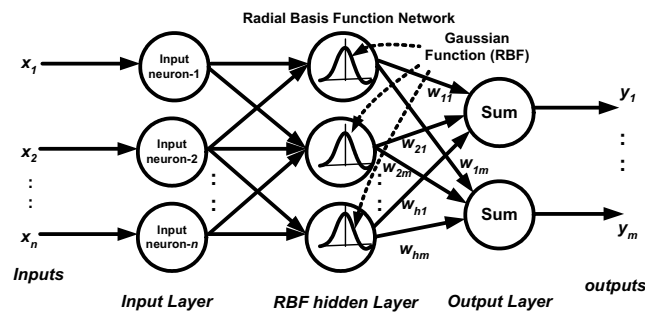


Figure 3.5. Configuration of an RBF network

In some cases, it is recommended to add a third training phase (Schwenker *et al*. 2001) in which the entire network architecture is adjusted using an optimization method.

3.3.3 Recurrent Networks

Research in the area of sequential and time-varying patterns recognition has created the need for time-dependent nonlinear input-output mapping using neural networks. To achieve this extended network capability, the ***time dimension*** has to be introduced into the network topology, for instance by introducing ***short-term memory features***, that would enable network to perform time-dependent mappings. Elman (1990) proposed a kind of ***globally feedforward, locally recurrent network*** using the ***context nodes*** as the principal processing elements of the network. Such nodes have also been the principal processing elements of the network proposed by Jordan (1986) for providing the networks with the dynamic memory. Both Jordan and Elman networks belong to the category of ***simple recurrent networks***.

An Elman network (Figure 3.6) is a **four-layer network** made out of input layer, hidden layer, output layer and the **context layer**, the nodes of which are the one-step delay elements embedded into the local feedback paths. In the network, the neighbouring layers are interconnected by adjustable weights.

Originally, Elman proposed his simple recurrent network for speech processing. Nevertheless, owing to its eminent dynamic characteristics the network was widely accepted for systems identification and control (Sastry *et al.*, 1994). This was followed by applications in function approximation and in time series prediction.

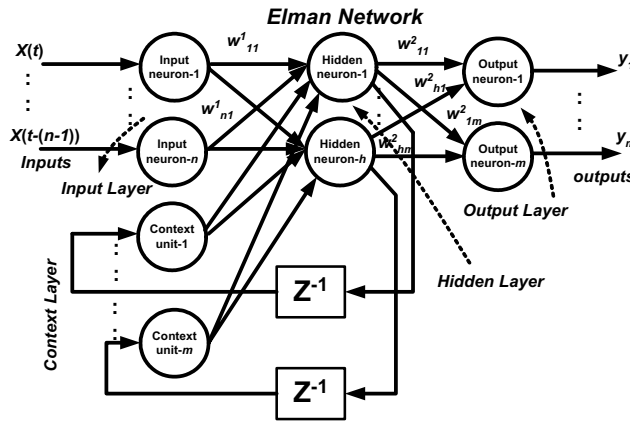


Figure 3.6. Configuration of the Elman network

Independently, Hopfield (1982) reported to the US National Academy of Sciences about neural networks with emergent collective computational abilities. In his report, Hopfield (1984) presented the neurons with graded response and their collective computational properties. He also presented some applications in neurobiology and described an electric circuit that closely reflected the dynamic behaviour of neurons, which is known as the **Hopfield network** (see Figure 3.7).

The Hopfield network is a single-layer fully interconnected recurrent network with a symmetric weight matrix having the elements $w_{ij} = w_{ji}$ and zero diagonal elements. As shown in Figure 3.7, the output of each neuron is fed back via a delay unit to the inputs of all neurons of the layer, except to its own input. This provides the network with some **auto-associative capabilities**: the network can store by learning, following the **Hebbian law** or the **delta rule**, a number of prototype patterns called **fixed-point attractors** in the locations determined by the weight matrix. The patterns stored can then be retrieved by associative recalls. On request to recall any of patterns stored, the network repeatedly feeds the output signals back to the neuron inputs until it reaches its stable state.

The recall capability of recurrent networks of retaining the past events and of using them in further computations is the advantage that the feedforward networks

do not have. This capability enables the networks to generate time-variable outputs in response to the static inputs.

Because of incorporating internal feedback loops, the critical issue of recurrent networks is their stability, determined by the time behaviour of the network **energy function**. For a **binary Hopfield net** with a symmetric weights matrix this function is defined as

$$E = -\frac{i}{2} \sum_{i=1}^n \sum_{j=1}^n w_{ij} x_i \dot{x}_j .$$

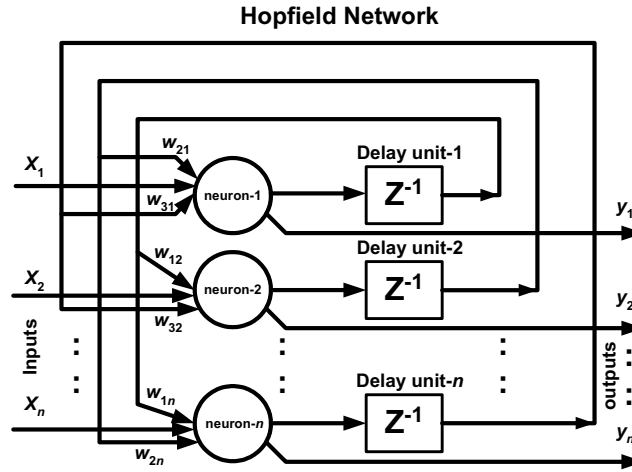


Figure 3.7. Configuration of a Hopfield network

In the case of a stable network this function must decrease with time and ultimately reach its minimum, or it's value remains constant. The minima reached are usually local minima because there are a number of states corresponding to fixed-point actuators or stored patterns to which the network must converge. Each finally reached state of the network has its associated energy defined above.

For the generalized form of binary Hopfield network, in which the sigmoid function

$$f(x) = \frac{1}{1 + e^{-x}}$$

is used, the changes in time are continuously described following the equation

$$\kappa \frac{du_j}{dt} = \sum_i w_{ji} y_i - \frac{u_j}{D_j} + U_j ,$$

where κ is a constant positive value, y_i is the output value of the unit i , D_j is the factor controlling the sigmoid decay resistance, and U_j is the external input to the unit j . The resulting energy function in this case is defined by

$$E = -\frac{1}{2} \sum_i \sum_j w_{ij} u_i u_j - \sum_i u_i U_i$$

Network stability, as proven by Hopfield (1982), is generally guaranteed by the symmetric network structure.

For the training of recurrent networks, Rumelhart *et al.* (1986) proposed a general framework similar to that used for training feedforward networks, called **backpropagation through time**. The algorithm is obtained by unfolding the temporal operation of the network into a layered feedforward growing with each time step. This, however, is not always satisfactory. Williams and Zipser (1988) presented a learning algorithm for continuously running **fully connected recurrent neural networks** (Figure 3.9) that adjusts the network weights in real time, *i.e.* during the operational phase of the network. The proposed learning algorithm is known as a **real-time recurrent learning algorithm**.

There are two basic learning paradigms for recurrent networks:

- **fixed-point learning**, through which the network reaches the prescribed steady state in which a **static input pattern** should be stored
- **trajectory learning**, through which a network learns to follow a trajectory or a sequence of samples over time, which is valuable for **temporal pattern recognition**, **multistep prediction**, and **systems control**.

For trajectory learning, both the backpropagation through time and the real-time recurrent learning are appropriate. From the mathematical point of view, using the backpropagation through time we turn the recurrent network - by unfolding the temporal operation - into a layered feedforward network, the structure of which at every time step grows by one layer.

Almeida (1987) and Pineda (1987) have presented a method to train the recurrent networks of any architecture by backpropagation. Under the assumption that the network outputs strictly depend only on present and not on the past input values, Almeida derived the **generalized backpropagation rule** for this type of network, and addressed the problem of network stability using the energy function formulated by Hopfield (1982). Pineda (1987), however, directly addressed the problem of generalization of the backpropagation training algorithm and its extension to recurrent neural networks. Hertz *et al.* (1991), based on the results of this work, have worked out a backpropagation algorithm for networks, the activation function of which obeys the **evolutionary law**

$$\tau \frac{dv_i}{dt} = -v_i + g(\sum_j w_{ij} v_j + x_i),$$

that was formulated by Cohen and Grossberg (1983). In the above equation, τ is the time constant and x_i is the external input to the unit i . Solving this equation and defining the network equilibrium state for the unit k of the network

$$h_k = \sum_j w_{kj} v_j + x_k ,$$

the network should relax and ultimately reach the value y_k . Thereafter, the weights are updated using the gradient descent method by

$$\Delta w_{lk} = \alpha v_l g(h_k) y_k ,$$

where v_l and h_k are the equilibrium values of unit l and the equilibrium net input to the unit k respectively, and y_k is the equilibrium value of the *matrix inverse unit*.

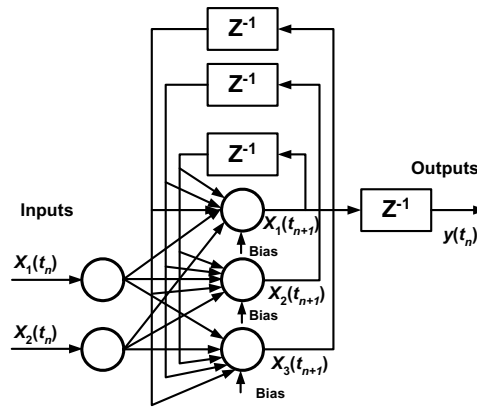


Figure 3.8. Fully connected recurrent neural network

A particular type of recurrent networks that do not obey the restrictions of the Hopfield networks are the *dynamic recurrent networks*, proposed for representation of systems whose internal state changes with time. They are particularly appropriate for modelling of nonlinear dynamic systems, generally defined by the *state-space equations*

$$\begin{aligned} X(k+1) &= f(x(k), u(k)) \\ Y(k) &= Cx(k). \end{aligned}$$

3.3.4 Counterpropagation Networks

A **counterpropagation network**, as proposed by Hecht-Nielsen (1987a, 1988), is a combination of a Kohonen's **self-organizing map** of Grossberg's learning. The combination of two neuro-concepts provides the new network with properties that are not available in either one of them. For instance, the network can for a given set of input-output vector pairs $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ learn the functional relationship $y = f(x)$ between the input vector $x = (x_1, x_2, \dots, x_n)$ and the output vector $y = (y_1, y_2, \dots, y_n)$. If the inverse of the function $f(x)$ exists, then the network can also generate the inverse functional relationship

$$x = f^{-1}(y).$$

When adequately trained, the counterpropagation network can serve as a **bi-directional associative memory**, useful for pattern mapping and classification, analysis of statistical data, data compression and, above all, for function approximation.

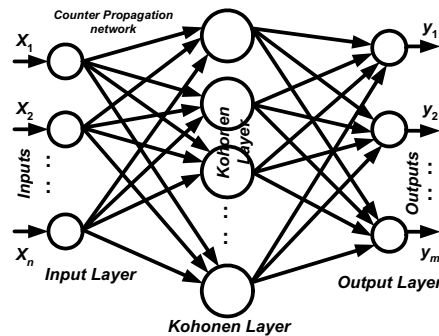


Figure 3.9. Configuration of a counterpropagation network

The overall configuration of a counterpropagation network is presented in Figure 3.9. It is a three-layer network configuration that includes the input layer, the Kohonen **competitive layer** as hidden layer, and the **Grossberg output layer**. The hidden layer performs the key mapping operations in a competitive **winner-takes-all fashion**. As a consequence, each given particular input vector $(x_{1p}, x_{2p}, \dots, x_{np})$ activates only a single neuron in the Kohonen layer, leaving all other neurons of the layer inactive (see Figure 3.10). Once the competition process is terminated, a set of weights connecting the activated neuron with the neurons of the output layer defines the output of the activated neuron (say p) as the sum of products

$$y_p = \sum_{i=1}^n w_{ji} x_i ,$$

where n is the number of input layer neurons connected with the activated neuron. Using the set of weights learnt and stored, the network is capable of recognizing the pattern once learnt and the patterns in its neighbourhoods because similar inputs will activate the same Kohonen neuron.

After locating the Kohonen neuron, we turn to the Grossberg layer, *i.e.* the output layer of the network, and train it. To produce the desired mapping of the pattern at the network output using the output of the activated Kohonen neuron, all we need is to connect this neuron with each neuron in the Grossberg layer using the corresponding weights. As a result, a star connection between the Kohonen neuron and the network output, known as **Grossberg's outstar**, builds the output vector $(y_{1p}, y_{2p}, \dots, y_{mp})$, as shown in Figure 3.10.

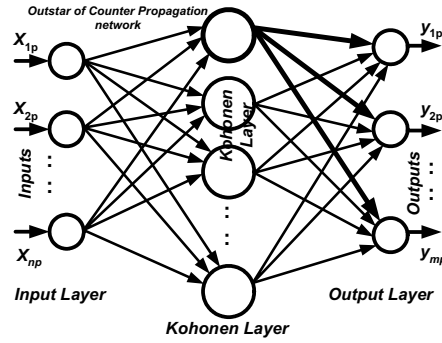


Figure 3.10. Outstar of counterpropagation network

The input vectors of a counterpropagation network should generally be normalized, *i.e.* they should satisfy the relation

$$\|x\| = 1 .$$

The normalization can be carried out by decreasing or increasing the vector length to be on the unit sphere using the relation

$$\bar{x} = \frac{x}{\|x\|} .$$

The question that remains is how to initialize the weight vectors before the network training starts. The preference of taking the randomized weight vectors has not always given reliable learning results. It has in some cases even created serious solution problems. The way out was found in using the *convex combination*

method by taking for all the weight vectors the same value $1/\sqrt{n}$, where n is the dimension of weight vectors.

3.3.5 Probabilistic Neural Networks

The idea of probabilistic neural networks was born in the late 1980s at Lockheed Palo Alto Research Centre, where the problem of special patterns classification into submarine/non-submarine classes was to be solved. Specht (1988) suggested using a newly elaborated special kind of neural network, the *probabilistic neural networks*. To solve the classification problem, the new type of network had to operate in parallel with a *polynomial ADALINE* (Specht, 1990).

