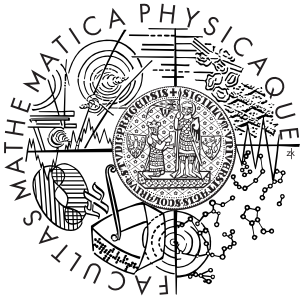
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Lukáš Kúdela

Forecasting Markets using Artificial Neural Networks

Department of Software Engineering

Supervisor:
Mgr. Jan Lánský, Ph.D.

Study program:
Informatics (B1801), General Informatics (1801R008)

2009

I would like to express my gratitude to Mgr. Jan Lánský, Ph.D., the supervisor of this thesis, for showing his sincere interest in its subject from the very beginning.

I declare that the thesis hereby submitted for the Bachelor degree at Charles University in Prague is my own work, and that no other sources than those cited have been used. I cede copyright of this thesis in favour of Charles University in Prague.

Prague, August 7, 2009 Lukáš Kúdela

Contents

Title: Forecasting Markets using Artificial Neural Networks
Author: Lukáš Kúdela
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský, Ph.D.
Supervisor's e-mail address: zizelevak@gmail.com

Abstract: The ability to forecast markets has become a major factor contributing to the success of any enterprise in today's business world. By employing a profitable prediction technique, a company can stay ahead of the game and thus gain a competitive edge on the market. The aim of this work is to investigate the forecasting capabilities of artificial neural networks in general, as well as provide some concrete guidelines that can be followed to effectively utilise their predictive potential. To approach this challenging objective, we have designed and implemented a potent artificial neural network class library, whose ambition is to provide all necessary functionality required in the process of choosing the 'best' forecasting model from all 'candidate' models. On top of this library, a lightweight console application has been built to automate the process of trying out and benchmarking the these models. We have centred our experimentation around a typical economic time series containing trend and seasonal variations.

Keywords: market forecasting, economic time series prediction, artificial neural networks, artificial neural network class library

Názov práce: Predpovedanie trhov pomocou umelých neurónových sietí
Autor: Lukáš Kúdela
Katedra (ústav): Katedra softwarového inžinierstva
Vedúci bakalárskej práce: Mgr. Jan Lánský, Ph.D.
e-mail vedúceho: zizelevak@gmail.com

Abstrakt: Schopnosť predpovedať trhy sa v súčasnoti stala kľúčovým faktorom prispievajúcim k úspechu v podnikaní. Aj používaním sofistikovaných predikčných techník sa spoločnosť môže udržať o krok pred svojou konkurenciou a zaujať tak vedúce postavenie na svojom trhu. Cieľom tejto práce je preskúmať schopnosť umelých neurónových sietí predpovedať trhový vývoj charakterizovaný časovými radmi. Okrem všobecných pozorovaní si práca kladie za cieľ ponúknuť i konkrétnejšie odporúčania ako efektívnejšie využiť predikčný potenciál tohto výpočetného modelu. Súčasťou práce je taktiež návrh knižnice pre prácu s umelými neúronovými sieťami, od ktorej očakávame asistenciu pri dosahovaní spomínaných cieľov. Ambíciou tejto knižnice je poskytnúť všetkú funkcionalitu potrebnú v procese výberu 'najlepšieho' predikčného modelu spomedzi uvažovaných 'kandidátov'. Nad knižnicou bola postavená jednoduchá konzolová aplikácia autamatizujúca proces ich testovania a hodnotenia. V našich experimentoch sme sa zamerali na ekonomické časové rady vykazujúce určitý trend a sezónne variácie.

Kľúčové slová: predpovedanie trhov, predikcia ekonomických časových radov, umelé neurónové siete, knižnica pre prácu s umelými neurónovými sieťami

Chapter 1

Introduction

"The only function of economic forecasting is to make astrology look respectable."

John Kenneth Galbraith

The last few decades of research into the field of artificial neural networks have firmly established their status as potent computational models. Because of their human-like approach to solving tasks, they have traditionally been employed in areas where it is extremely difficult at best, if not downright impossible, to formulate an *exact* step-by-step procedure which is to be followed to obtain the results. In other words, the artificial neural networks excel in areas where the conventional Von Neumann computational models are doomed to fail. Such areas are characterised by tasks that are easily solved by humans. However, if they were asked to describe how exactly they had arrived at the conclusion, they would probably not be able to put their finger on the particular steps involved. These areas include (but are not limited to) pattern classification and prediction.

The forecasting itself goes a somewhat longer way into the past. Practitioners of all sorts of disciplines, be those scientific, engineering or business, have always placed a big emphasis on the importance of being able to predict the future with at least a certain degree of accuracy. The instances of areas where this skill (or lack of thereof) may have a fundamental impact include planning, scheduling, purchasing, strategy formulation, policy making, and supply chain operations [17]. It is no wonder then, that there is a high demand for reliable forecasting methods and a considerable effort

has been spent to both perfect the already existing ones and search for new alternatives.

The earliest attempts at forecasting involved methods that are linear in nature. This linearity enabled their easy implementation and allowed their straightforward interpretation. However, it soon became apparent that they are hopelessly inadequate to describe the complex economic and business relationships. Being linear, they are simply unable to capture any non-linear dependencies among economic variables, which is necessary if one is hoping to design a trustworthy predictive model. The researchers began their search for alternative, non-linear methods and turned their attention to artificial neural networks among others.

The feature of artificial neural networks that appealed the most to forecasters was their inherent *non-linearity*. They correctly postulated that this can turn them into promising forecasting tools, able express the complex underlying relationships present in almost any real-world situation. Furthermore, when using the artificial neural networks, the ability to model linear processes does not have to be relinquished, as they are able to model these as well.

Naturally, non-linearity is not the only convenient property of this computational model. For instance, it has been proven by theoreticians that an artificial neural networks are capable of approximating any continuous function, provided certain requirements imposed on their structure are met. This truly is an essential characteristic that fully justifies the inclusion of artificial neural networks into the top echelon of forecasting methods. Any such method must be able to accurately capture the causal relationship between the predicted process and processes that influence it. As a bonus to what has already been mentioned, artificial neural networks are "data-driven non-parametric methods that do not require many restrictive assumptions on the underlying process from which the data is generated" [17]. This is also a valuable property, since the more parameters a model requires to be specified, the greater the chance of going astray by mis-specificating any of those.

Since the first attempts at applying artificial neural networks to forecasting, the field of *neural forecasting*, as it is commonly referred to, has expanded to one of the major application areas of artificial neural networks [17].

The remainder of this work is organized as follows. In Chapter 2, we

discuss what is meant by 'forecasting markets', define time series and classify the existing forecasting methods. Chapter 3 lays the foundations for our experimentation by providing a theoretical background on the artificial neural networks. Arguably, the most interesting aspect of artificial neural networks – learning – has been devoted a chapter on its own, Chapter 4. Chapters 5 and 6 present the **NeuralNetwork** class library and the **MarketForecaster** console application respectively. The work culminates in Chapter 7, in which we elaborate on the application of artificial neural networks to market forecasting. This chapter can be considered a focal point of the thesis, as it contains the description of experiments we carried out using the **NeuralNetwork** class library and the **MareketForecaster** console application. This is also the place where we present the conclusions we have reached in our experimentation.

Chapter 2

Forecasting Markets

When speaking about forecasting markets, the forecasting of some economic time series is usually meant. In this chapter, we will analyse this meaning in a step-by-step fashion. More precisely, we will

1. define time series,
2. characterise and classify economic time series,
3. explore the constituent components of economic time series,
4. introduce a particular economic time series we will be using in our forecasting attempts,
5. overview the taxonomy of forecasting methods, and
6. discuss the motivation behind forecasting markets using artificial neural networks.

2.1 Time Series

A *time series* consists of a set of observations ordered in time, on a given phenomenon (target variable). Usually the measurements are equally spaced, e.g. by year, quarter, month, week, day. The most important property of a time series is that the ordered observations are dependent through time, and the nature of this dependence is of interest in itself [2].

Formally, a *time series* is defined as a set of random variables indexed in time, $X_1, ..., X_T$. In this regard, an observed time series is denoted by $x_1, ..., x_T$, where the sub-index indicates the time to which the observation x_t pertains. The first observed value can be interpreted as the realization of the random variable x_1, which can also be written as $X(t = 1, \omega)$ where ω denotes the event belonging to the sample space. Similarly, x_2 is the realization of X_2 and so on. The T-dimensional vector of random variable can be characterized by different probability distribution [2].

For socio-economic time series the probability space is *continuous*, and the time measurements are *discrete*. The frequency of measurements is said to be *high* when it is daily, weekly or monthly and to be *low* when the observations are quarterly or yearly [2].

2.2 Time Series Decomposition Models

An important objective in time series analysis is the decomposition of a series into a set of non-observable (latent) components that can be associated to different types of temporal variations. Time series is composed of four types of fluctuations [2]:

1. A *long-term tendency* or *secular trend.*
2. *Cyclical movements* super-imposed upon the long-term trend. These cycles appear to reach their peaks during periods of industrial prosperity and their troughs during periods of depressions, their rise and fall constituting the business cycle.
3. A *seasonal movement* within each year, the shape of which depends on the nature of the series.
4. *Residual variations* due to changes impacting individual variables or other major events such as wars and national catastrophes affecting a number of variables.

Traditionally, the four variations have been assumed to be mutually independent from one another and specified by means of an *additive decomposition model*

$$X_t = T_t + C_t + S_t + I_t,$$

where X_t denotes the observed series, T_t the long-term trend, C_t the cycle, S_t seasonality and I_t the irregulars.

If there is dependence among the latent components, this relationship is specified through a *multiplicative model*

$$X_t = T_t \times C_t \times S_t \times I_t.$$

In some cases, mixed additive-multiplicative models are used.

2.3 International Airline Passengers Data

The economic time series used in thesis is the famous *international airline passengers data* listed in [1] as series G. The series tells the total number of international airline passengers for every month from January 1949 to December 1960. There are several compelling reasons for choosing this particular series, namely:

- Demand forecasting (in this case, the demand for airline transportation) is one of the most important areas of economic forecasting.
- The series follows a rising trend and displays a seasonal variation (more precisely, *multiplicative seasonality*) making it a first-grade choice to demonstrate the forecasting capabilities of neural networks.
- It can be considered a benchmark series, since several groups of researchers have already tried to forecast it, e.g. [11] or [5].
- Its length is representative of data found in forecasting situations (several years of monthly data).
- It is available free of charge.

2.4 Forecasting Methods and their Classification

A time series observation is made up of a *systematic component*, or *signal*, and a *random component*, or *noise*. Unfortunately, neither of these two components can be observed individually. A *forecasting method*, or *forecasting*

model, attempts to isolate the systematic component, so that a reasonable prediction (based on this component) can be made.