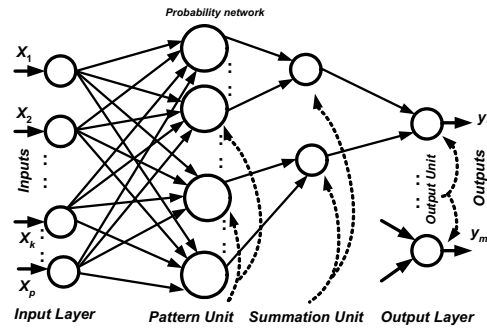


Figure 3.11. Architecture of a probability network

Supposing that P_1, P_2, \dots, P_m are the *a priori probabilities* for the vector \mathbf{x} to belong to a corresponding category, and denoting by L_i the merit of classification loss for the category i , the Bayesian decision rules $P_i L_i p_i$, for $i = 1, 2, \dots, m$, can help determine the largest product value. In case that, say, $P_i L_i p_i \geq P_j L_j p_j$ holds, the input vector \mathbf{x} is assigned to the category i . In this case the *decision boundary* for the above decision, that can be a nonlinear *decision surface* of arbitrary complexity, is defined by

$$P_i = \frac{P_j L_j p_j}{L_i P_i}.$$

The structure of probabilistic networks is similar to that of backpropagation networks, but the two types of network have different activation functions. In probabilistic networks the sigmoid function is replaced by a class of exponential functions (Specht, 1988). Also, the probabilistic networks require only a single training pass, in order that - with the growing number of training examples - the decision surfaces finally reach the Bayes-optimal decision boundaries (Specht, 1990). This is achieved by modelling the well-known Bayesian classifier that

follows the strategy of minimization of the expected classification risk. The strategy can be explained in terms of an n -dimensional input vector \mathbf{x} belonging to one of m possible classes with the *probability density functions*

$$p_1(x), p_2(x), \dots, p_m(x).$$

The architecture of a probabilistic network, shown in Figure 3.11, consists of an input layer followed by three computational layers. It has a striking similarity with a multilayer perceptron network. The network is capable of discriminating two pattern categories represented through the positive and negative output signals. To extend the network capability of multiplying discrimination, additional network outputs and the corresponding number of summation units are required.

The input layer of a probabilistic network is simply a distribution layer that provides the normalized input signal values to all classifying networks that make up a multiple classes classifier. The subsequent layer consists of a number of *pattern units*, fully connected to the input layer through adjustable weights that correspond to the number of categories to be classified. Each pattern unit forms the product of the input vector \mathbf{x} with the weight vector \mathbf{w} . The product value, before being led to the corresponding *summation unit*, undergoes the initial nonlinear operation

$$F(xw_i) = e^{\frac{(xw_i - 1)}{\sigma^2}}.$$

However, since both the input pattern and the weighting vectors are normalized to the unit length, the last relation is to be rewritten as

$$F(xw_i) = e^{\frac{\sum_{j=1}^n (x_j - w_{ij})^2}{2\sigma^2}}.$$

The summation units finally add the signals coming from the pattern units corresponding to the category selected for the current training pattern.

3.4 Network Training Methods

We now turn our attention to some training aspects of neural networks, particularly to the aspects of training process acceleration and training process results. Our primary interests are the *supervised learning algorithms*, the most frequently used in real applications, such as the *backpropagation training algorithm*, also known as the *generalized delta rule*.

The backpropagation algorithm was initially developed by Paul Werbos in 1971 but it remained almost unknown until it was “rediscovered” by Parker in 1982. The algorithm, however, became widely popular after being clearly formulated by Rumelhart *et al.* (1986), which was a triggering moment for

intensive use of multilayer perceptron networks in many simulated engineering applications. The real-life application had at that time to be “postponed” due to the lack of a suitable neuro-technology. In the 1990s Rumelhart put much effort into popularizing the training algorithm among the neural network scientific community. Presently, the backpropagation algorithm is also used (in slightly modified form) for training of other categories of neural networks.

In the following, we will confine our discussion mainly to multilayer perceptron networks. As mentioned earlier, this kind of networks, based on given training samples or input-output patterns, implements nonlinear mapping of functions that is applicable to function approximation, pattern classification, signal analysis, *etc.* In the process of training, the network learns through adaptation of **synaptic weights** in such a way that the discrepancy between the given pattern and the corresponding actual pattern at network output is minimized. Because the synaptic adaptation mostly follows the **gradient descent law** of parameter tuning, the backpropagation training algorithm is considered as the search algorithm of **unconstrained minimization** of a suitably constructed error function at network output.

In order to illustrate the basic concept of the backpropagation algorithm, let us consider its application to the training of a single neuron located in the output layer of a multilayer perceptron (see Figure 3.12). In addition, let us suppose that as the nonlinear activation function the hyperbolic tangent function

$$y = f(u_j) = \tanh(\gamma u_j) = \frac{1 - \exp(-\gamma u_j)}{1 + \exp(-\gamma u_j)} \quad (3.1)$$

is chosen, where

$$u_j = \sum_{i=1}^n w_i x_i + \theta_j, \quad \gamma > 0. \quad (3.2)$$

Furthermore, x_i is the i th input with corresponding interconnecting weight w_i to the neuron and θ_j is the bias input to the same neuron. Typically, all neurons in a particular layer of the multilayer perceptron have the same activation function. The aim of the learning algorithm is to minimize the instantaneous squared error function of the network output

$$S_j = 0.5(d_j - y_j)^2 = 0.5(e_j)^2, \quad (3.3)$$

defined as the square of the difference $(d_j - y_j)$ between the desired output signal and the actual output signal of the network, by modifying the synaptic weights w_i . The minimization process in parameter tuning steps Δw_i is based on the steepest descent gradient rule

$$\Delta w_i = -\eta \frac{\partial S_j}{\partial w_i} \quad (3.4)$$

where η is a positive learning parameter determining the speed of convergence to the minimum.

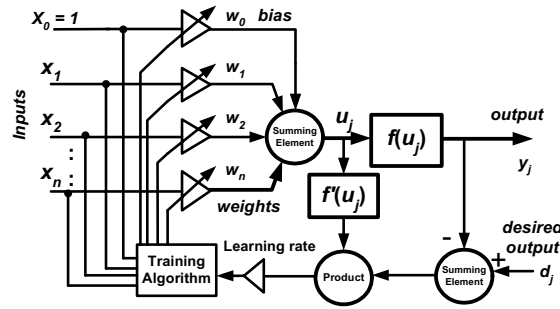


Figure 3.12. Backpropagation training implementation for a single neuron

Now, taking into account that from (3.3) follows:

$$e_j = (d_j - y_j) = (d_j - f(u_j)), \quad (3.5)$$

where

$$u_j = \sum_{i=0}^n w_i x_i.$$

By applying the chain rule

$$\Delta w_i = -\eta \frac{\partial S_j}{\partial e_j} \cdot \frac{\partial e_j}{\partial w_i} \quad (3.6)$$

to Equation (3.5) we get

$$\Delta w_i = -\eta e_j \cdot \frac{\partial e_j}{\partial w_i} = -\eta e_j \cdot \frac{\partial e_j}{\partial u_j} \cdot \frac{\partial u_j}{\partial w_i} \quad (3.7)$$

This can further be transformed to

$$\Delta w = \eta e_j \cdot \frac{\partial f(u_j)}{\partial u_j} \cdot x_i = \eta e_j \cdot f'(u_j) \cdot x_i = \eta \delta_j \cdot x_i$$

where δ_j can be expressed as

$$\delta_j = e_j f'(u_j) = -\frac{\partial S_j}{\partial u_j}. \quad (3.8)$$

The derivation $f'(u_j)$ of the selected activation function (3.1) is

$$f'(u_j) = \frac{\partial f(u_j)}{\partial u_j} = \gamma [1 - \tanh^2(\gamma u_j)] = \gamma [1 - y_j^2], \quad (3.9)$$

and the corresponding weight updates (3.7)

$$\Delta w_i = \eta \gamma e_j \cdot (1 - y_j^2) \cdot x_i, \quad (3.10)$$

with $\eta \gamma > 0$.

Note that the weight update stabilizes if y_j approaches -1 or $+1$, since the partial derivative $\partial y_j / \partial u_j$, equal to $\gamma(1 - y_j^2)$, reaches its maximum for $y_j = 0$ and its minima for ± 1 . However, if the sigmoidal activation function is used and if it is unipolar, described by

$$y_j = f(u_j) = \frac{1}{1 + \exp(-\gamma y_j)}, \quad (3.11)$$

then

$$f'(u_j) = \frac{\partial f(u_j)}{\partial u_j} = \gamma y_j (1 - y_j). \quad (3.12)$$

Therefore, the weight increment takes the form

$$\Delta w_i = \eta \gamma e_j \cdot y_j (1 - y_j) \cdot x_i. \quad (3.13)$$

It should also be noted that in this case the partial derivative $\partial y_j / \partial u_j$ reaches its maximum for $y_j = 0.5$ and, since $0 \leq y_j \leq 1$, it approaches its minimum as the output y_j approaches the value zero or the value one.

The synaptic weights are usually changed incrementally and the neuron gradually converges to a set of weights which solve the specific problem. Therefore, the implementation of the backpropagation algorithm requires an accurate realization of the ***sigmoid activation function*** and of its derivative.

The backpropagation algorithm described can also be extended to train multilayer perceptron networks.

3.4.1 Accelerated Backpropagation Algorithm

The backpropagation algorithm generally suffers from a relatively slow convergence and with the possibility of being trapped at a local minimum. Also, it can be accompanied by possible oscillation around the located minimum value. This may restrict its practical application in many cases. Therefore, such unwanted drawbacks of the algorithm have to be removed, or at least reduced. For instance, the speed of algorithm convergence can be accelerated:

- by selection of the best initial weights instead of taking the ones that are generated at random
- through adequate preprocessing of training data, *e.g.* by employing the feature extraction algorithms or some data projection methods
- by improving the optimization algorithm to be used.

Numerous heuristic optimization algorithms have been proposed for speed acceleration; unfortunately, they are generally computationally involved and time exhausting. In the following, only two of the most efficient are briefly reviewed:

- adaptation of learning rate
- using a momentum term.

It is usually assumed that the learning rate of the algorithm is fixed and uniform for all weights during the training iterations. In order to prevent parasitic oscillations and to ensure the convergence to the global minimum, the learning rate must be kept as small as possible. However, a very small value of learning rate slows down the convergence speed of algorithm considerably. On the other hand, a large value of the learning rate results in an unstable learning process. Therefore, the learning rate has to be optimally set between the two extreme values of learning rate, *e.g.* by using the ***adaptive learning rate***, and in this way the training time can be considerably reduced. Similarly, the speed up of convergence can be achieved by extending the training algorithm by a ***momentum term*** (Kröse and Smagt, 1996). In this case the learning rate can be kept at each iteration step as large as possible within the admitted values, while maintaining the learning process stable.

One of the simplest heuristic approaches of learning rate tuning is to increase the learning rate slightly (typically by 5%) in an iteration step if the new value of the output error (sum squared error) function S is smaller than the previous

iteration step. On the other hand, if the new value of the error function exceeds the value of the previous one, then the learning rate should be decreased by approximately 30%, and in the latter case the new weight updates and the error function are discarded, *i.e.* in this case we set weight update as

$$\Delta w_{ij}(k+1) = 0,$$

and that leads to weights in $(k+1)$ th iteration as identical as $(k-1)$ th, *i.e.*

$$w_{ij}(k+1) = w_{ij}(k-1).$$

After starting with a small learning rate, the approach will behave as follows:

$$\begin{aligned} \eta^{(k)} &= a\eta^{(k-1)}, \text{ for } S(w(k)) < S(w(k-1)), \\ \eta^{(k)} &= b\eta^{(k-1)}, \text{ for } S(w(k)) \geq k_0 S(w(k-1)), \\ \eta^{(k)} &= \eta^{(k-1)}, \text{ otherwise} \end{aligned} \quad (3.14)$$

with $a = 1.05$, $b = 0.7$ and $k_0 = 1.04$ being typical values (Vogl *et al.* 1988; Cichocki and Unbehauen, 1993).

In some training applications not all the training patterns are available before the learning starts. In such situations an on-line approach has to be used. Schmidhuber (1989) proposed the simple global updates of the learning rate for each training pattern as

$$\Delta w_{ij}(k) = -\eta^{(k)} \frac{\partial S_p}{\partial w_{ij}}, \quad (3.15)$$

with

$$\eta^{(k)} = \min \left\{ \frac{S_p - S_0}{\|\nabla S_p\|_2^2}, \eta_{(\max)} \right\}, \quad (3.16)$$

where the index $\eta_{(\max)}$ indicates the maximum learning rate (typically $\eta_{(\max)} = 20$) and S_0 is a small offset error function (typically $0.01 \leq S_0 \leq 0.1$).

Various suggestions have been made for practical use of both adaptable learning rate and the momentum term, with the best known being the conjugate gradient algorithm (Johansson *et al.*, 1992). Alternatively, the second-order derivative-based Levenberg-Marquardt algorithm (Hagan and Menhaj, 1994), proposed for accelerated minimization of the cost function, is preferably used for accelerated neural networks training. The key idea of the algorithm is to use a

search vector P_k to calculate the parameter value W_{k+1} , based on a current value W_k as

$$W_{k+1} = W_k + \alpha_k P_k, \quad (3.17)$$

where α_k is a scalar value. The search vector P_k is to be chosen so that the relation $V(W_{k+1}) < V(W_k)$ holds, where $V(W)$ is the performance index of the network, generally a sum square error function.

Now, considering the Taylor series expansion of $V(W_{k+1})$ at point W_k

$$V(W_{k+1}) = V(W_k + \alpha_k P_k) \approx V(W_k) + \alpha_k \nabla V(W_k)^T P_k. \quad (3.18)$$

it is obvious that, in order for the cost function V to decrease and for a positive value of α_k , the second term of (3.18) must be negative. This will be the case if the steepest descent condition

$$W_{k+1} = W_k - \alpha_k \nabla V(W_k) \quad (3.19)$$

is met. However, the steepest descent method, as discussed earlier, when used in its original form, exhibits some drawbacks that need to be eliminated for its practical use. To overcome this, the approximation of the objective function in the immediate neighbourhood of a strong minimum by a quadratic function with positive definite Hessian matrix or by using Newton's method for pursuing the minimization problem is preferred.

Let us now consider the Taylor series expansion

$$V(W_{k+1}) \approx V(W_k) + \nabla V(W_k)^T \Delta W_k + \frac{1}{2} \Delta W_k^T \nabla^2 V(W_k) \Delta W_k \quad (3.20)$$

where $\nabla^2 V(W_k)$ is the Hessian matrix and $\Delta W_k = \alpha_k P_k$. If the gradient of the truncated Taylor series expansion (3.20) is taken with respect to ΔW_k and set to zero (since we are looking for the minimum of the cost function), it follows that

$$\Delta W_k = -[\nabla^2 V(W_k)]^{-1} \nabla V(W_k). \quad (3.21)$$

This reduces the Newton method to

$$W_{k+1} = W_k - [\nabla^2 V(W_k)]^{-1} \nabla V(W_k). \quad (3.22)$$

Direct practical use of this method, however, is hampered by the need for Hessian matrix calculation, whose elements are the second derivatives of the performance index with respect to the parameter vector. To overcome this obstacle, the first and the second derivatives of the performance index

$$V(W_k) = \sum_{i=1}^N e_i^2(w_k) = e^T(w_k)e(w_k) \quad (3.23)$$

are built and expressed as

$$\nabla V(w_k) = J^T(w_k)e(w_k) \quad (3.24)$$

and

$$\nabla^2 V(w_k) = J^T(w_k)J(w_k) + \sum_{i=1}^N e_i(w_k)\nabla^2 e_i(w_k), \quad (3.25)$$

where $J(w_k)$ is the Jacobian matrix and

$$e(w_k) = T - Y(w_k), \quad (3.26)$$

with the target vector T and the actual output of the neural network $Y(w_k)$.