Forecasting methods can be classified into two top-level categories:

- *Subjective forecasting methods*
- *Objective forecasting methods*

As their names suggest, the *subjective forecasting methods* involve human judgement (hence also their alias, *judgemental forecasting methods*). They typically operate with either a small number of highly regarded opinions (a panel-of-experts approach, e.g. the *Delphi method*) or a considerably larger (statistically significant) number of less valuable opinions (a wisdom-of-the-crowds approach, e.g. *conducting a statistical survey*).

2.4.1 Objective Forecasting Methods

The *objective forecasting methods*, on the other hand, attempt to limit the human involvement as much as possible. However, as we will see later, in search for a definitive forecasting solution, they are no silver bullet. At least for now, one simply can not hope to automate the forecasting process to the point where it is devoid of any human interpretation.

Note that neither of the two principal approaches (objective or subjective) has been proven superior. Also, neither of them alone is sufficient to make reliable predictions, i.e. those that ultimately generate profit. After all, not even the their most brilliant fusion can possibly guarantee that. Forecasts obtained by employing only single-category methods are likely to be inaccurate. Therefore, these two families of forecasting methods should be though of as complementary rather than clashing. Not surprisingly, the most successful (meaning the most profitable) contemporary forecasting methodologies use sophisticated mixtures of both.

The objective forecasting methods are further classified into two categories:

- *Causal forecasting methods*
- *Time series forecasting methods*

The *causal forecasting methods*, also known as the *econometric methods*, assume that, while attempting to forecast a data-generating process, it *is* possible to identify a set of causal processes that influence it. Under such circumstances, the future data can be forecast using the past data of the causal processes.

2.4.2 Time Series Forecasting Methods

The *time series forecasting methods* assume that, while attempting to forecast a data-generating process, it *is not* possible (or affordable) to identify a set of causal processes that influence it. Under such circumstances, the future data can only be forecast using the process' own past data.

2.5 Why Use Artificial Neural Networks to Forecast Markets?

In economics, more often than not, it is extremely difficult to put one's finger on the causal processes bearing an influence on some data-generating process. Moreover, their interdependency is likely to be non-linear. The artificial neural networks are *universal function approximators*. A two-layer perceptron can approximate any continuous (linear or non-linear) function to any desired degree of accuracy. Therefore, they are able to learn (and generalize), linear and non-linear time series patterns directly from historical data, i.e. perform a *non-parametric data-driven* econometric modelling. Additionally, the underlying connectionist principles at work guarantee a fairly high degree of *error-robustness*. These abilities and properties qualify them as promising (both causal and time series) forecasting tools.

Chapter 3

Artificial Neural Networks

An *artificial neural network (ANN)*, or simply a *neural network (NN)* is a computational model imitating the structure and/or functionality of biological neural networks. It processes information using a *connectionist computation paradigm*. A high-quality source of neural network theory [10] is relied on and used extensively throughout this chapter.

Neural networks are essentially simple mathematical models defining a function

$$f : \mathbb{R}^{n_i} \to \mathbb{R}^{n_o},$$

where $n_i \in N$ is the number of *input neurons* and $n_o \in N$ is the number of *output neurons*. Each neural network model (or type) corresponds to a class of such functions $\mathcal{F}$. Naturally, a particular neural network instance corresponds to a particular function $f \in \mathcal{F}$.

The function f (the output of the network) is defined as a composition of other functions g_i which are themselves compositions of other functions. At the bottom of the composition hierarchy are simple values (the inputs of the network). The function f, however complex, can therefore be composed from an arbitrary number of very simple functions. This notion can be conveniently illustrated by an oriented graph.

- The graph itself represents the actual function f,
- the graph's vertices represent the functions g_i, and
- the graph's (oriented) edges represent the dependencies among functions g_i. An edge from vertex (function) g_i to vertex (function) g_j

represents the fact that the function g_j requires the output of the function g_i as one of its inputs.

Since the inspiration for artificial neural networks comes from biological neural networks, the terminology has been assimilated as well. We use the term *neural network* to refer to the function f (the graph itself), *neuron* to refer to any of its comprising functions g_i, (any of the graph's vertices) and *synapse* to refer to any of the involved dependencies (any of the graph's edges).

The composition of function f as well as the compositions of non-input functions g_i employ some *aggregation function* and some *activation function*:

$$f(x) = f_{activation}(f_{aggregation}([w_i], [g_i(x)])).$$

The *aggregation function* defines the way in which the inputs of a neurons are aggregated into its *inner potential* (or *input*). By far, the most widely used aggregation function is the *weighted sum* – the linear combination of synapse weights w_i and neuron's inputs g_i:

$$f_{aggregation}([w_i], [g_i(x)]) = \sum_i w_i g_i(x).$$

The *activation function*, or *transfer function*, defines the way in which a neuron's inner potential activates its *state*, in other words, the way in which the neuron's input is transferred to its output. It is an abstraction representing a rate of action potential firing in the cell. The most commonly used activation functions are the *logistic function)*:

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

and *hyperbolic tangent*:

$$f_{activation}(x) = \frac{e^{2x} + 1}{e^{2x} - 1}.$$

They both continuously approximate the *Heaviside step function*, also known as the *unit step function*, while being differentiable. This is an essential property when we want to train the network using a learning algorithm implementing a form of gradient descent.

Sometimes a *linear function* with positive slope (e.g. the *identity function*) is used as an activation function in the output layer. This overcomes the range limitation imposed by the two previously mentioned functions. While the *logistic logistic* function and the *hyperbolic function* can yield only values from intervals $(0, 1)$ and $(-1, 1)$ respectively, the rage of a linear function is unbounded.

Depending on whether their graph is a DAG or not, neural networks can be classified into two main classes:

- *feedforward neural networks* (their graph is a DAG) and
- *recurrent neural networks* (their graph is not a DAG).

As this thesis focuses on implementing a time series forecasting method, we will not attempt to identify any causal processes that would bear an influence on the process being predicted. Thus, we will not have to account for the influence the predicted process might have on the causal processes in turn. For this reason, a recurrent neural network will not be necessary and a simple feedforward neural network will suffice.

3.1 Feedforward Neural Networks

A *feedforward neural network (FNN)* is a neural network where connections (synapses) between units (neurons) do not form a directed cycle (i.e. their graph is a DAG). This type of network only propagates data from earlier processing stages to later processing stages.

The simplest of all feedforward neural network models (and neural networks as such) is the *single-layer perceptron (SLP)*, or simply *perceptron*. A perceptron consists of a single layer of output neurons, also known as *linear threshold units*, to which the inputs are fed directly via a matrix of weights. To train a perceptron, a simple learning algorithm, implementing a form of gradient descent, the *delta rule*, can be used [15].

Unfortunately, perceptron is only capable of correctly classifying training patterns that are *linearly separable*. Unfortunately this is rarely the case in practical situations. In fact, trivial mappings exist that are *linearly inseparable*. Interest in neural networks hit a rock bottom in 1969 when Marvin Minsky and Seymour Papert showed that it was impossible for a perceptron

to learn even the most trivial of all linearly inseparable training sets, the *XOR logical function*.

3.2 Multi-layer Perceptron

Arguably the most famous and the most widely used neural network model is the *multi-layer perceptron (MLP)* model. This model is a generalisation of the single-layer perceptron for an architecture with hidden layers. In other words, apart from the input and output layers, the multi-layer perceptron can contain an arbitrary (although customarily very small) number of hidden layers. The introduction of hidden layers removes the constraint imposed by the single-layer perceptron model regarding the linear separability of the data set. Firstly, we will describe the basic features of multi-layer perceptron and then we will suggest some possible modifications.

3.2.1 Organizational Dynamics

The organizational dynamics of a multi-layer perceptron (usually) specifies a fixed multi-layer feedforward neural network topology. Since the exact meaning of hidden neurons and their synapses (necessary to design a specialized topology) is unknown, a two-layer or a three-layer network is used as the standard.

The following notation we will be using to describe multi-layer perceptrons is adapted from

- X denotes the set of all n input neurons,
- Y denotes the set of all m output neurons,
- i, j, etc. are the indices used to denote neurons,
- ξ_i denotes the real inner potential (input) of the neuron i,
- y_i denotes the real state (output) of the neuron i,
- w_{ji} denotes the the real synaptic weight of the connection from the neuron i to the non-input neuron j,

- $w_{j0} = -h_j$ denotes the bias (the threshold h_j with the opposite sign) of the non-input neuron j corresponding to the formal unit input $y_0 = 1$,
- $j_{\leftarrow}$ denotes the set of all neurons for which a connection from them to the neuron j exists, hence the input neurons of the neuron j (including the formal unit input $0 \in j_{\leftarrow}$), and
- $j^{\rightarrow}$ denotes the set of all neurons, for which a connection from neuron j to them exists, hence the output neurons of the neuron j (i.e. the neurons for which the neuron j is an input neuron).

3.2.2 Active Dynamics

In the active mode, the multi-layer network evaluates the function:

$$y(w) : \mathbb{R}^n \to (0,1)^m,$$

determined by the synaptic weight configuration w, for a given input. The computation runs according to the following discrete active dynamics. In time 0, the respective states of the input neurons y_i ($i \in X$) are assume the values of the network input and the remaining neurons have their states undefined. In time $t > 0$, the real values of inner potentials

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

of all neurons j, whose inputs (from $j_{\leftarrow}$) have their state determined, are calculated. This means that in the time t, the neurons in the t-th layer are updated. Using the inner potential ξ_j, the real state $y_j = \sigma(\xi_j)$ of the neuron j is determined by a differentiable activation function $\sigma : \mathbb{R} \to (0,1)$ of standard sigmoid (continuously approximating the Heaviside function):

$$\sigma(\xi) = \frac{1}{1 + e^{-\lambda \xi}}.$$

The differentiability of the transfer function and the differentiability of the network function that follows from it is essential for the backpropagation learning algorithm. The real parameter of *gain* λ determines the non-linear growth (for $\lambda < 0$ decay) of the standard sigmoid in the neighbourhood of zero (for $\lambda \to \infty$ we get the Heaviside step function), i.e. the measure of

'decisiveness' of a neuron [10]. In the basic model, $\lambda = 1$ is usually assumed. However, generally every non-input (i.e. hidden or output) neuron j can have a different gain λ_j (and therefore a different activation function σ_j).

This way, provided that the multi-layer network is acyclic (connected) network, all neurons' outputs are calculated one by one. The active mode is over, when the state of all neurons in the network (especially the output neurons) is established. The output neurons yield the network output, i.e. the value of the network function for a given input.

3.2.3 Adaptive Dynamics

In the adaptive mode, the desired network function of multi-layer perceptron is given by a training set:

$$T = \{(\vec{x}_k, \vec{d}_k) | \frac{\vec{x}_k = (x_{k1}, \ldots, x_{kn}) \in \mathbb{R}^n}{\vec{d}_k = (d_{k1}, \ldots, d_{km}) \in (0,1)^m}; k = 1, \ldots, p\},$$

where x_k is a real input of the k-th training pattern and d_k is the corresponding real desired output given by the teacher. The goal of the adaptation is that for each input $x_k; k = 1, \ldots, p$ from the training set, the network (in its active mode) answers with the expected output, i.e.:

$$\vec{y}(\vec{w}, \vec{x}_k) = \vec{d}_k; k = 1, \ldots, p.$$

The network error $E(w)$ with respect to a training set is defined as a sum of the partial network errors $E_k(w)$ with respect to the individual training patterns and depends on the configuration of the network w:

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(w).$$

The partial network error $E_k(w)$ with respect to the k-th training pattern is proportional to the sum of the squares of the differences between the actual values of the network output for the k-th training pattern and the corresponding expected output values of that pattern:

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} (y_j(\vec{w}, \vec{x}_k) - d_{kj})^2.$$

The goal of the adaptation is to minimise the network error in the weight space. Since the network error directly depends on the complicated non-linear composite function of the multi-layer network, this goal poses a non-trivial optimisation problem. In the basic model, a simple gradient descent method (requiring the differentiability of the error function) is employed to solve it [10].

Chapter 4

Learning

Before a multi-layer perceptron (or any other neural network for that matter) can successfully be used, it has to undergo learning. *Learning*, in its most general meaning, refers to a process, during which the following characteristics of a neural network are determined:

- the architecture, namely the number of hidden layers and the numbers of hidden neurons comprising each hidden layer, and
- the synaptic weights.

Note that apart from those mentioned above, there are other architectural aspects to be considered. These involve

- the number of input and output neurons and
- the choice of aggregation and activation functions for each non-input layer.

However, these are usually addressed by the designer of the multi-layer perceptron before committing the neural network to the actual learning process. This is because they are either

- determined by the task itself (the number of input and output neurons) or

- they are fundamental to the task and require the irreplaceable attention of the designer (the choice of activation function for each non-input layer).

In this thesis, only learning algorithms operating on fixed-topology multi-layer perceptrons are discussed. Hence, when referring to the learning of a multi-layer perceptron, only the adjustment of synaptic weights is implied. Its more general meaning will either be apparent from the context or stated explicitly.

In the previous chapter, we have established that a particular neural network (parametrized by its synaptic weights) realizes a certain function f. Since, coming from this perspective, a particular neural network is something of a 'physical embodiment' of the function it computes, it is convenient not to distinguish between that particular neural network and its corresponding function. According to the same logic, a distinction between a neural network model and a corresponding class of functions $\mathcal{F}$ is not made in the following explanation.

More specifically speaking, learning is a process of using a set of observations (known as *data patterns* or *training patterns*) to find the function f^* (known as the *optimal function* or the *optimal solution*) from a given class of functions $\mathcal{F}$, that solves the specific task in an 'optimal sense'.

4.1 Cost Function

In order to have an exact understanding of what 'optimal sense' means, and hence be able to find the optimal function, we require a *cost function* (also known as *objective function*) $C : \mathcal{F} \to \mathbb{R}$ such that no solution has a cost less that the optimal solution, i.e. $(\forall f \in \mathcal{F})(C(f^*) \leq C(f))$. In plain English, the cost function provides a measure of 'goodness' to a class of functions $\mathcal{F}$. All learning algorithms search the solution space for a function that has the smallest possible cost.

The cost function is typically defined as a function of the observations - a statistic to which only approximations can be made, for example

$$C = E[(f(x) - y)^2]$$

where (x, y) are the data pairs drawn from some distribution $\mathcal{D}$ [14].

In practice however, only a finite number $k \in \mathbb{N}$ of training patterns is usually available. Because of this limitation, the cost is minimised over a sample of the data distribution (known as training set) instead of the entire data distribution:

$$\hat{C} = \frac{1}{k} \sum_{i=1}^{k} (f(x_i) - y_i)^2.$$

Unquestionably, the cost function may be picked arbitrarily, but more often than not, its choice is aided by the desirability of some of its properties (e.g. convexity) or the particular formulation of the problem itself. Ultimately the cost function will depend on the task we wish to perform.

In some situations, an infinite number of training patterns (an infinite training set) is theoretically available. The training patterns are provided one at a time and typically represent data obtained by periodically sampling some continuous event. Under such circumstances, some form of *on-line learning* has to be performed - the cost is partially minimised with each new training pattern encountered. This type of learning is especially appropriate if the distribution $\mathcal{D}$, from which the training patterns are drawn, changes slowly over time [14].

Since any time series is a result of periodical sampling of some continuous event, it can be used to provide a (potentially) infinite number of training patterns piece by piece. However, we have decided not to pursue this approach and instead opted for so-called *batch learning*.

4.2 Learning Paradigms

Three main learning paradigms are generally recognized in the neural network literature:

- supervised learning,
- unsupervised learning, and
- reinforcement learning.

Note that there is no strictly defined relation between neural network models and learning paradigms, i.e. none of the learning paradigms is associated with any particular type of neural network, and none of the neural network types is tied to any particular learning paradigm.

Depending on the task to solve, we may have a *(fully) labelled*, *partially labelled* or *unlabelled* training set at our disposal. Every training pattern contained in a labelled training set is labelled, i.e. contains both an input vector and a (desired) output vector. Unlabelled training set, on the other hand, consists of only unlabelled training patterns, i.e. training patterns containing only an input vector. Partially labelled training sets contain a mixture of labelled and unlabelled training patterns.

Labelled training sets are used during supervised learning, partially labelled training sets during semi-supervised learning and unlabelled training sets during unsupervised learning. Hence, each learning paradigm is an abstraction of a particular group of learning tasks.

When training a multi-layer perceptron to forecast time series based on past observation, we know what the correct answer to each input vector from the past should have been by looking at the series itself, i.e. every input vector has a desired output vector associated with it. Therefore, a labelled training set can be constructed for a supervised learning session.

4.3 Supervised Learning

In supervised learning, we are presented with a set of training patterns (v_i, v_o), where $v_i \in V_i$ is the *input vector* and $v_o \in V_o$ is the *(desired) output vector*, and the goal is to find a function $f : V_i \rightarrow V_o$, where V_i is the vector space of all input vectors and V_o is the vector space of all output vectors that matches all training patterns, i.e. when presented with a training pattern's input vector, it will return its desired output vector. To put it differently, we would like to infer the mapping implied by the data.

Ideally, the cost function should penalize the mismatch between the inferred mapping and the mapping contained in the training set. It may be custom-tailored by a designer to inherently contain prior knowledge about the problem domain. A favourite among the cost functions is the *mean-squared error* presented in section 4.1 of this chapter.

A convenient way to think about this paradigm is to imagine a teacher guiding the learning process of a network. The teacher knows what is the correct output to every input from the training set. Hence, he is able to train the network by providing it with feedback.

4.4 Learning Algorithms

Learning algorithm, or *training algorithm*, is an algorithm which trains a neural network. Numerous learning algorithms, *deterministic* or *stochastic* in nature, have been proposed.

A typical *deterministic learning algorithm* implements some form of *gradient descent* (also known as *steepest descent*). This is done by simply taking the derivative of the cost function with respect to a network's configuration (synaptic weights) and then changing that configuration in a gradient-related direction [14]. In this thesis, the most widely used of all deterministic learning algorithms, *Backpropagation*, will be discussed.

As opposed to the deterministic learning algorithms, the stochastic learning algorithms introduce a certain amount of 'randomness' to the learning process.[1]

The stochastic learning algorithms introduced in this thesis are each an implementation of some metaheuristic. A *metaheuristic* (greek for 'higher order heuristic') is a heuristic method for solving a very general class of computational problems by combining user-given black-box procedures – usually heuristics themselves – in the hope of obtaining a more efficient or more robust procedure.

In this thesis, some of the most interesting metaheuristics – *Genetic Algorithms*, *Simulated Annealing* and *Ant Colony Optimization* – are introduced. All of the these were originally designed to solve combinatorial optimization problems. Our goal was to implement them as learning algorithms, which means applying them to the continuous domain.

In should be noted that the learning process is not restricted to using a single learning algorithm. The learning algorithms may be combined to form a *hybrid learning algorithm*. The individual learning algorithms forming a hybrid do not necessarily have to belong to the same category (deterministic or stochastic). Particularly noteworthy hybrid learning algorithms that have been proposed include:

- a genetic algorithm where each individual performs a few iterations of 'hill-climbing' before being evaluated (GA-BP) and

[1] Of course, one may point out, and correctly so, that Backpropagation learning algorithm also employs a pseudo-random number generator. However, it does so only to initialize the synaptic weights; after the learning itself begins, the algorithm proceeds in a deterministic fashion.

- a simulated annealing where the parameters are controlled by a genetic algorithm (SA-GA).

4.4.1 Backpropagation

Backpropagation (abbreviation for 'backwards propagation of errors') is a deterministic supervised learning algorithm. Since it employs a form of gradient descent in search for the network error function's minimum, it requires that all non-input neurons in the network have differentiable activation functions.

Šíma and Neruda [10] give the following account of the adaptation procedure. In the beginning of the adaptation process, in time 0, the weights of the configuration $\vec{w}^{(0)}$ are set to random values from zero's neighbourhood, e.g. $w_{ji}^{(0)} \in [-1, 1]$. The adaptation runs in discrete time steps, which correspond to the *training cycles* (also referred to as *training epochs*). A new configuration $\vec{w}^{(t)}$ in time $t > 0$ is calculated as follows:

$$w_{ji}^{(t)} = w_{ji}^{(t-1)} + \Delta w_{ji}^{(t)}$$

where the change of weights $\delta w_{ji}^{(t)}$ in time $t > 0$ is proportional to the negative gradient of the error function $E(w)$ at the point $\vec{w}^{(t-1)}$:

$$\Delta w_{ji}^{(t)} = -\epsilon \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)}),$$

where $0 < \epsilon < 1$ is the learning rate.

The *learning rate* ϵ is a measure of the training patterns' influence on the adaptation (something of a 'willingness' to adapt) and is a rather delicate parameter. A learning rate that is 'too low' results in slow convergence and the network error may never drop below the maximum tolerable value even if the computational budget is generous. On the other hand, a learning rate that is 'too high' almost inevitably causes divergence and the network error rises above all imaginable bounds. As the learning rate requires a painstaking manual fine-tuning for each particular task, the most helpful recommendations one one can hope to get are nothing more than rules of thumb.As such, they are not intended to be strictly accurate or reliable for every situation.

The most helpful recommendations we have come across suggest maintaining a separate learning rate ϵ_{ji} for each weight w_{ji} in the network. At

the beginning of the adaptation, the values of ϵ_{ji} are recommended to be chosen conservatively (e.g. in the order of thousandths or ten thousandths), especially for larger topologies. In case of a successful convergence in time $t > 1$ (i.e. if $\delta w_{ji}^{(t)} \delta w_{ji}^{(t-1)} > 0$), the learning rate can be increased linearly:

$$\epsilon_{ji}^{(t)} = \kappa \epsilon_{ji}^{(t-1)},$$

for $\kappa > 1$ (e.g. $\kappa = 1.01$). Otherwise, in case of an apparent divergence, it can be decreased exponentially:

$$\epsilon_{ji}^{(t)} = \epsilon_{ji}^{(t-1)}/2.$$

During the backpropagation, the layers are updated *sequentially*, starting with the output layer and proceeding with the hidden layers. However the computation within the context of one layer can be *parallelized*.