The Gauss-Newton modification of the method assumes that the second term in the right-hand side expression of (3.25) is zero. Therefore, applying the former assumption (3.22) yields the Gauss-Newton method as

$$W_{k+1} = W_k - [J^T(w_k)J(w_k)]^{-1} J^T(w_k)e(w_k), \quad (3.27)$$

An additional difficulty appears here with when the Hessian matrix is not positive definite, *i.e.* its inverse does not exist. In this case the modification of the Hessian matrix

$$G = \nabla^2 V(w_k) + \mu I \quad (3.28)$$

should be considered. Suppose that the eigen-values and the eigen-vectors of $\nabla^2 V(w_k)$ are the sets $\{\lambda_i\}$ and $\{z_i\}$ respectively. Multiplying both sides of (3.28) by z_i we have

$$G z_i = \nabla^2 V(w_k) z_i + \mu I z_i = \lambda_i z_i + \mu z_i \quad (3.29)$$

$$G z_i = (\lambda_i + \mu) z_i \quad (3.30)$$

Therefore, the eigen-values and eigen-vectors of G are $\{\lambda_i + \mu\}$ and $\{z_i\}$ respectively. G can be made positive definite by increasing μ until $\lambda_i + \mu > 0$ for all i .

Therefore, the Levenberg-Marquardt modification to Gauss-Newton method is

$$W_{k+1} = W_k - [J^T(w_k)J(w_k) + \mu I]^{-1} J^T(w_k)e(w_k) \quad (3.31)$$

whereby the parameter μ is multiplied by some factor β whenever a step would result in an increased value of $V(w_k)$. When a step reduces this value, μ is divided by β . Notice that when μ is large the algorithm becomes steepest descent with the step size approximately $1/\mu$. On the other hand, for small μ the algorithm becomes Gauss-Newtonian.

Obviously, the calculation of the Jacobian matrix is the key step in applying this algorithm. At first, all the adjustable parameters of the network should be arranged in one column vector w_k . For a neural network mapping problem the terms in the Jacobian matrix can be computed by simple modification to the backpropagation algorithm (Hagan and Menhaj, 1994). In the standard backpropagation version, partial derivatives of the performance function with respect to the adjustable parameters are needed, while in Levenberg-Marquardt algorithm the derivative of the error is needed for the Jacobian matrix. This means that the Jacobian matrix can be calculated using the sensitivity term of the performance index derived in the standard backpropagation algorithm with one modification at the final layer, *i.e.* by dropping the error term (Hagan and Menhaj, 1994). The Jacobian matrix computation for a neuro-fuzzy network is described in Chapter 6.

The algorithm described above can easily be extended to train the multilayer perceptron networks.

3.5 Forecasting Methodology

Forecasting methodology is generally understood as a collection of approaches, methods, and tools for collection of time series data to be used for forecast or prediction of future values of the time series, based on past values. The forecasting methodology includes the following operational steps:

- **data preparation** for forecasting, *i.e.* acquisition, preprocessing, normalization, and structuring of data, determination of training and test data sets, and the like
- **network architecture determination**, *i.e.* selection of the type of network to be used for forecasting, determination of number of network input and output nodes, number of layers, the number of neurons within the layers, determination of interconnections between the neurons, selection of neuron activation functions, *etc.*

- design of **network training strategy**, *i.e.* selection of training algorithm, performance index, and the training monitoring approach
- **overall evaluation** of forecasting results using fresh observation data sets.

3.5.1 Data Preparation for Forecasting

Data used for analysis and forecasting of time series are generally collected by observations or by measurements. In engineering, of major interest is the analysis of data obtained by sampling of corresponding sensor signals and forecasting their future behaviour. Therefore, our attention will be primarily focused on forecasting of experimental data taken from **sensing elements** placed within the experimental setups or within the plant automation devices. Here, depending on the nature of signals provided by sensors, two main critical issues are:

- the number of data needed for representative characterization of the observed signal in view of its linearity, stationarity, drift, *etc.*
- the sampling period required for recording the entire frequency spectrum of the sampled signal, but that will still considerably limit the noise frequency spectrum.

In practice, the **preprocessing** of acquired data, because of the presence of noise, drift, and sensor inaccuracy, represents a trial-and-error procedure. In the preprocessing phase it should also be made clear whether data filtering, smoothing, *etc.* are needed, or whether mathematical transformation of data will facilitate the learning process of the network within its training and/or reduce the network training time.

Data normalization is a process of final data preparation for their direct use for network training. It includes the normalization of preprocessed data from their natural range to the network's operating range, so that the normalized data are strictly shaped to meet the requirements of the network input layer and are adapted to the nonlinearities of the neurons, so that their outputs should not cross the saturation limits.

In practice, the simplest normalization

$$x_{ni} = \frac{x_i}{x_{\max}}$$

and the linear normalization

$$x_{ni} = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

are most frequently used. Moreover, instead of linear normalization, **nonlinear scaling** or **logarithmic scaling** of input signals is used to moderate the possible nonlinearity problems during the network training. For instance, logarithmic transformation can squeeze the scale in the region of large data values, and

exponential scaling can expand the scale in the region of small data values, *etc.* But by far the most critical data preparation issue here is the risk of possible loss of critical information present within the acquired data.

Structuring of data is needed when preparing the mutually related input and output data pairs to be used in supervised learning and/or when preparing multivariate data in general. In the case of training the networks for forecasting purposes, the next value x_{t+1} of the univariate time series is related to the past values of the time series up to the present value x_t . In the next training step the value x_{t+2} is related to the past values of the time series up to the value x_{t+1} , *etc.*

Before structuring the data of a multivariate time series for training of a network forecaster, the fact should be recalled that this kind of time series is a set of simultaneously built multiple time series with the values of each individual time series being related to the corresponding values of other time series. This is because the multivariate time series are built by simultaneous observation of two or more processes, so that the resulting observation across all the individual samplings at a certain time builds an **observation vector**

$$x_i = [x_{i1} x_{i2} \dots x_{im}].$$

Thus, the resulting multiple time series in fact represents a set of observation vectors x_i , $i = 1, 2, \dots, m$, building up the **observation matrix**

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix},$$

in which the time series of individual processes are represented through the corresponding matrix columns.

A **training set** is used to teach the network to behave as a forecaster and the **test set** is used, after the training, to test its forecasting capability. Both data sets are to be built from the entire collected data set. Unfortunately, no selection guide is available for splitting the prepared data set into two subsets. The recommendations range from a 90% to 10% ratio, up to a 50% to 50% ratio. Haykin (1995) advocated that the numbers of patterns N in the training set required to classify the test examples with an error of ε should approximately be

$$N = \frac{W}{\varepsilon},$$

where W is the number of weights in the network.

Yet, whatever ratio is selected, attention should be paid to ensuring that the training data set is large enough to cover all the dominant characteristic features required for reliable network training as a forecaster. The remaining data set can then be used for testing the trained network on the data samples never used in the training. For this reason, it is recommended that the non-training data set should be large enough to enable building of not only the test data set but also the **validation data set** to be used in the overall network evaluation.

3.5.2 Determination of Network Architecture

This is the core task in building the neural network structure optimally adapted to the specific problem the network should optimally solve. In our case it would be the optimal predictor or the optimal forecaster. This task, although being very challenging, is also the most difficult to execute because it requires from the designer much skill and practical experience. Since being a nontrivial task with a multiplicity of possible solutions, there are opinions that this work is more a kind of art than an expert's routine. The issues addressed in the following present the activities to be carried out when developing the network architecture. They include the

- determination of input nodes required
- determination of output nodes
- selection of number of hidden layers
- selection of hidden neurons
- determination of node interconnection pattern
- selection of activity function of neurons.

Determination of the required **number of input nodes** is a relatively easy task, because it depends predominantly on the number of independent variables presented in the data set prepared. As a rule, each independent variable should be represented by its own input node. In the case of input data prepared for forecasting, the number of input nodes is directly determined by the number of **lagged values** to be used for forecasting of the next value

$$x(t+1) = f[x(t), x(t-1), x(t-2), \dots, x(t-n)],$$

as represented in Figure 3.13.

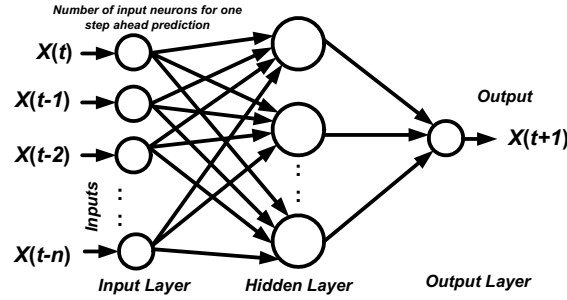


Figure 3.13. Number of input neurons for one-step-ahead forecasting

In practice, the single-step-ahead forecaster is most frequently selected because it is relatively simple and guarantees the most accurate forecasting results. Otherwise, when building a multistep predictor, the determination of the required number of input nodes is a trade-off process in the sense that (following the general inclination) this number should be selected as small as possible but so that it still guarantees good forecasting results, and as large as needed for the extraction of all relevant characteristic features and the *autocorrelation structure* embedded in the training data. To solve this problem optimally, some experimental runs could be of considerable use.

The *number of output nodes*, again, is also a problem-oriented task. In the one-step-ahead forecasting it is apparent that only one output node is sufficient as the forecasting node. Correspondingly, in the case of multistep-ahead forecasting, the number of output nodes should correspond to the forecasting horizon, *i.e.* to the number of forecasts to be simultaneously presented at the network output. Alternatively, a single output node can be used and all the future forecasts required determined in the iterative steps.

In most forecasting applications, only one *hidden layer* is used, although some aberrations are exceptionally needed. The sufficiency of a single layer is covered by the *Kolmogorov's superposition theorem*, which states that any continuous function $f(x)$ – which can also be an n -dimensional vector function $f(x_1, x_2, \dots, x_n)$ – defined on a closed n -dimensional cube, say $[0, 1]^n$, can be represented as

$$f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{2n+1} \psi_i \left(\sum_{j=1}^n \phi_{ji}(x_j) \right),$$

where ψ_i and ϕ_{ji} are continuous, single-variable functions. The functions ψ_i depend on the function to be approximated f and the functions ϕ_{ji} are monotonously increasing functions fixed for a given n .

The theorem, as originally formulated by Kolmogorov, is an existence theorem that does not suggest any particular function to be used for approximation of a

given mapping, so that its relevancy to neural networks was not directly evident. There were even opposite views to the relevance: one opposing the relevancy (Giroso and Poggio, 1989) and another in favour of it. However, it was the refinement of the theorem by Sprecher (1965) that motivated Hecht-Nielsen (1987b) to point out this reliance. He also proposed that the k th processing elements of the hidden layer should have the activation function

$$z_k = \sum_{i=1}^n \lambda^k \varphi(x_i + \varepsilon k) + k,$$

where the real constant λ and the monotonously increasing real continuous function φ depend on n , but are independent of f . Furthermore, the rational constant ε should satisfy the conditions of the **Sprecher theorem** $0 < \varepsilon < \delta$, $\delta > 0$. The activation function of the output layer units should be

$$y_j = \sum_{k=1}^{2n+1} g_j(z_k),$$

where g_j are the real and continuous functions depending on φ and ε . Consequently, as it was shown (Hecht-Nielsen, 1987b), the Kolmogorov's theorem can be implemented exactly by a three-layer feedforward neural network having n input elements in the input layer, $(2n+1)$ processing elements in the hidden layer, and m processing elements in the output layer. This confirms the statement that even a single hidden-layer network is sufficient to reveal all the characteristic features present on the input nodes of the network. Introducing additional hidden layers increases the feature extraction capability of the network at the cost of the significantly extended training and operational time of the forecaster.

Lippmann (1987), in his celebrated paper on neurocomputing, stated clearly that a three-layer perceptron can form arbitrarily complex decision regions and can separate meshed classes, which means that no more than three network layers are needed in perceptron-like feedforward nets. This particularly holds for the networks with one output, as required for one-step-ahead forecasting. Cybenko (1989), finally underlined that the networks never need more than two hidden layers to solve most complex problems. Also, the investigation of neural network capabilities related to their internal structure has proven that two-hidden-layer networks are more prone to fall into bad local minima. DeVilliers and Barnard (1992) even pointed out that both the one- and two-hidden-layer networks perform similarly in all other respects. This can be understood from the comparison of complexity degree of two investigated networks measured by the **Vapnik-Chervonenkis dimension**, as was done by Baum and Hausler (1989).

We now turn to the problem of the **number of hidden neurons** placed within the hidden layer. To determine the optimal number of hidden neurons there is no straight-forward methodology, but some rules of thumb and some suggestions how to do this have been proposed. For instance, in single-hidden-layer networks, it is recommended to take the number of hidden-layer neurons in the neighbourhood of 75% of the number of network inputs, or say between 0.5 and 3 times the number

of network inputs. The *geometric pyramid rule*, on the other hand, suggests assigning

$$N_h = \alpha \sqrt{N_i \times N_o} ,$$

hidden neurons to a single hidden layer, where N_i is the number of network inputs, N_o the number of its outputs, and α is multiplication factor the value of which, depending on the complexity of the problem to be solved, should be selected in the range $0.5 < \alpha < 2$. Baum and Haussler (1989) suggested the number of neurons in the hidden layer be determined as

$$N_h \leq \frac{N_{tr} \times E_{tol}}{N_{dp} + N_o} ,$$

where N_{tr} is the number of training examples, E_{tol} is the error tolerance, N_{dp} is the number of data points per training example, and N_o is the number of output neurons.

Anyhow, the determination of the optimal number of hidden neurons involves trial-and-error experimentation: starting with a number of neurons within the layer to be decided – based on final accuracy of each learning process – to increase or decrease the number of hidden neurons and to start a new learning process. In this way the redundant hidden neurons can be deleted and the neurons needed for optimal performance of the layer added. Here, both starting with a relatively large or small number of neurons is possible, but starting with a large number of neurons bears the risk of long-time computation and of getting trapped in local minima.

Khorasani and Weng (1994) have presented an approach to structural adaptation of feedforward neural networks by neuron pruning, *i.e.* by addition and deletion of hidden neurons based on the activity status of individual neurons during the learning, measured by the variance of the neuron output signal and by the strength of the backpropagated error. This is a proper indication of neuron activity that helps decide which low-activity redundant neurons are to be deleted.