The Backpropaagtion learning algorithm, as described above, has two major drawbacks. The first drawback is that it can be unacceptably slow, even for relatively small topologies. Plethora of modifications, achieving varying degrees of success, have been proposed to remedy this situation. We have chosen to implement one of the simplest, yet most promising in the **NeuralNetwork** class library.

Momentum It has been observed that a low learning rate results in a slow convergence and a high learning rate in a divergence. A simple and often-used modification that seeks to eliminate this shortcoming, takes into consideration the previous weight change – so-called *momentum* – when calculating the current weight change in the direction of the error function gradient:

$$\delta w_{ji}^{(t)} = -\epsilon \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t-1)}) + \alpha \delta w_{ji}^{(t-1)},$$

where $0 < \alpha < 1$ (e.g. $\alpha = 0.9$) is the parameter of momentum that determines the extent of previous changes's influence. This augmentation enables the gradient descent methods to 'slide' the surface of the error function $E(\vec{w})$ faster.

The second major drawback of the Backpropagation learning algorithm (and all learning algorithm based on the methods of steepest descent) is that the search for the *global minimum* may become 'stuck' in a *local minimum*. The fact that a convergence to a local minimum can not be recognised (and

possibly the learning process repeated) makes this drawback even more serious. The **NeuralNetwork** class library attempts to lower the probability of this happening by two simple remedies.

Multiple Runs The Backpropagation learning algorithm is run multiple times, each time starting with a random, and hopefully different, initial network configuration. Throughout the multiple runs, the best network configuration found so far is maintained. The more runs, the lower the probability that a gradient descent from all of them led only to local minima.

Jitter At regular intervals, i.e. once every some predetermined number of training cycles, the synaptic weights of the network are perturbed by having a small random value added to or subtracted from them. The idea is to introduce a healthy amount of randomness to the whole process, thus enabling the network to escape from local minima, should it become trapped in one.

4.4.2 Genetic Algorithm

Genetic algorithms (GAs) are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information. Genetic algorithms are often viewed as function optimizers, although the range of problems to which genetic algorithms have been applied is quite broad.

In his popular tutorial [13], Whitley summarises what he calls the *canonical genetic algorithm* as follows. An implementation of a genetic algorithm begins with a population of (typically random) chromosomes. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes which represent a better solution to the target problem are given more chances to 'reproduce' than those chromosomes which are poorer solutions. The 'goodness' of a solution is typically defined with respect to the current population.

4.4.3 Simulated Annealing

The problem of getting stuck in a local minimum is successfully dealt with by the *simulated annealing (SA)* method. In contrast with the hill-climbing algorithm (for our continuous case, the gradient descent algorithm) it can also accept a worse solution than the current one with certain probability, determined by the *Metropolis criterion*. Another important difference is that instead of exploring the neighbourhood and choosing the best solution (for our continuous case, calculating the gradient) the current solution is transformed into a new solution by a stochastic operator [12].

The inspiration for the method (and its name) comes from a physical process of annealing a solid body used to reduce its inner (crystallographic) defects. The body is heated to a high temperature and the slowly cooled. This enables its atoms to overcome local energetic barriers and settle into equilibria positions. Steady temperature decrease results in the equilibria positions of the atoms being stabilized. Hence, upon reaching the final temperature (which is considerably lower than the initial temperature), all atoms will have attained their equilibria positions and the body will not contain any inner defects.

4.4.4 Ant Colony Optimization

Ant Colony Optimization (ACO) metaheuristic, itself a member of the *Swarm Intelligence (SI)* family of computational methods, is an umbrella term for optimization algorithms inspired by the ants' foraging behaviour. Since their conception in 1992 by Marco Dorigo, several variations on the *canonical ant colony optimization* have been proposed. However, as Socha points out in [9] adhere to the following basic ideas:

- Search is performed by a population of individuals - simple independent agents.
- There is no direct communication between the individuals.
- The solutions are constructed incrementally.
- The solution components are chosen probabilistically, based on *stigmergic information*.

Stigmergy is a mechanism for spontaneous, indirect coordination between agents (in our case ants), where the trace left in the environment (in our case the pheromone trail) by an agent' action stimulates the performance of a subsequent action, by the same or a different agent. Stigmergy is a form of self-organization. It produces complex, apparently intelligent structures (in our case finds the shortest paths between the ant colony and a source of food), without need for any planning, control, or even communication between the agents. As such it supports efficient collaboration between extremely simple agents, who lack any memory, intelligence or even awareness of each other[16].

Even though ACO had initially been used for solving combinatorial optimization problems (e.g. the travelling salesman problem), it has since been extended (without the need to make any major conceptual changes) to continuous and mixed-variable domains.

Chapter 5

NeuralNetwork (Class Library)

The **NeuralNetwork** class library presented in this chapter has originally been conceived as nothing more than an auxiliary software tool to be developed for the purposes of this thesis. However, over time it has evolved into a capable neural network class library and took a more central stage in the following text. It provides almost all the necessary functionality one needs when attempting to forecast markets by means of simple time series prediction. Only a rather compact project needs to be built on top of it, as will be evident later when we introduce the **MarketForecaster** console application.

Instead of using a third-party neural network library, we have opted to design and implement our own. This decision was motivated by the following ideas.

First and foremost, we wanted to initiate the work on a neural network library that would be *following the principles of object oriented programming* as much as possible. In practice, this means that in situations where a trade-off between a design and performance issue had to be made, we mostly elected to protect the design at the cost of compromising performance. It has to be noted though that this approach is not a typical one, as the field of neuro-computing is characterised by a high demand for speed. We did not however attempt to compete with some of the fastest libraries currently available.

We also wanted the library to be as *accessible* to a researcher new to the field as possible. Some of the best neural network libraries in the world

appear rather intimidating when first seen.[1] Our library, on the other hand, offers an *elegant and straightforward API*, making it an ideal companion to a introductory class about neural networks.

It is often said, that only by writing the code itself, a novice to the field can make sure he or she really understands the problem. So the last (but certainly not the least compelling) reason to write the library ourselves revolved around the desire to *fully grasp the theoretical intricacies* involved in the field of neural networks. True, the task was challenging, but left us considerably more confident in our understanding.

5.1 Requirements Documentation

NeuralNetwork is an (artificial) neural network class library. It enables the user to

- build training sets,
- build neural networks,
- create learning algorithms (known as *teachers* in the library jargon),
- train neural networks, and
- (most importantly) use the trained neural networks.

5.1.1 Supported Features

Currently, labelled training sets, multi-layer perceptrons, one deterministic (Backpropagation), and three stochastic (based on genetic algorithm, simulated annealing and ant colony optimization) teachers and are supported 'out-of-the-box'. Combinations of a labelled training set, a multi-layer perceptron and Backpropagation teacher make up for the majority of neural networks applications today. However, the library is designed to facilitate the extension of available classes by an experienced user, should he or she require a different type of training set, neural network or teacher.

[1] Of course this is perfectly justified by the amount of computational power they offer.

5.2 Design and Architecture Documentation

NeuralNetwork's most important class is the NETWORK class. It provides an abstraction of a multi-layer perceptron. Every multi-layer perceptron consists of a number of layers (the LAYER class). At the very least it has to contain an input layer (the INPUTLAYER) and the output layer (the ACTIVATIONLAYER class). Furthermore, it may contain an arbitrary (although usually reasonably small) number of hidden layers (also represented by the ACTIVATIONLAYER class). A decision has been made to provide a separate class for the input layer, as the input layer is not a full-fledged layer (e.g. it does not have an activation function).

Any layer is basically just a collection of neurons (the NEURON class). The input layer holds together all the network's input neurons (the INPUTNEURON class), a hidden layer holds together a number of hidden neurons, and the output layer groups all the network's output neurons. Both hidden neurons and output neurons are abstracted into the ACTIAVTIONNEURON class. Again, two distinct classes (although sharing a common base class) representing neurons exist as there are some characteristics unique to the input neurons (e.g. they can be assigned the input to the network) as well as other unique to the hidden and output neurons (e.g. the ability to evaluate their state).

In a multi-layer perceptron, two neurons from neighbouring layers are connected by a (uni-directional) synapse (the SYNAPSE class). All the synapses between any two (hence not necessarily neighbouring) layers are held together in a collection called *connector* (the CONNECTOR class). The connector has no real analogy in the artificial neural network theory. Its existence is motivated by an attempt to make the process of building a network as straightforward as possible.

Two design aspects of the **NeuralNetwork** class library are definitely worth mentioning: *blueprints* and *decorable components*.

Despite our ultimate ambition to design a flexible library based on simple components (neuron, layer, synapse, connector, etc.) that could be assembled together to form custom-built neural networks, we have (at least for the time being) decided to impose certain restrictions on the process of building a neural network. As long as recurrent neural networks are not supported, it is imperative to prevent the user from building structures that would contain cycles in their graphs.

Through a series of *layer blueprints* (the LayerBlueprint, InputLayerBlueprint and ActivationLayerBlueprint classes) and *connector blueprints* (the ConnectorBlueprint class) constituting a *network blueprint* (the NetworkBlueprint class), the client can specify which layers are interconnected, i.e. he or she can determine the *flow of information* in the network. There is no way to build a neural network directly, one has to use a valid blueprint. A positive externality is that a blueprint, once created, can be used to comfortably construct any number of (identically-structured) neural networks. This is especially handy in situations where population-based computational models are employed.

All publicly available neural network class libraries we have seen tend to mix the concepts of neural networks and training algorithms, typically by tying a particular neural network type, say the multi-layer perceptron, with a particular learning algorithm. They do this by including state or functionality associated with that particular learning algorithm into the very definition of some concrete neural network class, e.g. a change in weight into a synapse. The problem is that while a change in weight is inherent to a synapse being trained by Backpropagation, it is *not* inherent to a synapse as such. We have decided not to repeat this conceptual mistake and set out to *completely separate* neural network from learning algorithms. We anticipated that if we succeed, we might afford the luxury of having the same network (meaning the same instance during run-time) trained by as many teachers as we desire. Furthermore, a teacher once constructed may enjoy a more productive lifetime by teaching a whole 'class' of neural networks.

Simply inheriting a new class, say BackpropagationNetwork from the Network class was just not enough, because a Network object, once instantiated can not be 'upgraded' to an instance of its classes sub-class. Thus a Network object could never become a BackpropagationNetwork for instance. We required a way to promote an already instantiated object into a new, more complex one, i.e. to enrich its state and functionality at the time of execution. We turned our attention to design patterns and quickly found out that the *Decorator* design pattern, providing a "flexible alternative to subclassing for extending functionality" [4], advertised exactly what we were looking for.

Inspired by the Decorator design pattern, decided to make every component in the **NeuralNetwork** class library a decorable one. As such, they can be created with minimal state and functionality and later dynamically

acquire some additional responsibilities. The BackpropagationTeacher class serves as an example of how every type of component (neuron, layer, synapse, connector, and even network itself) can be promoted to support new state (e.g. BackpropagationConnector adds the momentum) and functionality (e.g. BackpropagationLayer adds the Backpropagate function).

5.3 Technical Documentation

NeuralNetwork is a class library written in C# programming language (version 3) and targeted for Microsoft .NET Framework (version 3.5). It is being developed in Microsoft Visual C# 2008 Express Edition as a part of the **MarketForecaster** solution - an entire solution aimed at forecasting markets using artificial neural networks.

5.3.1 Dependencies

NeuralNetwork class library itself uses the following class libraries:

- **GeneticAlgorithm** (a genetic algorithm metaheuristic class library)
- **SimulatedAnnealing** (a simulated annealing metaheuristic class library)
- **AntColonyOptimization** (an ant colony optimization metaheuristic class library)

These were designed and developed as standalone and compact class libraries to support the main **NeuralNetwork** class library and are part of the **MarketForecaster** solution. **NeuralNetwork** class library does not use any third-party class libraries.

5.3.2 Relevant Projects from the MarketForecaster solution

Apart from the **NeuralNetwork** class library itself, the **MarketForecaster** solution contains the following relevant projects:

- **NeuralNetworkTest** (a **NeuralNetwork** class library test harness)
- **NeuralNetworkDemo** (a **NeuralNetwork** class library demonstration, containing the example from the End User Documentation section of this chapter)

5.4 End User Documentation

To use the **NeuralNetwork** class library, the client has to follow the standard procedure for using a third-party class library, i.e. add a reference to `NeuralNetwork.dll` to their project's references. As the whole class library is contained in the *NeuralNetwork* namespace, a `using NeuralNetwork` directive (or any other nested namespace) is required.

In this section, we will present a tutorial dedicated to building, training and using an multi-layer perceptron to compute a simple function - the *XOR logical function*. The XOR logical function has been chosen for two reasons. First, it is simple enough. Therefore, it will not complicate the explanation unnecessarily. Second, it is *linearly inseparable* and therefore not so simple. Some logical functions (e.g. *NOT*, *AND*, *OR*) are *linearly separable* and therefore so simple, that a single-layer perceptron can be build, trained and used to compute them, making the use of a multi-layer perceptron rather unnecessary.

5.4.1 Step 1 : Building a Training Set

The training set can be build either manually (in the program code itself) or automatically (from a text file).

Step 1 : Alternative A : Building a Training Set Manually

When building a training set manually, the input and output vector lengths have to be specified. The length of the input vector is the dimension of the function's domain (2 in our case):

```
int inputVectorLength = 2;
```

The length of the output vector is the dimension of the function's range (1 in our case):

```
int outputVectorLength = 1;
```

The training set can now be built:

```
TrainingSet trainingSet = new TrainingSet(
    inputVectorLength,
    outputVectorLength
);
```

Note that both dimensions have to be positive. Otherwise an exception is thrown.

Naturally, after the training set is build, it is empty, i.e. it does not contain any training patterns just yet. We have to add them manually one by one:

```
TrainingPattern trainingPattern;
trainingPattern = new TrainingPattern(
    new double[ 2 ] { 0.0,  0.0 }, new double[ 1 ] { 0.0 }
);
trainingSet.Add( trainingPattern );
trainingPattern = new TrainingPattern(
    new double[ 2 ] { 0.0,  1.0 }, new double[ 1 ] { 1.0 }
);
trainingSet.Add( trainingPattern );
trainingPattern = new TrainingPattern(
    new double[ 2 ] { 1.0,  0.0 }, new double[ 1 ] { 1.0 }
);
trainingSet.Add( trainingPattern );
trainingPattern = new TrainingPattern(
    new double[ 2 ] { 1.0,  1.0 }, new double[ 1 ] { 0.0 }
);
trainingSet.Add( trainingPattern );}
```

Note that when attempting to add a training pattern to a training set, its dimensions have to correspond to that of the training set. Otherwise an exception is thrown informing the client about their incompatibility.

The training set is now build and ready to be used.

Step 1 : Alternative B : Building a Training Set Automatically

When building a training set automatically, the name of the file (and path to it) containing the training set has to be specified.

```
TrainingSet trainingSet = new TrainingSet( "XOR.trainingset" );
```

Note that the contents of the file have to conform to the `.trainingset` file format precisely. Otherwise an exception is thrown.

The trainingset file format describes the way a training session is stored in a file. The firs row contains two numbers, the dimensions of the input and output vectors. Exactly one blank line has to follow. Next, the file contains an arbitrary number of training patterns, each stored on a separate line. The training pattern is stored by simply having its input and output vectors concatenated and the resulting sequence of real numbers written on a single line[2].

The training set is now build and ready to be used.

5.4.2 Step 2 : Building a Blueprint of a Network

Before we can build a network, we need a blueprint to control the construction process. A network blueprint has to contain an input layer blueprint, an arbitrary number of hidden layer blueprints (depending on the required number of hidden layers) and an output layer blueprint.

The input layer blueprint requires that the client specifies the number of input neurons, i.e. the length of the input vector:

[2]See the example trainingset files included.

```
LayerBlueprint inputLayerBlueprint = new LayerBlueprint(
    inputVectorLength
);
```

The hidden layers blueprints (being activation layer blueprints) each require that the client specifies the number of hidden neurons and the layer activation function:

```
ActivationLayerBlueprint hiddenLayerBlueprint =
    new ActivationLayerBlueprint(
        2, new LogisticActivationFunction()
    );
```

Apart from the most widely used activation function - the logistic activation function - several alternative activation functions are provided, namely the linear activation function (used almost exclusively as an output layer activation function) and the hyperbolic tangent activation function.

The output layer blueprint (being an activation layer blueprint) requires the number of output neurons and the layer activation function to be specified by the user:

```
ActivationLayerBlueprint outputLayerBlueprint =
    new ActivationLayerBlueprint(
        outputVectorLength,
        new LogisticActivationFunction()
    );
```

Note that the number of (input, hidden and output) neurons has to be positive. Otherwise an exception is thrown.

Now the we have constructed all layer blueprints, we can create the network blueprint:

```
NetworkBlueprint networkBlueprint = new NetworkBlueprint(
    inputLayerBlueprint,
    hiddenLayerBlueprint,
    outputLayerBlueprint
);
```

If more than one hidden layer is required, the library offers an overload of the network blueprint's constructor accepting a whole array of hidden layer blueprints instead of a single hidden layer as its second argument.

The network blueprint is now built and ready to be used.

5.4.3 Step 3 : Building a Network

Given we have already created a network blueprint, we can use it to build[3] the neural network[4]:

```
Network network = new Network( networkBlueprint );
```

The network is now built and ready to be trained.

5.4.4 Step 4 : Building a Teacher

While a teacher requires the training set to be specified, the validation and test sets need not be specified. However, shall the user decide to supply the teacher will validation and/or test sets as well, they have to be of the same type as the training set, i.e. their input vector lengths and output vector lengths have to match.

Currently, the teacher will use its training set to both train any given network and to evaluate how successful the network was at classifying the *within-sample* patterns, i.e. inferring the mapping from the data. Its test set, not presented during the training, is then used to evaluate how successful the network was at classifying the *out-of-sample* patterns, i.e. how well it can *generalise* the mapping. The teacher currently has no use for its validation set.

In our case, the teacher is provided only with a training set, as there is no point in generalising such a simple (and enumerable) mapping as the XOR logical function.

[3]Note that there is (as of now) no alternative way to build a network - the client has to use a blueprint. Since the network construction process is a complex one and offers many possibilities to go astray, it has been (for the time being) automated as much as possible.

[4]In fact, we can use it to build as many neural networks of the same architecture as requested.

```
TrainingSet validationSet = null;
TrainingSet testSet = null;
ITeacher backpropagationTeacher = new BackpropagationTeacher(
    trainingSet, validationSet, testSet
);
```

Apart from the most widely used teacher, the Backpropagation teacher (BackpropaagtionTeacher), several alternative teachers are supported. These include the genetic algorithm teacher (GeneticAlgorithmTeacher), the simulated annealing teacher (SimulatedAnnealingTeacher) and the ant colony optimization teacher (AntColonyOptimizationTeacher).

The teacher is now built and ready to be used.

5.4.5 Step 5 : Training the network

When training a neural network, it is possible to specify some requirements, e.g. that a allotted computational budget (the maximum number of iterations) is not exceeded or that a required level of accuracy (the maximum tolerable network error) is achieved. Depending on which requirements are specified, the network can be trained in three different modes:

Mode 1

Training in the first mode means a certain computational budget is allotted (the maximum number of iterations) and a certain level of accuracy (the maximum tolerable error of the network) is required. Under these circumstances, the training ends either when the budget is depleted or the accuracy is achieved.

```
int maxIterationCount = 10000;
double maxTolerableNetworkError = 1e-3;
TrainingLog trainingLog = backpropagationTeacher.Train(
    network, maxIterationCount,  maxTolerableNetworkError
);
```

The actual number of iterations used and the actual error of the network achieved might be of interest. These can be accessed via the TrainingLog's `IterationCount` and `NetworkError` properties respectively.

The network is now trained and ready to be used.

Mode 2

Training in the second mode means only a certain computational budget (the maximum number of iterations) is allotted. The level of accuracy is "absolute" (the maximum tolerable error of the network is 0). Under these circumstances, as the level of accuracy can hardly be achieved, the training ends when the computational budget is depleted.

```
int maxIterationCount = 10000;
TrainingLog trainingLog = backpropagationTeacher.Train(
    network, maxIterationCount
);
```

While the actual number of iterations used is of no interest, the actual error of the network achieved might be. It can accessed via the TrainingLog's `NetworkError` property.

The network is now trained and ready to be used.

Mode 3

Training in the third mode means only a certain level of accuracy (the maximum tolerable network error) is required. The computational budget is "absolute" (the maximum number of iterations is set to `Int32.MaxValue`). Under these circumstances, as the computational budget can hardly be depleted, the training ends when the accuracy is achieved.

```
double maxTolerableNetworkError = 1e-3;
TrainingLog trainingLog = backpropagationTeacher.Train(
    network, maxTolerableNetworkError
);
```

While the actual error of the network achieved is of no interest, the actual number of iterations used might be. It can be accessed via the TrainingLog's `IterationCount` property.

The network is now trained and ready to be used.

5.4.6 Step 6 : Examining the Training Log

The training log returned by the teacher's `Train` method contains useful information concerning that particular training session. More precisely, it conveys information of three types:

- the information regarding the training session itself: the number of runs used (`RunCount`), the number of iterations used (`IterationCount`) and the network error achieved (`NetworkError`),
- the measures of fit: the within-sample residual sum of squares (`RSS_TrainingSet`), the within-sample residual standard deviation (`RSD_TrainingSet`), the Akaike information criterion (`AIC`), the bias-corrected Akaike information criterion (`AICC`), the Bayesian information criterion (`BIC`) and the Schwarz Bayesian criterion (`SBC`), and
- the information related to the networks ability to generalise: the out-of-sample residual sum of squares (`RSS_TestSet`) and the out-of-sample residual standard deviation (`RSD_TestSet`).