There is also a reliable way to determine the number of hidden neurons using the *Akaike's information criterion* (AIC), originally defined as

$$AIC = (-2) \ln(\text{Maximum likelihood}) + 2(\text{number of adjusted parameters}).$$

The criterion statistically evaluates the goodness of a model by combining the evaluated mean squares error for training data and the number of parameters to be estimated. Seen otherwise, AIC combines a measure of fit and the penalty term to account for model complexity. Its potential application suitability for neural networks model building was recognized by Kurita (1990) and Fogel (1991), who reformulated the original form of the criterion (for statistically independent, normally distributed output errors with zero mean and with constant variance) as

$$AIC = Nk \ln(\sigma^2) + 2K ,$$

where N is the number of training data, k is the number of output units of the network, σ^2 is the *maximum likelihood estimate* of the mean square error for training data and K is the number of model parameters.

The application principle of the AIC is that, if two models have the same mean square error for a training data set, then the smaller sized model should be selected. Alternatively, from a set of possible models, the model with the smallest value of AIC is to be selected (Ishikawa and Moriyama, 1996; Anders and Korn, 1999). This, however, requests a set of models to be built and their parameter estimated before this application principle is used.

Unfortunately, direct application of the AIC to neural networks is rather circumstantial. It is, however, facilitated when using the *network information criterion* (NIC) of Stone (1977)

$$NIC = -\frac{1}{T} \ln(L(\hat{w})) + \frac{\text{tr}[BA^{-1}]}{T} ,$$

which is a generalization of the AIC. The first term in the above expression represents the estimated maximum logarithmic likelihood. The matrices A and B are defined as

$$\begin{aligned} A &\equiv -E[\nabla^2 \ln L_t] \\ B &\equiv E[\nabla \ln L_t \nabla \ln L_t]. \end{aligned}$$

If the classes of models investigated include the true model, then it holds asymptotically that $A = B$ and

$$\text{tr}[BA^{-1}] = \text{tr}[I] = K ,$$

where K is, again, the number of model parameters. In this case the NIC takes the form

$$NIC = -\frac{1}{T} \ln L(\hat{w}) + \frac{K}{T} .$$

This is similar to the AIC, which in this transcription becomes

$$AIC = -\frac{2}{T} \ln L(\hat{w}) + \frac{2K}{T} .$$

Murata *et al.* (1994) used this generalization to determine the number of hidden units required to mimic the system based on input-output examples only. Attention was paid to avoiding possible network *overfitting* by taking a small number of redundant hidden neurons. A large number of hidden layer neurons could, for the given training example, deliver better learning results but, due to the increased network complexity, for some fresh examples could deliver worse results.

What the interconnections of network nodes concerns, **full interconnection** is recommended for initial network configuration, in which the output of each neuron of a layer is connected with the input of each neuron of the subsequent layer. However, in some applications, deviations from full interconnection have also been successful.

For **activation function selection**, there is generally no rich choice left. For backpropagation networks, mostly the

- **sigmoid function**

$$y = \frac{1}{1 + e^{-x}}$$

is selected as an activation function in numerous applications, including time series forecasting. But in some applications the

- **hyperbolic tangent function**

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

has also been used successfully, for instance when solving the problems that rely on learning of deviations from average behaviour (Klimasauskas, 1991)

- **step** and **ramp function** are some additional alternatives favourable for processing binary variables.

In any case, to avoid functional destruction of the neuron, the function selected should be limited at its output, usually between the values -1 and $+1$. Although there are no guidelines for selecting the activation functions in individual network layers and for distributing them within the layers, it is still best to build homogeneous individual layers and for the hidden neurons possibly to use the **sigmoid activation function**. But still, some researchers have successfully used the hyperbolic tangent as an activation function of hidden-layer neurons. Very seldom heterogeneous network layers have been used. For time series forecasting, the general experience has shown that for output neurons the linear activation function delivers the best results. Some theoretical evidence for this has also been given (Rumelhart *et al.*, 1986). It was shown that only for forecasting of time series with trend, output neurons with a nonlinear activation function are required.

3.5.3 Network Training Strategy

Network training is a process in which the network learns to recognize the patterns inherent to the training signals. In network training for time series forecasting all relevant characteristic features embedded in the training data that reflect the autocorrelation structure of the time series should be revealed and learnt. The training is usually carried out in off-line mode using an unconstrained nonlinear minimization algorithm, most frequently a gradient descent method, for tuning the interconnection weights of the network. The objective is to achieve the optimal network behaviour across the training set.

Network learning can generally be executed in *supervised mode* (Hopfield model) or in *unsupervised mode* (Kohonen model). For supervised learning the network is provided by data examples that include the desired output. For unsupervised learning the desired output values are not required because the network finds the adequate output values itself.

The objective of training is to find the set of most suitable values of interconnecting weights through their tuning during the network training. By doing so, the network should still attain the highest *generalization attribute*. This, however, can be aggravated if, instead of the global minimum, only a local minimum has been found. So, particular precautions should be provided to avoid pitting into one of the local minima. Such and similar issues seriously affect the training success, so that some careful considerations are required when preparing the *experiment design* for network training. This includes some decisions to be made concerning the network initialization for training, selection of the appropriate training algorithm, monitoring the training process using an appropriate performance index, formulation of training stopping criteria, etc.

Network initialization is a decision that is to be made before the weights tuning process starts. This is a difficult decision, because the training speed and the total training time required are strongly influenced by this decision. To circumvent this, various suggestions have been made, the most popular being that, in order to prevent neuron saturation and other unpleasant phenomena, some small, randomly distributed parameter values should initially be taken. However, setting all weights initially at the same small value should be avoided because it could possibly hamper the tuning process to start and/or to learn. This definitely does not hold for unsupervised training, like it holds for training of a Kohonen layer of a counterpropagation network, where the competition process takes place. Here, the unique value $1/\sqrt{N}$ is initially taken for all weights, N being the number of network inputs. This is required because by starting the competition process it is advantageous that all competitors have the same initial parameter values for every training run.

Hebb (1949) has proposed the simplest training algorithm for neural networks, known as the *Hebb learning rule*. A neurophysiologist himself, he enunciated the learning principle of natural neurons: if two interconnected neurons at the same time fire, then the strength (weight) of the synapse connecting them increases. Extended to artificial neural networks, this principle states that the common weight

w_{ij} connecting the output of the perceptron i and the input of the perceptron j will increase by an amount

$$\Delta w_{ij} = \eta x_j y_i,$$

where x_j is the output of the perceptron j , y_i the output of the perceptron i , and η is a measure controlling the learning step size (Figure 3.14). Accordingly, the Hebbian learning updating the weights, or the *Hebbian learning rule*, can be expressed as

$$w_{ij}(t+1) = w_{ij}(t) + \eta x_j(t) y_i(t).$$

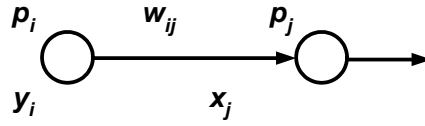


Figure 3.14. Interconnected perceptrons

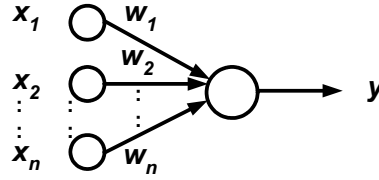


Figure 3.15. Multiple interconnected perceptron

The rule can be generalized and applied to a multiple-input perceptron as

$$w(t+1) = w(t) + \eta x^T w x,$$

where the relation

$$y = \sum_{j=1}^n w_j x_j = w^T x = x^T w$$

is taken into account (Figure 3.15).

Nevertheless, the direct application of the Hebbian rule bears the risk of an endless increase of weight values, which could saturate the output neurons. As a

remedy, an increase in the normalization of weights at every iteration step is necessary. Oja (1982) proposed using for this the normalization relationship

$$w_i(t+1) = \frac{w_i(t) + \eta x_i(t)y(t)}{\sqrt{\sum_i [w_i(t) + \eta x_i(t)y(t)]^2}},$$

derived through modification of the Hebbian rule itself. The modification normalizes the weight vector size to the value 1 by decreasing the values of all other weight vectors if one of its components increases, in this way keeping the total length of the vector constant.

The above rule modification can, for a small value of η and after power expansion, be approximated as

$$w_i(t+1) \cong w_i(t) + \eta y(t)[x_i - y(t)w_i(t)],$$

which is known as Oja's rule.

Yet, the fact that the application of the Hebbian rule is considerably limited to single-layer neural networks, the original version of the **backpropagation algorithm** is favoured for training of multilayer networks. The training is performed off-line in a supervisory learning mode, which is convenient because, in practice, a large number of data are available that have to be processed prior to their application for training. Besides, for forecasting purposes the pairs of related input and output data also have to be built and processed. Finally, the supervisory mode of learning facilitates the implementation of monitoring of training performance and the determination of the training stopping point.

When applying the backpropagation algorithm, which is a typical gradient steepest descent method, decisions have to be made concerning the

- **learning rate**, *i.e.* the step size or the magnitude of weight updating
- **momentum**, which is required for escaping the trapping in local minima.

An appropriate selection of learning rate is particularly important because the steepest descent method suffers from slow convergence and weak robustness. Convergence acceleration by taking a larger learning rate bears the danger of network oscillatory behaviour around the minimum. To avoid this, and still to take a larger learning rate, addition of a momentum parameter was recommended (Rumelhart *et al.*, 1986). By doing this, the original learning step according to the delta rule

$$w(t+1) = w(t) + \eta \varepsilon_p(t) x_p(t)$$

is extended by the momentum term to result in

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_i(t) x_j(t) + \alpha [w_{ij}(t) - w_{ij}(t-1)],$$

where α is the **momentum constant**, with the value $0.5 < \alpha < 0.9$. The added term represents the memorized value of the last increment so that the next weight change keeps approximately the same direction as the last one. This stabilizes the learning convergence.

An alternative way for speeding up and stabilizing the convergence was found in adaptive step size implementation. Silva and Almeida (1990) recommend the following weight update strategy

$$w_{ij}(t) = w_{ij}(t-1) + \eta_{ij}(t) \nabla_{ij} C(t),$$

where $\nabla_{ij}(t)C(t)$ are the gradient components of individual iteration steps

$$\nabla_{ij} C(t) = \sum_{v=1}^N \frac{\partial J(\theta_v)}{\partial w_{ij}},$$

with N as the number of training set samples. In the above updating relation, $\eta_{ij}(t)$ is taken as

$$\begin{aligned} \eta_{ij}(t) &= c_1 \eta_{ij}(t-1) & \text{if } \nabla_{ij} C(t) \nabla_{ij} C(t-1) > 0 \\ \eta_{ij} &= \frac{1}{c_1} \eta_{ij}(t-1) & \text{if } \nabla_{ij} C(t) \nabla_{ij} C(t-1) < 0, \end{aligned}$$

where c_1 is a positive constant.

To circumvent the problem of avoiding the numerous flat and steep regions of the error surface Yu *et al.* (1995) advocated the **dynamic learning rate** to be imbedded into the backpropagation algorithm, based on information delivered by the first and the second derivatives of the objective function with respect to the learning rate. The clue to the proposed strategy is that it avoids the calculation of the values of the second derivative in weight space, using the information collected from the training instead. To bypass the calculation of the pseudo-inverse Hessian matrix that is inherent in second-order optimization methods, the conjugate gradient method is used.

The overwhelming number of upgraded learning algorithms are mainly focused on learning velocity increase and search stability improvement by adding a term containing the derivatives in weight space. But, some improvements of both objectives, namely of learning velocity and of convergence stabilization, are also achievable by manipulating the parameters of the neuron transfer function. Such an updating proposal was made for supervised pattern learning that adaptively manipulates the learning rate by updating neuron internal nonlinearity (Zhou *et al.*, 1991). Using some simulated data sets, it was shown that the updating law proposed increases the learning speed and is very suitable for identification of nonlinear dynamic systems.

3.5.4 Training, Stopping and Evaluation

Originally, the simple principle was accepted that the network should be trained until it has learnt its task. This is certainly difficult to find out, because there is no direct approach how to do this. The general statement that a high enough number of iterations, or training steps, is good enough, in the sense that the network has learnt well enough to be a qualified expert in a specific domain, say in forecasting, does not hold. Thus far, at least theoretically, reaching the global minimum of the objective function is accepted as the *training efficiency merit*, so that by approaching this minimum the error function will steadily decrease until the minimum has been reached. Finding out that there is no further decrease of the error function would then be an indication to stop the training process.

In practice, to find the global minimum, network training can require a number of repeated training trials with various initial weight values. After each training run the training results have to be evaluated and compared with the results achieved in the previous runs, this in order to select the best run. Some researchers have here centred their attention on the problem of *a priori* determination of a maximum number of training runs required for the training. Iyer and Rhinehart (2000) have developed an analytical procedure for determining the desirable lower number of training runs, sufficient - within a certain level of confidence - that the best one is within them. The procedure is based on the *weakest-link-in-the-chain analysis* described by Bethea and Rhinehart (1991).

The authors use the cumulative distribution function for the weakest link in a set of N training, with runs starting with the random initial weight values

$$F_w(a) = 1 - [1 - F_x(a)]^N.$$

This, rearranged as

$$F_x(a) = 1 - [1 - F_w(a)]^{\frac{1}{N}},$$

represents the probability that any single optimization has an error value $x \leq a$. The two relations, simultaneously taken, define the required number of random starts as

$$N = \frac{\ln[1 - F_w(a)]}{\ln[1 - F_x(a)]}.$$

For example, if, at the confidence of 99% level, the best of random starts should result in one of the best 20% values for the sum of squared errors, then the required number of random starts will be

$$N = \frac{\ln(1 - 0.99)}{\ln(1 - 20)} \cong 20.$$

A more recent approach to solving the problems of appropriate training termination departs from some **stopping** criteria. For instance, based on the automated stopping criterion of Natarajan and Rhinehart (1997), Iyer and Rhinehart (2000) take as the stopping criterion the **performance-to-cost ratio** of the network. Assuming that the entire cost of a validation set consisting of N_v data points is $C_v = CN_v$, where C is the cost of single data points, and assuming that the cost of training and test data sets are CN_t and CN_c respectively, then the corresponding performance-to-cost ratio is

$$\rho = \frac{1}{E_{ce}C(N_t + N_c + N_v)},$$

where E_{ce} is the cumulative error on the test set for a trained network. Setting this result in relation to the total costs for training termination has reached the minimum RMS error without the validation cost will become

$$\sigma = \frac{1}{EC(N_t + N_c)},$$

so their ratio

$$\xi = \frac{\rho}{\sigma} = \vartheta \frac{N_t + N_c}{N_t + N_c + N_v},$$

with

$$\vartheta = \frac{E}{E_{ce}}.$$

However, even when using the predetermined number of training steps, there will generally be no guarantee that the network parameters will be adequately tuned. The optimal stopping strategy is to stop training after the network has learnt all about the problem class it has to solve. This happens when the training stopping is effected at the point where the network has reached the maximal **generalization**. For the practising expert, this means that the stopping should be triggered exactly at the point where the network output error has reached its minimal value, This is known as **early stopping**. If the training is continued beyond this point, then the result could be the **network overtraining** or **network overfitting**.