5.4.7 Step 7 : Using the Trained Network

After a network has been trained, the client can use it to evaluate the output vector for an input vector[5]:

```
double[] inputVector = trainingSet[ 0 ].InputVector;
double[] outputVector = network.Evaluate( inputVector );
```

[5] Obviously, this vector does not have to be one of those presented to the network during the training. In our simple example though, we have no such vector at our disposal.

Chapter 6

MarketForecaster (Console Application)

6.1 Requirements Specification

MarketForecaster is a console application for forecasting markets using simple time series prediction. It is designed to be as lightweight as possible, delegating all the 'grunt work' to the underlying **NeuralNetwork** class library.

6.2 Technical Documentation

MarketForecaster is a console application written in C# programming language (version 3) and targeted for Microsoft .NET Framework (version 3.5). It was developed in Microsoft Visual C# 2008 Express Edition as a part of the **MarketForecaster** solution - an entire solution aimed at forecasting markets using artificial neural networks.

6.2.1 Dependencies

MarketForecaster console application uses the **NeuralNetwork** class library. This has been designed and developed specifically to support the **MarketForecaster** console application and is a part of the **MarketForecaster** solution. **MarketForecaster** console application does not require

any third-party class libraries.

6.3 End User Documentation

MarketForecaster console application is launched from command line takes (exactly) three arguments:

```
MarketForecaster.exe
    airline_data_scaled_0,01.timeseries
    airline_data.forecastingsession
    airline_data_scaled_0,01.forecastinglog
```

6.3.1 Argument 1 : The Time Series File Name

The first argument we need to provide is the name of the file from which the time series will be loaded. This file has to conform to the `timeseries` file format.

The `timeseries` file format describes the way a time series is stored in a file. The most important characteristic is that a single `timeseries` file can not store more than one time series and that a single time series can not be stored in more than one file. Thus, a `timeseries` file always contains exactly one time series. A `timeseries` file is allowed to contain only real numbers separated by *exactly one* space character. Whether they are all stored in a single line or each on a separate line, or whether the file contains blank lines is irrelevant. This leniency in `timeseries` file format specification allows for an arbitrary tabulation of contained data, e.g. to be easily interpreted by a human reader. Taking our `airline_data.timeseries` file as an example, it contains 12 rows (representing years from 1949 to 1960) each consisting of 12 real values (each representing the monthly total of international airline passengers).

Please note that once a time series is loaded, no preprocessing of the data takes place. Any preprocessing should therefore be carried out before the time series is loaded. Also notice that, since the time series preprocessing can easily be handled by simple scripts, we have decided that a similar functionality in the **MarketForecaster** would have been rather redundant.

6.3.2 Argument 2 : The Forecasting Session File Name

The second argument we need to provide is the name of the file from which the forecasting session will be loaded. This file has to conform to the `forecastingsession` file format.

The `forecastingsession` file format describes the way a forecasting session is stored in a file. The most important aspect is that a single `forecastingsession` file can store an arbitrary amount of so-called *(forecasting) trials*. Each trial is stored on a separate line, so the `forecastingsession` file is, in effect, a row-oriented batch file. Here is how a single row from `forecastingsession` file looks like:

```
12; -4, -3, -2, -1; 0; 1
```

We will refer to this row when describing the format of a trial.

The motivation for grouping trials into a forecasting session is twofold. First, it *automates* the process of launching multiple trials. Second, as there is one-to-one relation between a forecasting session and a forecasting log (described in the following subsection), the results of all trials stored in one forecasting session are logged into a single file. Being stored in a single file, their details and achievements can be compared and contrasted with ease.

The *trial* format consists of four fields separated by semicolon:

1. the first field specifies the actual number of forecast, i.e. the size of the test set,
2. the second field lists the lags,
3. the third field lists the leaps, and
4. the fourth field specifies the number of hidden neurons of a multi-layer perceptron (containing one hidden layer) used in the trial.

It is not necessary for the lags and leaps to be ordered in ascending fashion, although following this convention is recommended as it will contribute to the clarity of the `trainingsession` file.

Currently, only one-step-ahead forecasts are supported, i.e. at any given moment in time, only the immediate next time series element can be predicted. See discussion in *Choosing the Leaps* section of *Applying Artificial Neural Networks to Market Forecasting* chapter for a way to overcome this limitation.

Having familiarized ourselves with the trial format, the example above can be interpreted as follows. Forecast the last 12 months of the time series (the last year) taking the values at lags 4, 3, 2 and 1 as the input values and the value at leap 0 (the immediate future) as the output value (one-step-ahead forecasts). Use a multi-layer perceptron with 1 neuron comprising the hidden layer.

Typically, all trials constituting a forecasting session will attempt the same number of identically leaped forecasts (their first and third fields' values will match). However, to achieve this, they will usually attempt to use differently lagged variables (second column) and different numbers of hidden neurons (fourth column) in the multi-layer perceptron.

6.3.3 Argument 3 : The Forecasting Log File Name

The third argument we need to provide is the name of the file into which the forecasting log will be saved. The training log will be created if the file does not already exist or overwritten otherwise. The forecasting log produced by the **MarketForecaster** console application, once created, conforms to the `forecastinglog` file format.

The `forecastinglog` file format describes the way a forecasting log is stored in a file. The most important characteristic is that by the time the execution of the forecasting session is over, the forecasting log will have contained exactly the same number of entries as is the number of trials comprising the forecasting session. As the `forecastingsession` file format is row-oriented (with each row representing one trial), the `forecastinglog` file format is row-oriented as well (with each row representing one entry). Naturally, the n-th row of the `forecastinglog` file (excluding the header) contains an entry regarding the trial contained in the n-th row of the `forecastingsession` file. Here is how a single row from the `forecastinglog`

file looks like[1]:

```
-4, -3, -2, -1; 128; 1; 7;
    10,4416889220462; 0,285614591194998;
    -306,79666181207; -279,832449964632;
    3,31320667739403; 0,525452715712368}
```

We will refer to this row when describing the format of an entry.

The *entry* format consists of 10 fields separated by semicolon:

1. the first field lists the lags of the trial (-4, -3, -2, -1),
2. the second field holds the number of effective observations presented to the multi-layer perceptron during the training session, i.e. the size of the training set (128),
3. the third field holds the number of hidden neurons of the multi-layer perceptron (1),
4. the fourth field holds the number of parameters, i.e. the number of synapses of the the multi-layer perceptron (7),
5. the fifth field holds the residual sum of squares for the within-sample observations (10,4416889220462)
6. the sixth field holds the residual standard deviation for the within-sample observations (0,285614591194998),
7. the seventh field holds the Akaike information criterion (-306,79666181207),
8. the eighth field holds the Bayesian information criterion (-279,832449964632),
9. the ninth field holds the residual sum of squares for the out-of-sample observations (3,31320667739403),

[1]Please take into consideration that due to the length of the entry being greater than the width of this page, it has been typeset into three lines. See the actual example of a `traininglog` file included on the CD-ROM.

10. the tenth field folds the residual standard deviation for the the out-of-sample observations (0,525452715712368).

Chapter 7

Applying Artificial Neural Networks to Market Forecasting

When the idea of applying neural networks to market forecasting (economic time series prediction) first emerged, it was widely believed that neural networks might actually be the 'ultimate solution' for this task. Being universal function approximators, capable of approximating any non-linear function, many were hoping they could be employed to infer the highly complex and non-linear relationships among various economic processes. Their black box nature considerably lowered the knowledge requirements one had to met to be able to operate them. These, combined with their relatively high level of fault-tolerance and error-robustness, enticed further research into the field.

However, it soon became apparent that it was unlikely that the process of economic forecasting and modelling will be fully automated and the experienced and skilled statisticians rendered redundant. Applying the artificial neural networks to market forecasting remained a relatively complex task that still involves many important decisions and numerous chances to go astray. Admittedly, much of it still bears the characteristic of a *trial-and-error* process.

In the subsections of this chapter, the most important issues involved in successfully applying neural networks to market forecasting (economic time series prediction) are discussed. These issues are presented according to the order in which they naturally arise during their application. Every issue is discussed both from a general point of view and this thesis' perspective.

7.1 Step 1: Validating the Data

Before we actually begin to forecast the time series, we have to undertake two preliminary steps. First we need to *validate* the data, i.e. check it for range under and/or overflows, logical inconsistencies and ensure that no observations are missing. It is advisable not to skip this step even when the data has been obtained from a reliable vendor. The most common issue a forecasting practitioner is likely to come across is the issue of missing observations. Take the Dow Jones Industrial Average, the most famous market index, as an example [3]. The missing observations (Saturdays and Sundays) are due to the stock market being closed on weekends. One safe way to handle the missing observations is to assume the 'smoothness' of the time series and express them as averages of nearby observations. Although more sophisticated interpolating techniques could be used, a simple arithmetic mean will suffice.

The time series we are attempting to forecast in this thesis - the international airline passengers data - has been validated by ensuring it contains all 144 observations (12 years of monthly data).

7.2 Step 1: Preprocessing the Data

Preprocessing the time series data in a clever way can relieve the stress placed on any forecasting method and improve its the predictive accuracy. However, an elaborate data pre-processing will not be discussed because we wanted to assess the neural networks' performance when no 'help' (e.g. in the form of *feature extraction*) is available.Thus, an absolutely essential data transformation – *scaling* – and two other, most commonly used ones – (*first differencing* and *logging*) are presented in this section. Furthermore, the data preprocessing is an active research field on its own, and exploring its findings beyond these simple techniques point lies outside the frame of this work.

7.2.1 Scaling

Even if we decide not to pre-process the raw data in any way, we have to *scale* them so that they all lie within the bounds of the output layer activation function, that is between 0 and 1 for the logistic function and between -1

and 1 for the hyperbolic tangent function. As a linear function's range is not bounded, no scaling is expected to be necessary when using the identity activation function.

In our forecasting effort, we have conducted three experiments. The logistic function was chosen as the hidden layer's activation function in all of them. However, they differed in the choice of *scaling factors* and output layer activation functions.

1. In the *first experiment*, raw data (scaling factor equal to 1) was forecast using the identity output layer activation function.

2. In the *second experiment*, the exact same architecture was given a chance to adapt to scaled data (scaling factor equal to 0.01).

3. In the final, *third experiment* we conducted, the identity function was replaced by the logistic function and the data was scaled to fit into its range (scaling factor equal to 0.001).

7.2.2 First Differencing and Logging

First differencing (i.e. taking the changes in the dependent variable instead of its absolute value) can be used to remove a linear trend from the data [6]. *Logging* the data (i.e. taking the natural logarithms) is useful for time series consisting of both large and small values. Furthermore it converts any multiplicative relationships to additive (e.g. the multiplicative seasonality to additive). It has been suggested in [8] that this simplifies and improves network training. However, taking into consideration that the neural networks are able to cope both with linearity and non-linearity, we have instead decided to focus our attention and effort on the actual design of these neural networks rather than concocting sophisticated data transformation schemes.