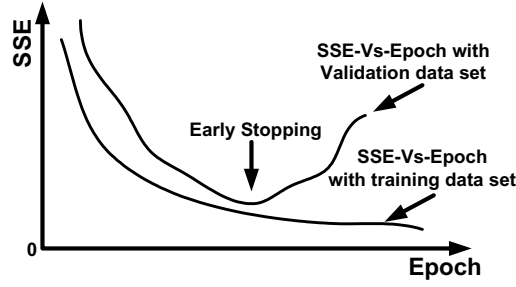


Figure 3.16. Early stopping of training

But still, the dilemma remains: in order to stop the training process, how do we realize that the network has learnt all the required knowledge from the training data and has reached its maximum generalization? Then, from learning theory we know that after reaching the point of maximum generalization, the network – although learning more and more from the training set - will start impairing the related test set performance (Figure 3.16) due to its overtraining (Vapnik, 1995). To prevent this, the method of *early stopping with cross-validation* has been suggested by Prechelt (1998).

Cross-validation is a traditional statistical procedure for random partitioning of collected data into a **training set** and a **test set**, and for further partitioning of the training set into the **estimation set** and the **validation set**. It is obvious that, if only a restricted data set is available, the partition of the entire set reduces the size of the training set. This, again, makes the location of the early stopping point difficult. For managing this problem, a predicate or a **stopping criterion** should be found that can indicate when to stop the training.

Prechelt (1998), using the error function (or the objective function) E , training error E_{tr} (as the average error per example across the training set), and the test and validation errors E_t and E_v respectively, has defined three possible stopping criteria:

- Stop as soon as the generalization loss exceeds a threshold value ε , *i.e.* when $g_{loss}(t) > \varepsilon$, where the error function $g_{loss}(t)$ is based on the lowest validation set error E_{opt} and the validation error E_v .
- Stop as soon as the quotient

$$\frac{g_{loss}(t)}{P_{tr}(t)} > \varepsilon,$$

where $P_{tr}(t)$ is the **training progress** defined by

$$P_{\text{tr}}(t) = 1000 \frac{\sum_{t'} E_{\text{tr}}(t')}{k \min_{t'} E_{\text{tr}}(t')},$$

with $t' = t - k + 1$, and the **training strip length** k .

- Stop when the generalization error increased in v successive strips.

Prechelt (1998), in order to interrogate the validity of the criteria, conducted 1296 training runs, producing 18144 stopping criteria. In the experiments, 270 of the records from 125 different runs reached automatically the 3000 epoch limit without using stopping criteria.

We will now consider the problem of **network overtraining** or **network overfitting** in more detail. Both the problem of overfitting and the opposite problem of **underfitting** arise as a consequence of improper training stopping. Therefore, both of them should be prevented because each of them lowers the generalization capability of the trained network. For example, if a network to be trained is less complex than the task to be learnt, then the network - after being trained - can suffer from underfitting and can, therefore, poorly identify the features within a large training data set. On the contrary, a too complex network can, after being trained, suffer from overfitting and can, therefore, extract the features within the training set along with the superposed noise. As a consequence, a complex network can produce predictions that are not acceptable.

Network complexity is primarily related to the number of weights. The term is used in connection with the model selection for prediction in the sense that the prediction accuracy of a network determines its complexity. This is the starting point of network model selection: how many and of what size of weights (and how many hidden units) should the model have in order to implement the wanted prediction accuracy without (or at least with a low) overfitting?

From the statistical point of view, the underfitting and overfitting are related to the statistical **bias** and the statistical **variance** they produce. They strongly influence the generalization capability of the trained network as follows:

- the **statistical bias** is related to the degree of target function fitting and restricts the network complexity, but does not care about the trained network generalization
- **statistical variance**, which is the deviation of network learning efficiency within the set of training data, cares about the generalisation of the trained network.

For instance, underfitting produces a very high bias at network outputs, whereas overfitting produces a large variance. The difficulty of their simultaneous reduction or their balancing in the process of learning, which is essential for achieving the highest possible degree of generalization, is known as the **bias-variance dilemma**. The dilemma is to be understood as follows: the bias of a neural network with a high fitting performance across the given training set of data is very low, but its variance is very high. By reducing the variance the network data fitting performance of the network will decrease. As a consequence, a trade-off between

the low bias and the low variance is necessary, as demonstrated in Figure 3.17 on the example of *polynomial curve fitting* of a set of given data points.

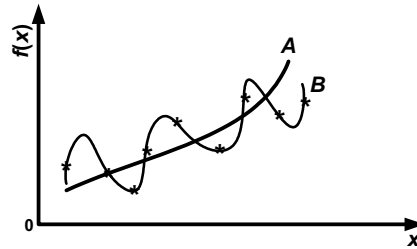


Figure 3.17. Polynomial curve fitting of data

A polynomial of degree n can exactly fit a set of $(n + 1)$ data points, say training samples. If the degree of the polynomial is lower, then the fitting will not be exact because the polynomial (as a regression curve A) cannot pass through all data points (Figure 3.17). The fitting will be erroneous and will suffer from **bias error**, formulated as the minimized value of the mean square error. In the opposite case, if the degree of the polynomial is higher than the degree required for exact fitting of the given training data set, the excess number of its degrees will lead to oscillations because of missing constraints (curve B in Figure 3.17). The polynomial approximation will, therefore, suffer from **variance error**. Consequently, a polynomial of the optimal degree should be chosen for data fitting that will provide a low bias error as well as a low variance error, in order to resolve the **bias-variance dilemma**.

Translated in terms of neural network training, polynomial fitting is seen as an optimal nonlinear regression problem (German et al., 1992). This means that, in order to fit a given data set optimally using neural network, we need a corresponding model implemented as a structured neural network with a number of interconnected neurons in hidden layer. If the size of the selected network (or the order of its model) is too low, then the network will not be able to fit the data optimally and the data fitting will be accompanied by a bias error that will gradually decrease with increasing network size until it reaches its minimal value. Increasing the network size beyond this point, the network will also start learning the noise present in the training data, because there will be more internal parameters than are required to fit the given data. With this, also the variance error of the network will increase. The cross-point of the bias and the variance error curve will guarantee the lowest bias error and the lowest variance error for fitting the given data set. The corresponding network size (i.e. the corresponding number of neurons) will solve the given data fitting problem optimally. At this point the network training should be stopped, which is known as **early stopping** or **stopping with cross-validation**. The network trained in this way will guarantee the **best generalization**.

For probabilistic consideration of polynomial fitting, the expected value of the minimum square error across the set of training data

$$MSE_D = E_D \{ [p(x) - f(x)]^2 \}$$

is taken, where the training points are represented by the function $f(x)$ and the fitting polynomial or the actual network output by $p(x)$. Expanding the MSE_D formally as

$$MSE_D = E_D \{ [p(x) - E_D \{p(x)\} + E_D \{p(x)\} - f(x)]^2 \}$$

and rearranging its expansion as

$$MSE_D = E_D \{ [p(x) - E_D \{p(x)\}]^2 \} + E_D \{ E_D \{p(x)\} - f(x) \}^2 \},$$

one gets the sum of the statistical variance

$$VAR_D = E_D \{ [p(x) - E_D \{p(x)\}]^2 \}$$

and the statistical bias

$$BIAS_D = E_D \{ E_D \{p(x)\} - f(x) \}^2 \}.$$

In summary, the optimal network size is essential for optimal problem solving because a relatively small network will not be able to fit the given data accurately and thus will not be able to learn the most important features incorporated in the data. For this reason, the network size should be increased. On the other hand, because a large-sized network tends to learn not only the characteristic features of the given data, but also the accompanying noise and other non-relevant components' idiosyncrasies hidden in the data, its size should be reduced. In both cases, a network size reduction and/or an increase in optimal network size should be found that ensures the optimal network performance. In practice, this is usually achieved by balanced **network growing** and/or by **network pruning**.

Network growing is a process of successive addition of new neurons and their related interconnections to the initial small-sized network until the optimal network performance is reached. This is a common way of designing optimal-sized radial basis function networks.

Network pruning, again, is a process of successive elimination of less relevant interconnections between the neurons within the large-sized network until the further elimination essentially worsens the network performance. A survey of algorithms to be used for network pruning was given by Reed (1993), who distinguished two major pruning methods:

- **sensitivity calculation methods**, based on the sensitivity of the error function of the trained network with respect to the removal of individual weight connections as the indication of their pruning
- **penalty term methods**, based on modification of the error function of a trained network by a penalty term.

Mozer and Smolensky (1988) used ρ as a measure of relevancy, defined as the difference between the error after removing a unit and the error before removing a unit. Karinin (1990), however, considers the error sensitivity with respect to removal of individual connections and removes the low-sensitivity connections. Le Cun *et al.* (1990), again, proposed the **optimal brain damage** procedure under the condition that the Hessian matrix H is diagonal and estimated the **saliency of the weights** and the second derivative of the error with respect to the weights. Hassibi *et al.* (1992) removed the diagonality restriction of the Hessian matrix and considered the general case of an arbitrary form of Hessian matrix, which they termed the **optimal brain surgeon**. Both approaches are based on consideration of sensitivity of weights perturbation on the error function E using the Taylor series

$$\delta E = \left(\frac{\partial E}{\partial w} \right) \delta w + \frac{1}{2} \delta w^T H \delta w + \left(\|\delta w\|^3 \right),$$

where

$$\delta E = E(w + \delta w)$$

and

$$H = \left(\frac{\partial^2 E}{\partial w^2} \right)$$

is the corresponding Hessian matrix.

Now, knowing that for a network trained to the local minimum in error, the partial derivative

$$\frac{\partial E}{\partial w} = 0$$

holds. Neglecting all higher order terms in the corresponding Taylor series and eliminating a specific weight, say w_{ij} , measures should be undertaken to minimize the increase in error δE , taking into account the condition of weight elimination as given by

$$\delta w_{ij} + w_{ij} = 0.$$

The condition of weight elimination in vectorial form is given by

$$e_{ij}^T \delta w + w_{ij} = 0,$$

where e_{ij}^T is the unit vector in the weight space and w_{ij} is the weight connecting the i th input of the j th hidden unit.

To solve the minimization problem, we form the corresponding Lagrangian

$$L = \frac{1}{2} \delta w^T H \delta w + \lambda (e_{ij}^T \delta w + w_{ij}),$$

where λ is the Lagrange multiplier. The derivative of the Lagrangian with respect to δw and the equation

$$e_{ij}^T \delta w + w_{ij} = 0,$$

define the optimal weight change

$$\delta w = \frac{w_{ij}}{[H^{-1}]_{ij}} H^{-1} e_{ij}.$$

Correspondingly, the related optimal value of Lagrangian L for the weight w_{ij} is

$$L_{ij} = \frac{1}{2} \frac{w_{ij}^2}{[H^{-1}]_{ij}},$$

where $[H^{-1}]_{ij}$ is the i th element of the inverse Hessian matrix H . The L_{ij} value of the Lagrangian determined in this way represents the increase of mean square error caused by the removal of the weight w_{ij} , known as **saliency of the weight** w_{ij} . It is obvious that, because the saliency depends on the square value of w_{ij} , the small values of weights have a low influence on the mean square error. However, because the saliency is inversely proportional to $[H^{-1}]_{ij}$, small values of $[H^{-1}]_{ij}$ can also have a strong influence on the mean square error.

Although **pruning** methods, such as **optimal brain damage**, and **optimal brain surgeon**, rely on the weight ranking with respect to saliency, *i.e.* on changes in training error caused by pruning an individual weight, there is still an essential difference between them: the optimal brain damage procedure does not require retraining of the network after removing a weight element, whereas the optimal brain surgeon procedure requires this.

The disadvantage of both methods is that, if no **stopping criterion** is built, the removal of the least significant weights can lead to network overfitting. As an efficient stopping criterion, the calculation of the test error using Akaike's (1970) **final prediction error** (FPE) estimation and its modification is used to cover the estimation of average generalization error in regularized networks (Moody, 1991).

In practice, to apply the above procedures, the second derivative (Buntine and Weigend, 1994) of the inverse of Hessian matrix (Hassibi *et al.*, 1992) has to be calculated anew for every weight to be eliminated. Stahlberger and Riedmiller (1996) proposed a fast network pruning method, called Uni-OBS, that still relies on the optimal brain surgeon procedure but it requires only a single calculation of the inverse Hessian matrix to eliminate a group of weights. This certainly simplifies the calculation of net pruning. For accelerated calculations of matrix multiplication, some fast computational algorithms are required or some algebraic transformations that also accelerate the calculation process. An amendment of the Uni-OBS method, called G-OBS (**generalised optimal brain surgeon**), can simultaneously eliminate, say m , weights in one step with slight increase in error given as

$$\delta E = \frac{1}{2} \delta w^T H \delta w,$$

The related elimination condition is given by

$$(w + \delta w)^T S_m,$$

S_m being the selection matrix that determines the m weights to be removed simultaneously. Using the above weights elimination conditions and the corresponding Lagrange method, we get for the resulting error the relation

$$\delta w = -H^{-1} S (S^T H S)^{-1} S^T w$$

and

$$\delta E = \frac{1}{2} w^T S (S^T H S)^{-1} S^T w.$$

For acceleration of the pruning process, Levin *et al.* (1994) proposed a method for elimination of excess weights.

Another way was followed by Jolliffe (1986). To improve the network generalization capability, he used the method of **principal component analysis**. This is a valuable mathematical tool for reducing a system's dimensionality by eliminating its redundant variables. This method transforms the variables to a basis in which the system covariance is diagonal and the projection is in the low variance directions. To detect the variables that have a low significant influence on the error function, a **saliency measure** is used, which demonstrates the relationships between the proposed methods and the optimal damage and optimal surgeon procedures of network pruning. The pruning consists in removing the **eigen-nodes** with low saliency to reduce the effective number of network parameters. In contrast to the optimal brain damage and optimal brain surgeon procedures, which reduce the rank by eliminating actual weights, the proposed

method reduces the rank of weights in each layer by deletion of the smallest salient eigen-nodes. Finally, the proposed method does not require network training.

A network pruning approach is preferably used in designing networks with a high **generalization capability**, *i.e.* networks that are not only good enough to solve the prediction or classification problems present in the training set, but also some similar problems using some fresh, never seen and not previously known training sets of data. This is achieved through a trade-off between the intention that the trained network should be capable of learning a broad spectrum of similar problem categories, which would require a large-sized network, and the requirement that the network should be as simple as possible, in order to avoid the **overtraining**.

In practical application of a trained network, there is a fundamental recommendation, *i.e.* where several trained networks have approximately the same final performances, the structurally simplest network should be selected as the best generalized one. This recommendation reflects **Occam's razor philosophy**, which recommends that a scientific model should favour simplicity.

Many training strategies have been interrogated for network simplification at lower training cost. Such strategies have been discovered within the framework of minimization of the error function extended by a penalty term. To this category of strategies belong:

- the **weight decay** approach (Hinton, 1989), a subset of regularization approaches based on minimization of the weight tuning rule augmented by a complexity penalty term

$$\Delta w_{ij}(t+1) = \pi \delta_i x_j - \lambda w_{ij}$$

that penalizes the large weight values.