7.3 Step 2: Choosing the Leaps

The *leaps* describe the future values of the time series we wish to forecast. More formally, the leaps is a vector $[i_1, i_2, \ldots i_n]$ where n is the number of future values we want to predict at once and i_j specifies the offset of the j-th future value, i.e. how far the future value is. As an example,

consider the leaps [0]. This means, we only want to predict the immediate future value (*one-step-ahead forecast*). Compare this with the leaps $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$ predicting the next consecutive 12 values at once (*multi-step-ahead* forecast). We might not be interested in the immediate future value at all, and instead forecast the year-ahead value by specifying the following leaps [12].

Currently, only one-step-ahead forecasts are supported, i.e. at any given moment in time, only one future value of a time series can be predicted. There are compelling reasons to always use only one output neuron. It has been observed by Masters [8] that neural networks with multiple outputs, especially if these outputs are widely spaced, will produce inferior results as compared to a network with a single output.

However, should the multi-step-ahead forecast be required, there exists an intuitive way to simulate it using one-step-ahead forecasts. We just need to run one-step-ahead forecast for each step we want to obtain, and feed the output back into the network between each two consecutive runs, i.e. replace the lag 1 input of the later run with the output of the earlier run. Naturally, this workaround is not by any means a full substitute for the multi-step-ahead forecasting, as we are are using data already flawed with predictive inaccuracies to predict further data, thus cumulating the forecasting error.

Another solution, suggested by Kaastra and Boyd [6], is to maintain a specialised separate neural network for each individual leap.

7.4 Step 4: Choosing the Lags

The *lags* describe the past values of the time series we with to forecast. More formally, the lags is a vector $[i_1, i_2, \ldots i_n]$ where n is the number of past values we want to base our prediction on and i_j specifies the offset of the j-th past value, i.e. how far the past value is. As an example, consider the lags $[-4, -3, -2, -1]$. This means, we want to base our prediction on the past four values. Compare this with the lags
$[-13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1]$. Here we base our prediction on the past thirteen values (a whole year and an additional month). It is perfectly possible not to use the entire range of values between the maximum and minimum lag. For instance, we may decide to base our prediction on the past two values and this month's last year value, $[-12, -2, -1]$.

For our forecasting session, we have devised a total of 21 different trials. Each trial is uniquely distinguished by the lags and the number of hidden neurons is prescribes. The following sets of lags were tried:

1. $[-2, -1]$
2. $[-12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1]$
3. $[-14, -13, -12, -2, -1]$
4. $[-13, -12, -2, -1]$
5. $[-13, -12, -1]$
6. $[-12, -2, -1]$
7. $[-12, -1]$

The sets of lags evolved as follows. Let us, for the purpose of this explanation, assume we wanted to forecast the values of the last year (1960) of the *international airline passengers data* based on the previous eleven years (1949 – 1959, both included) [1]. Also, suppose we are about to forecast the number of passengers for January 1960 now.

The **first attempt** was a naïve one. We decided to let the multi-layer perceptron look for the way any month is related to the two months that immediately precede it. After we realised the data displays seasonal variations, the **second attempt** at forecasting the January 1960 was made. It took the whole year 1959 into consideration. Studying the trained network's configuration we realised that lags $-11, -10, \ldots, -3$ (from February 1959 to October 1959) bear little influence on the value being forecast. On the other hand, lags $-14, \ldots, -12$ (November and December 1958, and January 1959) influence the value of January 1960 much more significantly. This key observation is captured in our **third attempt**. The attempts **four**, **five** and **six** and **seven** each seek to narrow the set of dependencies even further, with the **seventh** attempt taking the key idea to the extreme.

[1] As a matter of fact, this is exactly the way we carried out our experiment.

7.5 Step 5: Building the Training, Validation and Testing Sets

The available data (e.g. the whole data set extracted from time series) is usually divided into three distinct sets – the *training*, *validation* and *test* sets. Of these, the *training set* should be the largest and is used as first to train the neural network. The *validation set*, containing observation not seen during the training, is used afterwards to evaluate the networks ability to generalise. After repeating this two-step process for neural networks with different architectures and with different training parameters, the model that performed best on the validation set is selected. This model is trained yet again, this time using both training and testing sets. The final model is accessed using the *testing set*, which should consist of the most recent contiguous observations [6]. Note that after assessing the final model with the test set, you must not tune it any further.

The observations contained in the training and validation sets are called *within-sample*, as they are used, in one way or the other, to optimise the configuration of a neural network. On the other hand, the observations contained in the test set are called *out-of-sample*, since the neural network is not allowed to adapt further as soon as it sees them.

In our forecasting experiment, we have decided not to use the validation set, but instead choose the best model according to various statistics. Their choice is discussed in section *Step 7: Selecting Criteria for Picking the Best Model*. Concerning the test set composition, we elected to follow the recommendations, and picked the last twelve observations.

7.6 Step 6: Determining Multi-layer Perceptron's Architecture

While the number of input neurons and output neurons is dictated by the lags and leaps respectively, the number of hidden layers and the number of hidden neurons comprising them remain open to experimentation.

7.6.1 Deciding on the Number of Hidden Layers

A neural network can, in theory, contain an arbitrary number of hidden layers. In practice however, networks with more than two hidden layers are seldom used. There are strong reasons not increase their number beyond this point. First, there is no need to do so, as theoretically, a network with one hidden layer is capable of approximating any continuous function, provided the hidden layer contains a sufficient number of hidden neurons. Practical applications seem to agree with theory, as the introduction of additional hidden layers could not be sufficiently justified. On the contrary, a greater number of them increase the computational time.

However, the most serious is the danger of *overfitting*. Overfitting happens when the number of synapses of a network is relatively high with respect to the size of the training set. Under such circumstances, the network is able to learn and reproduce every single training pattern individually, and there is no pressure on it to generalise. Overfitting is characterised by excellent within-sample measures of fit on one hand, and disappointing out-of-sample forecast accuracy on the other.

Taking into consideration the length of the international airline passengers data and some experimentation, we have concluded that in order to eliminate the threat of overfitting, no more than one hidden layer will be used.

7.6.2 Deciding on the Number of Hidden Neurons in Each Hidden Layer

Deciding on the 'ideal' number of hidden neurons is more or less a process of trial-and-error. Obviously, many rules of thumb have been proposed in literature, some of them relating the number of hidden neurons to the number of input neurons, others to the size of the training set. Of all we have come across, the following recommendation stood out as particularly insightful. According to Klimasauskas [7], there should be at least five times as many training facts as weights, which sets an upper limit on the number of input and hidden neurons.

Instead of attempting to implement any particular heuristic, we have decided to try out several possible numbers of hidden neurons with each set of lags. The only (upper) bound for this number was the number of input

neurons determined by any particular set of lags.

Note that in addition to the so-called *fixed* approach where either a validation set or the human inspection determines the 'best' number of hidden neurons, more advanced techniques exist. These are based on incremental *addition* or *removal* of hidden neurons during the training process itself.

Regardless of the method used to select the range of hidden neurons to be tested, the rule is to always select the network that performs best on the testing set with the least number of hidden neurons [6]

7.6.3 Deciding on the Activation Functions for Each Non-input Layer

Generally speaking, when trying to forecast markets, non-linear activation functions are preferred to linear activation functions. Two commonly used activation functions are the *standard sigmoid function*, or *logistic function* and the *hyperbolic tangent function*. A *linear function* (e.g. the *identity function*) can also be used, but only in the output layer. Depending on the choice of the output layer's activation function, the data may require scaling so that it is consistent with the range of this function.

Both logistic and linear activation functions have been tried as the output layer activation functions in our experiment. Although we expected the logistic function to perform much better, we have arrived to a conclusion, that provided the data is suitably scaled, they are practically interchangeable.

7.7 Step 7: Training the Multi-layer Perceptron

This is the point where the features of the underlying **NeuralNetwork** class library are fully appreciated. There is no need to specify the parameters of the Backpropagation learning algorithm (namely the learning rate and the momentum), as these will be maintained by the algorithm itself. The teacher also minimises the chance of getting stuck in a local minimum by restarting itself a number of times and running the entire adaptation process using a (generally) different set of initial synaptic weights. Only the computational budget (the maximum number of runs and the maximum number of

iterations in a run) and the required level of accuracy has to be specified.

To train the multi-layer perceptrons in our experiment, we have found that the maximum number of runs need not be higher than 10, as this already accounts for a fairly satisfactory chance the training has not been trapped in a local minimum in each of these rounds. We also limited the maximum number of iteration to undertake during each run by 10000. It has been observed that such a generous computational budget can accommodate for a satisfying decline of network error. To force the algorithm into the exhaustion of the entire computational budget, we decided to require the absolute level of accuracy[2].

7.8 Step 8: Selecting Criteria for Picking the Best Model

Selecting criteria for picking the best model out of many that have undergone training is a tricky task, partially because so many have been suggested. Yet again, there is no such thing as the universal criterion that would be applicable in every possible situation. The closest one can hope to get is the *residual sum of squares (RSS)*:

$$RSS = \sum_{i=1}^{n} e_i^2,$$

where $e_i = y_i - f(x_i)$. As this depends on the number of training patterns presented (which is generally different for every trial in a forecasting session), it is difficult to compare the suitability of differently lagged models with each other. A normalised version of RSS, the *residual standard deviation (RSD)* removes this obstacle:

$$RSD = \sqrt{\frac{RSS}{n}}$$

where n is the number of observations presented to the network during the training session.

Notice that since these statistics can only become smaller as the model becomes larger, they have a tendency to favour colossal models with a multitude of synapses. However, as we've already seen, the larger the model,

[2]Naturally, this level of accuracy could hardly ever be met, no matter the extent of the computational budget.

the higher the more imposing the danger of overfitting is. Thus, a criterion that would take the size of the neural network model into consideration was sought.

7.8.1 Information Criteria

Fortunately, a whole group of evaluation criteria that address the issue of penalising the gigantic neural network models exists – the so-called *information criteria*:

- *Akaike information criterion* ($AIC = n \log \frac{RSS}{n} + 2p$),
- *bias-corrected Akaike information criterion* ($AIC_C = AIC + \frac{2(p+1)(p+2)}{(n-p-2)}$),
- *Bayesian information criterion* ($BIC = n \log \frac{RSS}{n} + p + p \log n$), and
- *Schwarz Bayesian criterion* ($SBC = n \log \frac{RSS}{n} + p \log n$).

Note that for all of these criteria, the lower the value, the better the forecasting model. Also note that while the Akaike information criterion (AIC) punishes the extra parameters rather lightly, its bias-corrected version (AICC) and the Bayesian information criterion (BIC) do so much more severely. All of these are offered by the **NeuralNetwork** class library.

7.9 Results

In this section we present the original results of our experimentation. We will be referring to the three experiments as described in the *Step 1: Preprocessing the Data* section. The results of these experiments (forecasting sessions) are stored in three forecasting logs: `airline_data_raw.forecastinglog` stores the results of the firs experiment,
`airline_data_scaled_0.01.traininglog` contains the results of the second experiment and finally `airline_data_scaled_0.001.traininglog` stores the results of the third experiment. In the following subsections, we will take a closer look at the results contained in these forecasting logs.

7.9.1 First Experiment

In the first experiment (refer to the `airline_data_raw.forecastinglog`), the multi-layer perceptrons were fed *raw untransformed data*. This necessitated the use of a *linear activation function* in the output layer. The results yielded by this experiment are downright disappointing. When forecasting time series whose values are expected to fall somewhere between 0 and 500, the out-of-sample RSD of approximately 211 (achieved by the best models) is utterly unacceptable. It looks like the multi-layer perceptrons got overwhelmed by the high outputs of the network to the point where no effective adaptation could take place. In this experiment, we have proven that at least some pre-processing of the data has to be carried out before it is presented to a multi-layer perceptron.

7.9.2 Second Experiment

After it became apparent the multi-layer perceptrons in the first experiment could not deal with raw data, we decided to scale the data by the factor of 0.01. Since their range was still not compatible with the logistic activation function, we had no other choice but to stick with the identity output layer activation function. Admittedly, we did not expect any noticeable improvement. In the end, we were pleasantly surprised just how much this simple data transformation simplified the perceptrons' job. This time, as the data was scaled by a factor of 0.01, we were expecting the forecast values to fall between 0 and 5. The most successful model achieved an out-of-sample RSD of 0.166. Scaling this value back, we obtain the best out-of-sample RSD of 16.6 with respect to the raw data. That is almost 13 times better that the first, naïve experiment. This is a very satisfying result, especially considering it was delivered by one of the smallest model, using only two inputs.

The explanation lies in the importance of these inputs. The first input the last year's value for the same month (lag -12) and the second input is the most recent month (lag -1). Surely, these are the most helpful month to look at when dealing with economic time series containing trend and seasonality.

However, the most important question we posed was whether the information criteria we were looking at, could point us in the right direction. In other words, whether they can, by looking at the models' performance on the within-sample (training set) observations and taking their size into

consideration, identify the best models that would perform the best on the out-of-sample (test set) observations. The idea these criteria seem to be exploiting is the *penalisation* of high numbers of parameters (weights). They assume such colossal models are bound to be over-fitted and will inevitably perform poorly on the out-of-sample observations.

In the second experiment (refer to the `airline_data_scaled_0.01.forecastinglog`), the AIC picked the "-13, -12, -1; 2" model (AIC value of -527) and the BIC picked the model "-13, -12, -1; 1" (BIC value of -502). Notice that these two criteria picked very similar candidates, differing only in one hidden neuron. However, after the forecasting accuracies of all models was measured (by presenting them with out-of-sample test set observations), we found out that both the AIC-favourite (scoring 0.219) and the BIC-favourite (scoring 0.244) were outperformed. Several other models achieved lower out-of-sample RSDs, most notably by the "-12, -1; 2" model, which achieved the value of 0.166. Therefore in this particular forecasting session, the information criteria, despite choosing relatively acceptable models, failed to identify the best ones.

7.9.3 Third Experiment

In the last experiment (refer to the `airline_data_scaled_0.001.traininglog`), we have decided to try out the logistic function as the output layer activation function. For this, we had to scale the data so that they all fit into the interval $[0, 1]$. Hence a scaling coefficient of 0.001 was chosen

During this forecasting session, the lowest AIC value was achieved by the model "-13, -12, -1; 2" (AIC value of -1072), the very same model as in the previous experiment. The BIC too repeated its choice from the second experiment, and again picked the "-13, -12, -1; 1" model (BIC value of -1043). These two favourites achieved the out-of-sample RSDs of 0,023 and 0.026 respectively. In comparison with other models' achievements, these are again not very convincing results. This time, the "-14, -13, -12, -2, -1; 4" model outperformed all with the out-of-sample RSD of 0.020.

A point worth noting is that in this forecasting scenario, the difference in forecast accuracies between the models picked by the information criteria and those that eventually performed the best was less striking.

A question we were asking was, whether the information criteria can iden-

tify the models that would eventually perform the best when their forecast accuracies are measured. Unfortunately, we have to conclude that although their respective choices were not totally misguided (both the ACI-favourite and the BIC-favourite performed moderately well), neither the Akaike information criterion nor the Bayesian information criterion could be entrusted with the task of choosing the best forecasting model.

A possible solution, not further investigated in this work, is to assemble a 'panel of experts' (a set of information criteria with 'weighted' opinions on the matter) whose overall predictive potential will hopefully be greater than that of any of its constituting members.

Chapter 8

Conclusion

The aim of this work was to investigate the forecasting capabilities of artificial neural networks in general, as well as provide some concrete guidelines that can be followed to effectively utilise this predictive potential. We have divided our path to achieving this goal into two stages.

The purpose of the first stage was to lay solid theoretical and practical (software) foundations for any work that would follow. Its ultimate objective was to design an implement a robust, yet easy-to-use artificial neural network class library. This objective has been successfully achieved through the **NeuralNetwork** class library. This library, although originally conceived as nothing more than an auxiliary project, evolved into an extensible general-purpose artificial neural network class library. However, as of now, only one type of neural network is supported – the multi-layer perceptron. Instead of supporting more types of neural network architectures, we elected to concentrate our effort on supporting as many learning algorithms as possible, so that a potential client can experiment with training the same network in a variety of ways.

Despite being satisfied and confident with the final result of our achievement, we see room for potential improvement. The following augmentations did not make it to the first 'release', either because their addition has not been justified yet, or because their full support would exceed the allotted time budget (and partial support would satisfy only the least demanding clients):

- support for neuron-unique gains of logistic activation function,

- wider range of possibilities for adapting learning rate during the training,
- support for constructing more sophisticated training strategies (e.g. allowing more control over the way the training patterns are presented to the network being trained),
- support for constructing hybrid training algorithms,
- support for adding or removing the hidden neurons and/or layers 'on the run' (i.e. constructive and/or destructive learning algorithms),
- serialisation support, and
- multi-threading support.

The **NeuralNetwork** class library can easily be considered the main contribution of this work.

In the second stage, the actual experimentation was carried out. To automate the process of launching a batch of forecasting trials, we decided to create a lightweight **MarketForecaster** console application on top of the **NeuralNetwork** class library. With its assistance, we were able to group a number of forecasting trials into a forecasting session, execute this session, and store the results into a forecasting log.

Our forecasting session attempted to try out every sensible combination of lagged variables as possible. For every such combination, several different numbers of hidden neurons were tested.

Three particular launches of this forecasting session were presented in this thesis. During the *first* session, we used multi-layer perceptrons with the identity function as the output layer activation function and fed them with raw data. The results suggest the neural networks were not able to cope with untransformed data.

For our *second* session, we attempted to scale the data by a factor of 0.01, still using the identity activation function in the output layer. During this session, almost all the non-naïve trials yielded results of reasonable accuracy. However, neither information criterion was able to pick the candidate that later performed the best on the out-of-sample observations. The candidates picked by the criteria did not perform badly on the test set, but were both outperformed by a comfortable margin.

The *third* session was executed using data scaled by the factor of 0.001. This placed all the elements of the time series within a range compatible with the logistic function, and thus allowed us to use it as an output layer activation function. The forecasting accuracies obtained were somewhat less satisfying than those obtained from the second session. The problem of information criteria being unable to pick the most promising model persisted. Both, the one picked by the Akaike information criterion and the one picked by the Bayesian information criterion[1] were still outperformed, even if not so convincingly, by other models.

[1] As a matter of fact, both criteria repeated their choice from the previous session.

Bibliography

[1] Box, G. E. P., Jenkins G. M. and Reinsel, G. C. (1994) Time Series Analysis, Forecasting and Control, 3rd edn., Eaglewood Cliffs: Prentice Hall.

[2] Dagum, E. B. and Cholette, P. A. (2006) Benchmarking, Temporal Distribution, and Reconciliation Methods for Time Series, *Lecture Notes in Statistics*, Volume **186**, page 15, Springer New York.

[3] YAHOO! Finance, Dow Jones Industrial Average – Historical prices, `http://finance.yahoo.com/q/hp?s=^DJI`.

[4] data & object factory, *Decorator*, `http://www.dofactory.com/Patterns/PatternDecorator.aspx`.

[5] Faraway J. and Chatfield C. (1997) Time Series Forecasting with Neural Networks: A Comparative Study Using the Airline Data, *Applied Statistics*, Volume **47**, Issue 2 (1998), 231-250.

[6] Kaastra, I. and Boyd, M. (1995) Designing a neural network for forecasting financial and economic time series, *Neurocomputing*, Volume **10** (1996) 215-236.

[7] Klimasauskas, C. C. (1993) Applying neural networks, in R. R. Trippi and E. Turban, eds., *Neural Networks in Finance and Investing: Using Artificial Intelligence to Improve Real World Performace* (Probus, Chicago) 64-65.

[8] Masters, T. (1993) Practical Neural Network Recipes in C++, Academic Press, New York.

[9] Socha, K. (2004) ACO for Continuous and Mixed-variable Optimization, *Lecture Notes in Computer Science*, Volume **3172**, 25–36, Springer Berlin / Heidelberg.

[10] Šíma J. a Neruda R. (1996) *Teoretické otázky neuronových sítí*, Matfyzpress, Praha.

[11] Tang Z., de Almeida C. and Fishwick, P. A. (1991) Time series forecasting using neural networks versus Box-Jenkins methodology, *Simulation*, **57**, 303-310.

[12] Vašíček, Z. Simulované žíhání (Simmulated Annealing), Metropolisův algoritmus.

[13] Whitley, D. (1994) A Genetic Algorithm Tutorial, *Statistics and Computing*, Volume **4**, 65–85.

[14] Wikipedia – The Free Encyclopedia, *Artificial neural network*, `http://en.wikipedia.org/wiki/Artificial_neural_network`.

[15] Wikipedia – The Free Encyclopedia, *Feedforward neural network* `http://en.wikipedia.org/wiki/Feedforward_neural_network`.

[16] Wikipedia – The Free Encyclopedia, *Stigmergy*, `http://en.wikipedia.org/wiki/Stigmergy`.

[17] Zhang, G. P. et al. (2004) Neural Networks in Business Forecasting, Idea Group Publishing, London.

Appendix A

CD-ROM Contents

The contents of the attached CD-ROM are organised as follows:

- `MarketForecaster` – *Microsoft Visual C# 2008 Express Edition* solution containing **NeuralNetwork** class library, **MarketForecaster** console application and other projects used in this work,
- `Documentation` – **MarketForecaster** solution documentation generated by Doxygen,
- `Example Data` – training sets, time series, forecasting session and forecasting logs used in this work,
- `Bachelor Thesis` – this work in digital format (DVI, PS and PDF) and its LaTeX source code, and
- `Contact Details.txt` – contact details for the author.