- the **weight elimination** approach (Weigend *et al.*, 1991), based on minimization of network training cost function to which a term is added that accounts for the number of parameters:

$$\Delta w_{ij}(t+1) = \eta \delta_i x_j - \lambda \frac{w_{ij}(t)}{[1 + w_{ij}^2(t)]^2},$$

where λ represents the weight decay constant, δ_i is the local error, x_j is the local activation, and η is the learning rate.

In contrast to weight decay, which shrinks large values of weights more than small ones, the weight elimination shrinks predominantly the small weight values and is to a certain degree similar to the pruning process. Hansen and Rasmussen (1994) have demonstrated that network pruning may result when the weight decay parameter is determined by data. The added term punishes the large weight values and forces them to obtain small absolute values and simultaneously retains the other values unchanged. This, however, is favourable in preventing worsening of

the network generalization capability. Therefore, care should be taken in selecting the decay constant λ , because an inappropriate value can deteriorate the generalization capability of the weight decay process. As a remedy, Weigend *et al.* (1991) recommend updating the λ value on-line during the network training in iterative steps.

Adding the penalty function in the weight decay and optimizing the augmented performance index corresponds to the **regularization method** in which the penalty term is added to the cost function to act as a restriction to the subsequent optimization problem. In approximation theory, the added term penalizes the curvature of the original solution, seeking for a smoother solution of the optimization problem.

The regularization method is generally used to solve **ill-posed problems**. In the theory of learning, the problems of learning smooth mappings from examples are mostly ill-posed problems. For their solution Tikhonov (1963) proposed optimization of the cost function I extended by a term J , which also represents a cost function. Thus, the resulting cost function to be optimized becomes

$$I_{res} = I + \lambda J,$$

where λ represents the **regularization parameter**, which determines the **degree of regularization** in the sense of balancing the **degree of smoothness** of the solution and its **closeness** to the training data. The regularization helps in stabilizing the solution of the ill-posed problem because the added term, representing the penalty to the original optimization problem, smoothens the cost function (Morozov, 1984).

The regularization approach determines the so-called **Tikhonov functional**

$$I_{res}(f) = \sum_{i=1}^n (y_i - f(x_i))^2 + \lambda \|Pf\|^2,$$

the first term of which represents the closeness to the data, and in the second term f is the input-output function, P is a linear differential constraint operator, and $\|\cdot\|^2$ is a norm on the function space to which Pf belongs. This operator also embodies the *a priori* knowledge about the problem solution.

To solve the regularization problem we proceed with the minimization of extended cost function I_{res} , using the resulting partial derivatives with respect to f in order to build the Euler-Lagrange equation

$$\hat{P}Pf(x) = \frac{1}{\lambda} \sum_{i=1}^n (y_i - f(x))\delta(x - x_i),$$

in which the operator P and its adjoint operator \hat{P} build the differential operator $\hat{P}P$. Therefore, the above Euler-Lagrange equation is a partial difference equation. Its solution can, therefore, be expressed as the integral transformation of the right-

hand side of the equation, with the kernel defined by Green's function of the differential operator $\hat{P}P$

$$\hat{P}PG(x, x_i) = \delta(x - x_i) .$$

Bearing in mind the definition of Green's function and taking into account the presence of the delta function on the right-hand side of the equation, the integral transformation will generate a discrete sum of terms, so that the function f can be defined as

$$f(x) = \frac{1}{\lambda} \sum_{i=1}^n (y_i - f(x_i)) G(x, x_i) ,$$

where $G(x, x_i)$ is **Green's function** centred at x_i . The last equation represents the solution of the regularization problem as a linear combination of n Green's functions with the expansion centre x_i and expansion coefficients $(y_i - f(x_i))$. Consequently, the solution of the regularization problem lies in the n -dimensional subspace of the space of smooth functions, with the n Green's functions as its basis (Poggio and Girosi, 1990). Furthermore, the basis function depends on stabilizer P , that represents the *a priori* knowledge of the problem domain as a kind of constraint.

Introducing the definition of the expansion weights as

$$w_i = \frac{y_i - f(x_i)}{\lambda} ,$$

the above solution equation becomes

$$f(x) = \sum_{i=1}^n w_i G(x, x_i) .$$

Now, to determine the expansion weights w_i , the last two equations have to be written in matrix form as

$$w = \frac{1}{\lambda} (y - f)$$

and

$$f = Gw$$

which result in

$$(G + \lambda I)w = y.$$

Here, I represents the n -dimensional identity matrix and G is the corresponding **Green's matrix**

$$G = \begin{bmatrix} G(x_1, x_1) & G(x_1, x_2) & \dots & G(x_1, x_n) \\ G(x_2, x_1) & G(x_2, x_2) & \dots & G(x_2, x_n) \\ \dots & \dots & \dots & \dots \\ G(x_n, x_1) & G(x_n, x_2) & \dots & G(x_n, x_n) \end{bmatrix},$$

which is a symmetric matrix with the property

$$G(x_i, x_j) = G(x_j, x_i)$$

because the identity matrix I is also symmetric.

From the solution equation

$$f(x) = \sum_{i=1}^n w_i G(x, x_i)$$

the corresponding **regularization network** (Figure 3.18) can be structured. The input layer of the network has an equivalent number of units to the dimension of the input vector, *i.e.* to the number of independent variables of the problem to be solved. The subsequent hidden layer, fully connected with the input layer with the fixed value weights, has the same number of nonlinear units as the number of data points and the activation function in the form of a Green's function with the output $G(x, x_i)$. It does not participate in the training process. Finally, the output layer, also fully connected to the hidden layer, contains one or more linear units with the weights w_i that correspond to the unknown coefficients of the above solution equation.

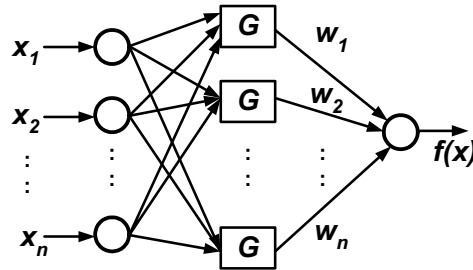


Figure 3.18. Regularization network

Obviously, the structure of the regularization network is mainly determined by the problem to be solved, with the exception of the weights between the input layer and the hidden layer, which are fixed. The main attributes of the network are:

- the regularization network is an optimal network because it minimizes the performance index that defines the proximity of the elaborated solution to the real solution defined by the training data
- the regularization network represents the *best approximator* (Girosi and Poggio, 1990) in the sense that for a given function there always exists a number of coefficients that approximate the given function better than any other set of coefficients and – by properly defining the stabilizer – guarantee that the regularization network has the desirable degree of smoothness
- the regularization network is a *universal approximator* that, given a sufficiently large number of hidden neurons, can approximate any continuous multivariate function arbitrarily well on a compact domain, a property that is based on the classical *Weierstrass theorem*.
- when it is used for simplification of linear networks, particularly of basis function networks, this corresponds to the *ridge regression method*.

The above objectives can, at least in principle, be reached by “extensive” network training. Although this might lead to network overfitting, this can be prevented by training stopping with cross-validation and by network structure reduction, for which various approaches have been suggested.

3.6 Forecasting Using Neural Networks

Unlike the traditional approaches to time series analysis and forecasting, neural networks need a reduced quantity of information to forecast the future time series data. Based on the available time series data, network internal parameters are tuned using an appropriate tuning algorithm. This can, if necessary, also include the modification of the initially chosen network architecture to better match the architecture required by the problem at hand. The related issues have been discussed extensively in this chapter, so that our attention will be focused on the comparison of the traditional approach to time series forecasting and on the approach using neural networks. This will be followed by pointing out the benefits of forecasting by merging both kinds of approaches and by building a nonlinear combination of forecasts. Finally, some issues related to the forecasting of multivariable time series using neural networks will be presented.

3.6.1 Neural Networks versus Traditional Forecasting

Comparison of forecasting performance of traditional statistical methods and of *neuro forecasters* has, since the early 1990s, attracted the attention of many researchers. Their reports have, however, been inconsistent because they were based on experimental investigations using various network configurations with

various performance quality. Added to this came that the experiments used different time series data. For instance, forecasting collected linear data using nonlinear mapping of neural networks cannot give better results than the forecasting using linear statistical algorithms. In the reverse case, when dealing with considerably nonlinear time series data, forecasting using nonlinear neural networks could definitely deliver better results than the traditional algorithms. Consequently, when dealing with mixed linear/nonlinear time series data a combination of the traditional and the neural approach could be optimal.

Lapedes and Farber (1988) were the first to report that simple neural networks can outperform traditional methods by up to many orders of magnitude. This was radically investigated by Sharda and Patil (1990) on a set of 75 different time series with the objective to compare the forecasting accuracy of the Box-Jenkins method and of a neuro forecaster. Using a subset of 14 time series of Sharda and Patil, Tang *et al.* (1991) extended the comparative analysis to some additional aspects and identified a number of facts that make neural networks or traditional approaches deliver better forecasting results. They found by experiments that, generally:

- for time series with long memory, both approaches deliver similar results
- for time series with short memory, neural networks outperform the traditional Box-Jenkins approach in some experiments by more than 100%
- for time series of various complexity, the optimally tuned neural network topologies are of higher efficiency than the corresponding traditional algorithms.

As typical examples for experimental study

- international airline passenger data
- domestic car sales data in the US and
- foreign car sales data in the US

were used.

For experiments, the most typical traditional forecasting approach, the ARMA model of Box-Jenkins approach

$$\phi_p(B)\phi_p(B^L)(1-B^L)^D(1-B)^d y_t = \theta_q(B)\theta_q(B^L)a_t + \delta$$

was used with the autoregressive operator ϕ , moving-average operator θ , and the back shift operator B . In the model equation, a_t , y_t , and δ represent the white noise, the time series data, and a constant value respectively.

To simplify matters, in all experiments with neuro forecasters, one-hidden-layer networks and networks without a hidden layer were used alternatively. The experimental results showed that hidden-layer networks have a better forecasting performance.

Hill *et al.* (1996) compared six traditional methods with the neuro forecaster on 111 different time series and found that neuro forecasters are significantly better than the statistical methods taken into consideration. However, Foster *et al.* (1992) came to the opposite conclusion. After extensive analysis of forecasting accuracy

of neuro and traditional forecasters, they concluded that linear regression and the simple average of the exponential smoothing method are superior to a **neuro forecaster**. Denton (1995), again, demonstrated that, under standard statistical conditions, there is only a slight difference in prediction accuracy between the regression models and neural models. Some additional results of comparative analysis have been communicated by Nelson *et al.* (1994), Gorr *et al.* (1994), Srinivasan *et al.* (1994), and Hann and Streurer (1996).

3.6.2. Combining Neural Networks and Traditional Approaches

Application of **hybrid**, *i.e.* combined neural networks and traditional approaches, to time series forecasting was a challenging attempt to increase forecasting accuracy beyond the limits that either one of the two approaches used alone would be able to reach. In the following, we will consider the advantages of combining the neural and ARIMA model approach in time series forecasting. Voort *et al.* (1996) used for this combination the Kohonen self-organizing map as the neural network part for short-term traffic-flow forecasting. Sue *et al.* (1997) used this type of hybrid combination to forecast a time series of reliability data and showed that the hybrid model produced better forecasts than either the ARIMA model or the neural network by itself could produce. Tseng *et al.* (2002) investigated the combination of a seasonal time series model SARIMA and a backpropagation network, resulting in a SARIMABP hybrid combination. They found that the combination outperforms the SARIMA model used alone and the backpropagation model with the de-seasonalized or differentiated data.

For experimental purposes, the time series $z_i, i = 1, 2, 3, \dots, k$, is generated by a SARIMA $(p, d, q)(P, D, Q)$ process with mean μ and modeled by

$$\varphi(B)\Phi(B^S)(1-B)^d(1-B^S)^D(z_i - \mu) = \theta(B)\Theta(B^S)a_i,$$

where S is the periodicity, d and D are the number of regular and seasonal differences respectively, B is the polynomial degree, and a_i is the estimated residual at time t . The experimental results show that the SARIMABP method benefits from the forecasting capability of the SARIMA and from the capability of backpropagation to reduce the residuals further, which guarantees a lower forecasting error. As forecasting accuracy evaluation criteria, the mean square error (MSE), mean absolute error (MAE), and mean absolute percentage error (MAPE) have been used.

For a real-life application example, time series data of the total production revenues of the Taiwanese machinery industry were taken for various periods of time. For instance, a five-year data set has been used as the input of the ARIMA $(0, 1, 1)(1, 1, 1)_{12}$ model

$$(1 + 0.309B^{12})(1-B)(1-B^{12})z_i = (1 - 0.7159B^{12})a_i$$

and for a three-year data set as the input of the ARIMA $(0, 1, 1)(0, 1, 0)_{12}$ model

$$(1-B)(1-B^{12})z_t = (1-0.88126B)a_t.$$

In both cases the experiments were carried out with two, three, and seven neurons in the network hidden layer.

Hybrid ARIMA-neural network methodology was also the subject of an experimental study by Zhang (2003), whose objective was to identify whether the given time series data were generated by a linear or a nonlinear process. This is essential for making a decision on whether, in a given case, the use of a linear (*i.e.* the traditional) or a nonlinear (*i.e.* a neural network) approach will be more appropriate. Here, the combined approach could ease the problem solution. After all, because real-world time series are seldom purely linear or nonlinear, it is favourable to use a hybrid approach.

In experimental practice, the assumption is made that a time series to be processed is composed of a linear autocorrelation structure L_t and a nonlinear component N_t :

$$z_t = L_t + N_t.$$

The linear component of the time series can be processed using an ARIMA model, and the residuals

$$e_t = z_t - \hat{L}_t,$$

containing only the nonlinear relationships, can be processed by neural networks. This can be done using a residual model, *e.g.*

$$e_t = f(e_{t-1}, e_{t-2}, \dots, e_{t-n}) + \varepsilon_t,$$

which corresponds to a neural network with n input nodes and the nonlinearity function $f(\cdot)$. In the above residual model, ε_t represents the random error. The benefits of the proposed hybrid methodology approach have been confirmed on three real-life examples from different application areas.

A remarkable contribution was reported by Wedding and Chios (1996), who combined the Box-Jenkins model and an RBF network.

3.6.3 Nonlinear Combination of Forecasts Using Neural Networks

Because a large number of time series forecasting methods are available, it makes sense for the application expert to select the best one among them in each particular case. Thus, it becomes interesting to combine a group of forecast methods and to examine the forecasting accuracy of the combination. The issue was discussed in Section 2.8.6 from the traditional point of view. It was shown that the best forecasting results are achievable when the combination of traditional forecasting methods is nonlinear. In the meantime, various combination techniques

have been suggested and examined using different intelligent technologies, primarily with neural networks.

In engineering practice, choosing the “best” forecasting method means choosing a method that is the best in the given circumstances. For instance (McNees, 1985), experience has shown that no forecasting model retains its accuracy for all values of variables all the time. Also, it has been experimentally proven that if for a forecasting method the short run is good, then there is no guarantee that the long run will also be good. Therefore, it is worthwhile seeking for an adequate combination for each application situation. This is because the combination of methods incorporates different cognition capabilities and can, in a specific case, produce better forecasts than either of methods within the combination itself. Moreover, experimental investigations confirm (Winkler and Markridakis, 1983) that the resulting accuracy of combined forecasts increases with the increase in the number of forecasting methods involved. Mahmoud (1984) also came to a similar conclusion, that the accuracy of the combined forecast improves as more methods are included in the combination.

In forecasting non-stationary, non-seasonal time series one can evaluate the forecast values subsequently generated by a Box-Jenkins ARMA or ARIMA model, Holt-Winter’s exponential smoothing, extrapolation of trend curve, Kalman filtering, *etc.* and mutually compare the results achieved. Out of the possible forecasting methods the analyst may prefer to use his own favourite methods that will produce different forecasts of a given time series. Moreover, using a particular method (say, ARMA/ARIMA) different analysts may come up with a different order of the models required for forecasting and, again, with different forecast results. Therefore, forecast models developed using different methods and by different analysts will rarely be identical. This may be very confusing to someone who wants to take a decision on the basis of various forecasts suggested by various analysts.

From the above, it follows that it is inadvisable to prefer one particular forecasting method over another, because no single forecasting method will in every situation produce forecasts of the same accuracy. Rather, it is more advisable to take a combination of a few forecasts generated by different methods. This was even clearly formulated by Bates and Granger (1969).

A number of advanced approaches have been suggested for nonlinear combination of forecasts using neural networks (Shi and Liu, 1993; Harald and Kamastra, 1997). The problem is defined here starting with the availability of k different forecasts $f_1, f_2, f_3, \dots, f_k$, of some random variable z , that should be combined into a single forecast f_c . The straight away step would be to form a linear combination of forecasts

$$f_c(z) = \sum w_i f_i(z)$$

where w_i is the assigned weight of i th forecast f_i .

The simplest approach to determine the weights w_i of the combination would be to take equal weights for each term. This has proven to be relatively robust and accurate. But still, in practice, the linear combination of forecasts is not likely to be

the optimal combination like the nonlinear combinations are. This can be demonstrated on the following example.

Suppose that k different forecast models are available and the i th individual forecast has an information set $\{I_i : I_c, I_i\}$, where I_c is the common part of information used by all k models and I_i is the specific information for the i th forecast only. Denoting the i th forecast by $f_i = F_i(I_i)$, we can express the linear combination of forecasts as

$$F_c = \sum w_i F_i(I_i),$$

where w_i is the weight of the i th forecast. On the other hand, every individual forecasting model can also be regarded as a subsystem for information processing, while the combination model $f_c = F_c(I_1, I_2, \dots, I_k)$ is regarded as such a system. It follows that the integration of forecasts is more than their sum, *i.e.* the performance of the integrated system is more than the sum of its subsystems. So, the trustworthiness of the linear forecast combination is quite questionable. More trust should be paid to a nonlinear interrelation between the individual forecasts, such as

$$f_c = \psi[F_1(I_1), F_2(I_2), F_3(I_3), \dots, F_k(I_k)],$$

where ψ is a nonlinear function. While the given information is processed by individual forecasting models, it is likely that parts of the entire information can be lost, which means that, say, the information set I_i is not being used efficiently. Furthermore, different forecasts may have different parts of information lost. This is why it is preferable that as many different forecasts as possible should be present in the combination, even when the individual forecasts depend on the same set of information.

As a forecasting example (Palit and Popovic, 2000), a 2-6-6-1 feedforward network, *i.e.* a network with two inputs, and two hidden layers with each layer containing six neurons and one output, is used, as shown in Figure 3.19b. The network is trained using the Levenberg-Marquardt algorithm, which guarantees much faster learning speed than the standard backpropagation method, and hence requires less training time. The algorithm also uses the gradient descent method, based on Jacobian matrix, according to which the update is

$$\Delta w = -\left[J^T(w)J(w) + \mu I\right]^{-1} J^T(w)e(x)$$

or

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

$$w(k+1) = w(k) - \left[J^T(w)J(w) + \mu I\right]^{-1} J^T(w)e(w)$$

where $J(w)$ is the Jacobian matrix with respect to network-adjustable parameters w (all weights and the biases) of dimension $(q \times N_p)$, and q being the number of

training sets, N_p being the number of adjustable parameters in the network, and I is the identity matrix of dimension $(N_p \times N_p)$.

Table 3.1. Nonlinear combination of two forecasts of a temperature series using an artificial neural network (ANN: Neural networks combined forecast; BJ: Box-Jenkins forecast, HW: Holt-Winters exponential smoothing)

Serial No.	Forecast	Data sets from HBXIO matrix	SSE	RMSE
1.	BJ	151 to 224 (column-1)	0.4516	0.112
2	HW	151 to 224 (column-2)	0.3174	0.0933
3	ANN (2-6-6-1)	1 to 150 (training)		
4	ANN (2-6-6-1)	151 to 224	0.1306	0.0594
5	ANN (2-2-6-1)	151 to 224	0.2425	0.0810

The parameter μ is multiplied by some factor μ_{inc} whenever an iteration step increases the network performance index (*i.e.* sum squared error) and it is divided by μ_{dec} whenever a step reduces the network performance index. Usually the factor $\mu_{inc} = \mu_{dec}$ and in our case it is selected as 10.

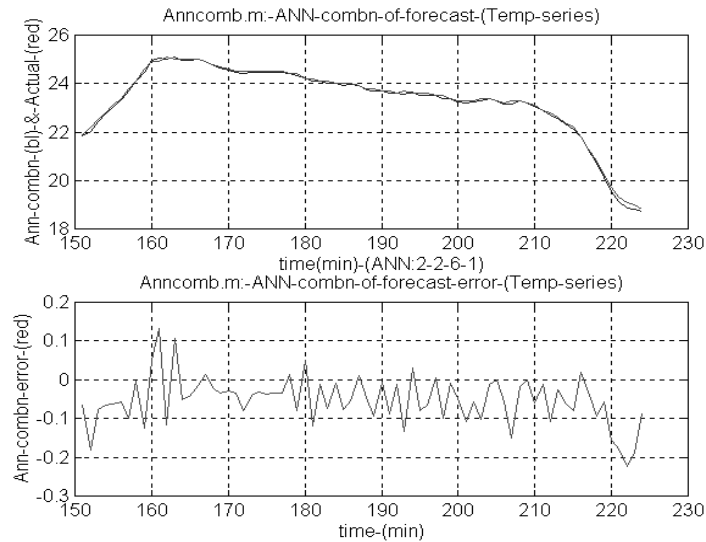


Figure 3.19(a). The combination of forecasts using a 2-2-6-1 artificial neural network

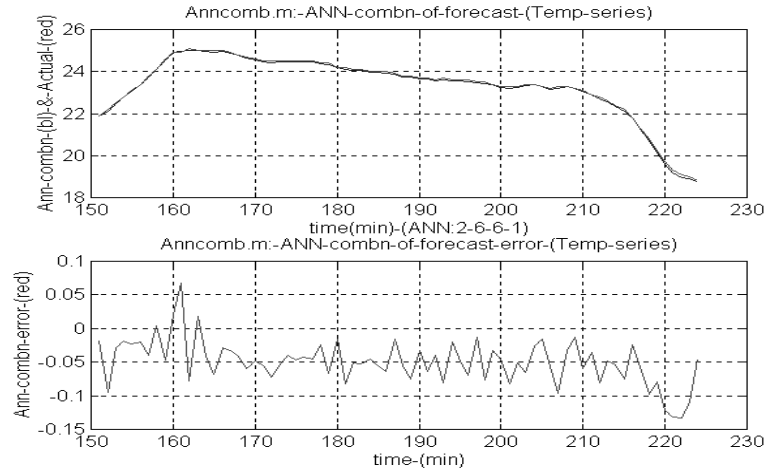


Figure 3.19(b). The combination of forecasts using a 2-6-6-1 artificial neural network

In our practical example, the first 150 input-output samples were used to train the network. Thereafter, the values of the interconnecting weights and biases are saved for network performance testing using the remaining 151 to 224 samples of data. From the experimental results shown in Figure 3.19(a) and Figure 3.19(b) and Table 3.1, it is obvious that the network output very closely matches the actual time series, indicating that a nonlinear combination of the forecasts is better than the individual forecasts.

3.6.4 Forecasting of Multivariate Time Series

Chakraborty *et al.* (1992) conducted experimental investigations on forecasting of multivariate time series using neural networks. They focused their attention on the statement that, in the case of substantial cross-correlation of individual variables of multivariable time series data, the forecasting accuracy of each variable can be improved when simultaneously changing the values of other variables within the time series is taken into account. This has been observed in ***multivariate statistical analysis*** when, based on observation data, identifying the interdependencies of variables involved in a multivariate system. To prove this, Chakraborty *et al.* (1992) analyzed the one-step and multistep prediction behaviour of a ***trivariate time series*** $x_t = [x_{1t}, x_{2t}, x_{3t}]$ in the interval of $t = 1-100$ samplings using

- ***separate modelling*** of each component of the multivariable time series, interpreted as mutually independent univariate time series
- ***combined modelling***, by simultaneous consideration of all three variables
- ***statistical modelling***, using the statistical model developed by Tiao and Tsay (1989).

The analysis of separate modelling was carried out using alternatively 2-2-1, 4-4-1, 6-6-1, and 8-8-1 networks and by evaluating the results for each time series component using the mean square error as the performance indicator. The analysis has shown that a combined modelling approach is superior to separate modelling, and that both of them are superior to statistical modelling. In addition, the experiments with the 2-2-1 backpropagation networks have delivered, in one-step and multistep cases, the best forecasting accuracy, which shows that the 4-4-1 and 6-6-1 networks are oversized for this purpose.

The experimental investigations presented above deliver forecasting results that depend considerably on the art of experiment design used for this purpose. For this reason the results are not coherent and are sensitive to the application field. We are still short of a general theoretical formulation of this phenomenon, but some encouraging trials have been made in this direction (reported by Yang, 2000), related to methods of combining forecasting procedures for forecasting continuous random univariate time series.

References

- [1] Aizerman MA, Braverman EM, and Rozenoer LI (1964) Theoretical foundation of potential function method in pattern recognition. *Automation and Remote Control* 25: 917–936.
- [2] Akaike H (1970) Statistical predictor identification, *Annals of the Institute of Statistical Maths.*, 22: 202–217.
- [3] Almeida LB (1987) A learning rule for asynchronous perceptrons with feedback in a combinatorial Environment. *IEEE 1st International Conf. on Neural Networks*, San Diego, CA II:609–618.
- [4] Amari S and Maginu K (1988) Statistical neurodynamics of associative memory, *Neural Networks* 1: 63–73.
- [5] Anders U and Korn O (1999) Model selection in neural networks. *Neural Networks* 12: 309–323.
- [6] Bashkirov OA, Braverman EM, and Muchnik IB (1964) Potential function algorithms for pattern recognition learning machines. *Automation and Remote Control* 25:692–695.
- [7] Bates JM and Granger CWJ (1969) The combination of forecasts, *Operation Research Quart.* 20: 451–461.
- [8] Baum EB and Haussler D (1989) What Size Net Gives Valid Generalisation? *Neural Computation* 1:151–160.
- [9] Bethea RM and Rhinehard RR (1991) *Applied Engineering Statistics*. Marcel Dekker, New York.
- [10] Block HD (1962) The Perceptron: a model of brain functioning. *Review of Modern Physics*, 34:123–135.
- [11] Broomhead DS and Lowe D (1988) Multivariable functional interpolation and adaptive networks. *Complex Systems* 2: 321–355.
- [12] Butine WL and Weigend AS (1994) Computing Second Derivatives in Feedforward Networks: A Review. *IEEE Trans. on Neural Networks* 3: 480–488.
- [13] Chakraborty K, Mehrotra K, Mohan ChK, Ranka S (1992) Forecasting the behavior of Multivariate Time Series Using Neural Networks. *Neural Networks* 5: 961–970.
- [14] Cichocki A and Unbehauen R (1993) *Neural Networks for Optimization and Signal Processing*. Wiley, Chichester, West Sussex, UK.

- [15] Cohen MA and Grossberg S (1983) Absolute Stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Trans. on Systems, Man, and Cybernetics* 13: 815–826.
- [16] Cybenko G (1989) Approximation by superpositions of a sigmoidal function. *Mathematical Control Signals Systems* 2:303–314.
- [17] Denton JW (1995) How good are neural networks for causal forecasting? *J. of Business Forecasting* 14(2):17–20.
- [18] Elman JL (1990) Finding structure in time. *Cognitive Science* 14: 179–211.
- [19] Fogel DB (1991) An Information Criterion for Optimal Neural Network Selection. *IEEE Trans. On Neural Networks* 2: 490–497.
- [20] Forster WR, Collopy F, Ungar LH (1992) Neural network forecasting of short, noisy time series. *Computers and Chemical Engineering* 16(2): 293–297.
- [21] German SE, Bienenstock, and Doursat R (1992) Neural networks and the bias/variance dilemma. *Neural Computation* 1: 1–58.
- [22] Girosi F and Poggio T (1989) Representation Properties of Networks: Kolmogorov's Theorem is Irrelevant. *Neural Computation* 1: 465–469.
- [23] Girosi F and Poggio T (1990) Networks and the best approximation properties. *Biological Cybernetics*:169–176.
- [24] Gorr WL, Nagin D, Szczypula J (1994) Comparative study of artificial neural network and statistical models predicting student grade point averages. *Intl. J. of Forecasting* 10: 17–34.
- [25] Grossberg S (1988) Competitive Learning: From interactive activation to adaptive resonance, *Neural Networks and Neural Intelligence*, Grossberg S. (Eds.), MIT Press, Cambridge, MA.
- [26] Hagan MT and Menhaj MB (1994) Training feedforward networks with the Marquardt algorithm, *IEEE Trans. on Neural Networks*, vol. 5(6): 989–993.
- [27] Hann TH and Steurer E. (1996) Much ado about nothing? Exchange rate forecasting: Neural networks vs. linear using monthly and weekly data. *Neurocomputing* 10: 323–339.
- [28] Hansen IK and Rasmussen CE (1994) Pruning from adaptive regularization. *Neural Computation* 6: 1223–1232.
- [29] Harald PG and Kamastra M (1997) Evolving artificial neural networks to combine the financial forecasts, *IEEE Trans. on Evolutionary Computation*, vol. 1(1): 40–51.
- [30] Hassibi B, Stork DG, and Wolff GJ (1992) Optimal brain surgeon and general network pruning. *IEEE Intl Conf on Neural Networks*, San Francisco 1:293–299.
- [31] Haykin S (1994) *Neural Networks: a comprehensive foundation*. McMillan, USA
- [32] Hebb DO (1949) *The organisation of behaviour*. Wiley, New York.
- [33] Hecht-Nielsen R (1987a) Counterpropagation Networks. *Applied Optics* 26(23): 4979–4984.
- [34] Hecht-Nielsen R (1987b) Kolmogorov's Mapping Neural Network Existence Theorem, *IEEE Conf. On Neural Networks*; San Diego, CA. III: 11–14.
- [35] Hecht-Nielsen R (1988) Application of counterpropagation networks, *Neural Networks* 1: 131–139.
- [36] Hertz J, Krogh A, and Palmer RG (1991) *Introduction to theory of neural computation*, Addison-Wesley, Reading, MA.
- [37] Hill T, O'Connor M, Remus W. (1996) Neural network models for time series Models forecasts. *Management Sciences* 42(7): 1082–1092.
- [38] Hinton GE (1989) Connectionist learning procedures, *Artificial Intelligence*, 40: 185–243.
- [39] Hopfield JJ (1982) Neural Networks and physical systems with emergent collective computational abilities. *Proc. of the Nat. Acad. of Sciences, USA*, 79: 2554–2558.

- [40] Hopfield JJ (1984) Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. of the Nat. Acad. of Sciences, USA* 81: 3088–3092.
- [41] Hornik K, Stinchcombe M, White H (1989) Multilayer feedforward networks are Universal approximators. *Neural Networks* 2(5): 359–366.
- [42] Hu MJC (1964) Application of the ADALINE system to weather forecasting. Master Thesis, Technical Report 6775–1, Stanford El. Lab., Stanford, CA.
- [43] Ishikawa M. and Moriyama T (1996) Prediction of time series by a structural learning of neural networks. *Fuzzy Sets and Systems* 82: 167–176.
- [44] Iyer MS and Rhinehart RR (2000) A novel method
- [45] Iyer MS and Rhinehart RR (2000) A Novel Method To Stop Neural Network Training. 2000 American Control Conference, paper WM17–3
- [46] Johanson EM, Dowla EU, and Goodman DM (1990) Backpropagation learning for multi-layer feedforward neural networks using the conjugate gradient method, Report UCRL-JC–104850, Lawrence Livermore National Laboratory, CA.
- [47] Jolliffe IT (1986) *Principal Components Analysis*. Springer-Verlag.
- [48] Jordan M (1986) Attractor dynamics and parallelism in a connectionist sequential machine. *Proc. of the Eight Annual Conference on Cognitive Science Society* :532–546.
- [49] Karnin ED (1990) A simple procedure for Pruning back-propagation trained neural networks. *IEEE Trans on Neural Networks* 2: 188–197.
- [50] Khorasani K and Weng W (1994) Structure Adaptation in Feedforward Neural Networks. *Proc. IEEE Internat. Conf. on Neural Networks, III*: 1403–1408.
- [51] Klimasauskas CC (1991) Applying Neural Networks. Part 3: Training a Neural Network. *PC-AI*, May/June: 20–24. B,
- [52] Kohonen T (1989) *Self-Organisation and Associative Memory*. 3rd Edition, Springer, Berlin, NY.
- [53] Kröse B and Smagt P (1996) An introduction to neural networks, The University of Amsterdam, Eighth edition, November, <http://www.fwi.uva.nl/research/neuro>.
- [54] Kubat M (1998) Decision trees can initialise radial-basis-function networks. *IEEE Trans. on Neural Networks*. 9: 813–821.
- [55] Kurita T (1990) A method to determine the number of hidden units of three-layered neural networks by information criteria, *Trans. of Inst. of Electronics, Information and Commun. Engineers*, J73-D-II–11: 1872–1878 (in Japanese).
- [56] Lapedes A and Farber R (1988) Nonlinear signal processing using neural networks: Prediction and system modelling. Technical Report LA-UR-87-2662, Los Alamos National Laboratory, Los Alamos, NM.
- [57] Le Cun Y, Denker JS, and Solla SA (1990) Optimal Brain Damage. In: Touretzky S (Ed.). *Advances in Neural Information Processing Systems 2*, San Mateo, CA, Morgan Kaufman.
- [58] Levin AU, Leen TK, and Moody JE (1994) Fast pruning using principle components, In: *Advances in Neural Information Processing Systems 6*, Covan JD, Tesauro G and Alspector J, Editors: 35–42, Morgan Kaufman Publi. Inc., San Mateo, CA.
- [59] Lippmann RP (1987) An introduction to computing with neural nets. *IEEE ASSP Magazine* (April): 4–22
- [60] Mahmoud E (1984) Accuracy in forecasting: A survey. *J. of Forecasting* 3:139–159.
- [61] McClelland JL and Rumelhart DE (1988) *Exploration in Parallel Distributed Processing*. Cambridge, MA, MIT Press.
- [62] McCulloch WS, Pitts W, (1943) A logical Calculus of the ideas Immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5:115–133.
- [63] McNees SK (1985) Which forecast should you use. *New England Economic Review*, July/August: 36–42.

- [64] Minsky ML and Papert S, (1969) *Perceptrons*. MIT Press, Cambridge MA.
- [65] Moody JE (1991) Note on Generalization, Regularization and Architecture Selection in Nonlinear Systems, *Proc. of the IEEE-SP Workshop* : 1–10.
- [66] Moody JE and Darken CJ (1989) Fast learning in networks of locally-tuned processing units. *Neural Computation* 1: 281–294.
- [67] Morozov VA (1984) *Methods for Solving Incorrectly Posed Problems*. Springer-Verlag, Berlin.
- [68] Mozer MC and Smolensky P (1990) Skeletonization: A technique for trimming the fat from a network via relevance assessment. In: *Advances in Neural Information Processing 1*, Touretzky DS (Ed.) : 107–115.
- [69] Murata N, Yoshizawa S, and Amari S (1994) Network Information criterion – Determining the number of Hidden Units for an Artificial Neural model. *IEEE Trans. On Neural Networks* 6: 865–871.
- [70] Natarajan S and Rhinehart RR (1997) Automated Stopping Criteria For Neural Network Training. *Proc. of the 1997 American Control Conf.*, paper #TP09–4.
- [71] Nelson M, Hill T, O’Connor M (1994) Can a neural network be applied to time series forecasting and learn seasonal patterns: An empirical investigation. *Proc. of the 20th Annual Hawaii Intl. Conf on System Sciences*: 649–655.
- [72] Oja E (1982) A simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology* 15: 267–273.
- [73] Palit AK and Popovic D (2000) Nonlinear combination of forecasts using artificial neural network, fuzzy logic and neuro-fuzzy approaches, *FUZZ-IEEE*, 2: 566–571.
- [74] Pineda FJ (1987) Generalisation of back-propagation to recurrent neural networks. *Physical Review Letters* 59: 2229–2232.
- [75] Poggio T and Girosi F (1990) Networks for Approximation and Learning. *Proc. IEEE* 78:1481–1497.
- [76] Powel MID (1988) Radial basis function approximation to polynomials, *Numerical Analysis Proceedings*, Dundee, U.K.: 223–241.
- [77] Prechelt L (1998) Early Stopping – but when? In: Orr GB and Moeller K-R (Eds.), *Neural Networks: Tricks of the Trade*. Springer, Berlin: 55–69.
- [78] Reed R (1993) Pruning Algorithms – A Survey. *IEEE Trans. on Neural Networks* 4: 740–747.
- [79] Rosenblatt F, (1958) The Perceptron: A probabilistic model for information storage and organisation of the brain. *Psych. Review* 65: 386–408.
- [80] Rumelhart DE and McClelland (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* MIT Press, Cambridge, MA.
- [81] Rumelhart DE, Hinton GE, and Williams RJ (1986) Learning internal representation by back-propagation errors. In: Rumelhart DE, McClelland JL, the PDP Research Group(Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, MA.
- [82] Sastry PS, Santharam G, and Unnikrishnan KP (1994) Memory neuron networks for identification and control of dynamic systems. *IEEE Trans. on Neural*
- [83] Schmidhuber J (1989) Accelerated learning in backpropagation net, In: *Connectionism in Perspective*, Elsevier, North Holland, Amsterdam, pp. 439–445.
- [84] Sharda R and Patil RB (1990) Neural Networks as Forecasting Experts: An Empirical Test, *Proc. of the IJCNN Meeting*, Washington: 491–494.
- [85] Shi S and Liu B (1993) Nonlinear combination of forecasts with neural networks. *Proc. of Intl. Joint Conf. on Neural Networks ’93 (IJCNN ’93)*, Nagoya, Japan, 952–962.
- [86] Silva FM and Almeida LB (1990) Speeding-up backpropagation, In: *Advances of Neural Computers*, Eds. Eckmiller R, Elsevier Science Publish. BV., North Holland, pp. 151–158.

- [87] Specht DF (1988) Probabilistic neural networks for classification, or associative memory, Proc. of IEEE Intern. Conf. on Neural Networks, San Diego, 1: 525–532.
- [88] Specht DF (1990) Probabilistic neural networks and the polynomial ADALINE as complementary techniques for classifications. IEEE Trans. on Neural Networks, 1: 111–121.
- [89] Sprecher DA (1965) On the Structure of Continuous Functions of Several Variables. Trans. Amer. Math. Soc. 115:340–355.
- [90] Srinivasan D, Liew AC, Chang CS (1994) A neural network short-term load forecaster. Electric Power Systems Research 28: 227–234.
- [91] Stahlberger A and Riedmuller M (1996) Fast network pruning and feature extraction using the Unit-OBS algorithm. Advances in Neural Information Processing systems (NIPS'96), Denver.
- [92] Stone M (1977) An asymptotic equivalence of choice of model by cross-validation and Akaike's criterion cross validation. J. of the Royal Statistical Soc. B36:44–47.
- [93] Sue CT, Tong LI, and Leou CM (1997) Combination of time series and neural network for reliability forecasting modelling. J. Chin. Inst. Ind. Eng. 14(4): 419–429.
- [94] Tang Z, Almeida de Ch, and Fishwick, PA (1991) Time series forecasting using neural networks vs. Box-Jenkins methodology. Simulation 57(5): 303–310.
- [95] Tiao GC and Tsay RS (1989) Model specification in multivariate time series. J. of the Royal Statistical Society B 51: 157–213.
- [96] Tikhonov AN (1963) On solving incorrectly posed problems and methods of regularisation. Doklady Akademii Nauk USSR 151: 501–504.
- [97] Tseng F-M, Yu H-Ch, and Tzeng G-H (2002) Combining neural network model with seasonal time series ARIMA model. Technological Forecasting.
- [98] Vapnik V (1995) The Nature of Statistical Learning Theory, Springer-Verlag, NY.
- [99] Villiers de J and Bernard E (1992) Backpropagation Neural Nets with one and Two Hidden Layers. IEEE Trans. On Neural Networks : 136–141.
- [100] Vogl TP, Mangis JK, Rigler AK, Zink WT and Allcon DL (1988) Accelerating the convergence of backpropagation method, Biological Cybernetics, vol. 59: 257–263.
- [101] Voort VD, Dougherty M, and Watson M. (1996) Combining Kohonen Maps with ARIMA time series models to forecast traffic flow. Transp. Res. Circ. (Emerg. Technol.) 4C(5): 307–318.
- [102] Wedding II DK and Cios KJ (1996) Time series forecasting by combining RBF networks certainty factors, and the Box-Jenkins model. Neurocomputing 10: 149–168.
- [103] Weigend AS, Rumelhart DE, and Huberman BA (1991) Generalisation by weight-elimination with application to forecasting. Adv. In Neural Information Processing Systems, Morgan Kaufmann, San Mateo, CA 3: 875–882.
- [104] Werbos P (1990) Backpropagation through time what it does and how to do it, Proc. of IEEE, 78(10):1550–1560.
- [105] Werbos PJ (1974) Beyond Regression: New Tool for Prediction and analysis in the Behavioural sciences. Ph.D. Thesis, Harvard University, Cambridge, MA.
- [106] Werbos PJ (1989) Backpropagation and neural control: A review and prospectus. Internat. Joint Conf. of Neural Networks, Washington, 1: 209–216.
- [107] Widrow B and Hoff ME (1960) Adaptive Switching Circuits. In: Anderson J and Rosenfeld E. (eds.) Neurocomputing. MIT Press, Cambridge, MA, 126–134.
- [108] Williams RJ and Zipser D (1989) A learning algorithm for continually running fully recurrent neural networks. Neural Computation 1: 270–280.
- [109] Winkler R and Makridakis S (1983) The combination of forecasts, Journal of the Royal Statistical Society, Series A: 150–157.
- [110] Yang Y (2000) Combining different procedures for adaptive regression, J. of Multivar. Analysis, 74: 135–161.

- [111] Yu X-H, Chen G-A, and Cheng S-X (1995) Dynamic Learning Rate Optimization of the Backpropagation Algorithm. *IEEE Trans. on Neural Networks* 3: 669–677.
- [112] Zhang PG (2003) Time series forecasting using a hybrid ARIMA and neural network models. *Neurocomputing* 50:159–175.
- [113] Zhou S, Popovic D, and Schulz-Ekloff G (1991) An Improved Learning Law for Backpropagation Networks. *IEEE Int. Conf. on Neural Networks*, San Francisco: 573–579.

Selected Reading

- [114] Anderson JA (1972) A Simple Neural Network Generating an Interactive Memory, *Mathematical Biosciences* 14: 197–220.
- [115] Cybenko G (1988) Continuous valued neural networks with two hidden layers are sufficient. Technical Report, Taft University.
- [116] Kohonen T (1972) Correlation Matrix Memories. *IEEE Transactions on Computers* 21: 353–359.
- [117] Kolmogorov AI (1957) On Representation of Continuous Function of Many Variables by Superposition of Continuous Functions of One Variable and Addition. *Dokl. Akad. Nauk USSR* 114:953–956.
- [118] Kurkova V (1991) Kolmogorov's Theorem is Relevant, *Neural Computation*, 3: 617–622.
- [119] Kurkova V (1992) Kolmogorov's Theorem and Multilayer Neural Networks, *Neural Networks* 5: 501–506.
- [120] Moody JE (1992) The Effective Number of Parameters: An Analysis of Generalization and Regularisation in Nonlinear learning Systems. In: *Advances in Neural Information Processing 4* (Moody JE, Hanson SJ, and Lippmann RP (Eds.)), Morgan Kaufman Publ., San Mateo, CA.
- [121] Schwenkler F, Kestler H, Palm G (2001) Three learning phases for radial-basis-function networks. *Neural Networks* 14: 439–458.
- [122] Schwenkler F, Kestler H, Palm G, and Höher M (1994) Similarities of LVQ and RBF learning, *Proc. IEEE International Conference SMC*: 646–651.
- [123] Xiaosong D, Popovic D, and Schulz-Ekloff G (1995) Oscillation-Resisting in the Learning of Backpropagation Neural Networks. 3rd IFAC/IFIP Workshop on Algorithms and Architectures for Real-Time Control, 31 May – 2 June, Ostend, Belgium.

Computational Intelligence in Time Series Forecasting
Theory and Engineering Applications

Palit, A.K.; Popovic, D.

2005, XXII, 372 p., Hardcover

ISBN: 978-1-85233-948-7