UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

AN INVESTIGATION INTO USING NEURAL NETWORKS FOR STATISTICAL CLASSIFICATION AND REGRESSION

by

EBEN UYS

23009595

Submitted in partial fulfillment of the requirements for the degree

MSc Mathematical Statistics

in the

Faculty of Natural & Agricultural Sciences

at the

University of Pretoria

PROMOTER: DR L. FLETCHER

March 2010

Abstract

Neural networks are seldom used as a modelling tool by statisticians. This is often due to the lack of knowledge in the field of neural networks as neural networks are frequently perceived as mysterious methods that evolved from the field of computer science. In this dissertation an attempt will be made to show that neural network methods are closely related to statistical methods. In particular we will show how a backpropagation neural network can be used for statistical applications like regression and classification which will include the setting up a of neural network for different objectives and also using a neural network for predictive inference. Through simulations we will show an efficient method to fit a neural network in practical applications. A neural network will then be employed in a practical application to illustrate how to use a neural network in a regression or classification context. This application will also show the necessity of statistical knowledge when using a neural network as a modelling tool.

I, Eben Uys declare that the dissertation, which I hereby submit for the degree Master of Mathematical Statistics at the University of Pretoria, is my own work and has not previously been submitted by me for a degree at this or any other tertiary institution.

SIGNATURE: ..... DATE: .....

Contents

Chapter 1

Introduction

When statisticians are presented with a modelling application they usually make use of linear or nonlinear regression methods to approach the problem. Despite the claims being made that neural networks have excellent prediction capabilities, neural networks are very seldom used by statisticians as a modelling tool even if no interpretability of the underlying process is necessary.

Neural networks is a method that was developed in the computer science field of machine learning and although many exaggerated claims were made about neural networks' abilities, they can still be used as a predictive modelling tool with great success. As with any modelling tool, there is a lot of preparation before one can build a model that is effective. This preparation involves specifying the goal of the model, cleaning the data and determining relevant variables to go into the model among other things. The time spent on this preparation phase is often directly related to the performance of the model.

The problem is often that many of the techniques and methods that are used in the preparation phase are of a statistical nature and computer scientists which are the most

active users of neural networks, often lack this statistical knowledge and therefore end up with poor neural network models. On the other end, statisticians often have no knowledge of neural networks and therefore cannot use any of the techniques with great success. This lack of knowledge in neural networks is usually attributed to neural network literature being difficult to understand for people with a statistical background and usually very little theoretical foundation for the neural network methods is given.

In this dissertation an attempt is made to explain neural networks in terms and notation that are familiar to persons with a statistical background. We will draw the analogy between a neural network and a statistical regression model. We will start with statistical methods of nonlinear regression and nonparametric regression and then explain how neural networks also fit into this framework of a regression model. We will also show how neural networks can be used for statistical classification and regression applications.

The layout of this dissertation is as follows: In chapter 2 a literature review will be conducted. This literature review will start with an introduction to nonlinear regression and nonlinear optimisation methods in section 2.1 and then also an introduction to nonparametric regression in section 2.2. This will give the necessary background that is necessary to fully understand how a neural network fits into a regression context and also how neural networks are fitted. In section 2.3 we will have a look at how neural networks evolved from the simple single layer networks to the more complex multilayer neural networks. This will give the reader an idea of exactly what a neural network is. In section 2.4 we will link neural networks with statistical methods of regression and classification and show how these methods are related. In this section we will also show how a neural network can be used for statistical applications with special reference to issues such as overfitting and generalisation.

In chapter 3 we will use a neural network for a business problem. This problem is not a

typical statistical application and therefore we will need to discuss the problem extensively before attempting to apply a neural network. In section 3.2 we will discuss the different neural networks that will be used for regression and classification in more detail. In section 3.3 we will show through simulations how a neural network can be used for regression and classification. In section 3.4 we will discuss the business problem for which the neural network is to be used in more detail. The analysis and results for the neural networks applied to this problem is presented in sections 3.6 and 3.7. We will end this chapter with some of the difficulties experienced with this neural network, future research ideas and some final remarks in sections 3.10, 3.12 and 3.11 respectively.

It should also be mentioned that the amount of literature available which approaches neural network from a statistical point of view is rather limited. The pioneering book of Christopher Bishop by the name of Neural Networks for Pattern Recognition (Bishop, 1995) is by far the best for explaining neural networks from a statistical perspective. Therefore we will use this book intensively in this dissertation and many references will be made to this book.

Chapter 2

Literature Review

2.1 Nonlinear regression

2.1.1 Introduction

This section gives a brief overview of nonlinear regression and also some of the estimation techniques used to fit nonlinear models by least squares. This is deemed necessary as neural networks can be seen as a complex nonlinear regression. When a nonlinear model is fitted and the objective is to minimise the sum of squares, it causes the normal equations to be nonlinear, resulting in equations which cannot be expressed in an explicit form as in the case of the linear model. This causes a problem since the parameters in the model cannot be estimated in a simple way and the sum of squares function is then usually directly minimised by a numerical optimisation algorithm involving complicated iterative calculations (Draper and Smith, 1998).

An in-depth discussion about the statistical properties of least squares estimators will not be conducted here as this is not the main focus of this dissertation but there is a vast amount of literature available for the interested reader, e.g. Gallant (1975b); Jennrich (1969); Malinvaud (1966, 1970).

2.1.2 Nonlinear regression models

A linear regression model fitted to data usually has the form:

$$\begin{aligned} Y &= \beta_0 + \beta_1 Z_1 + \beta_2 Z_2 + \cdots + \beta_p Z_p && \text{where } p \text{ is the number of predictors} \\ &= \sum_{i=0}^{p} \beta_i Z_i && \text{with } Z_0 = 1 \end{aligned}$$

where the Z_i can represent any functions of the predictor variables $X_1, X_2, \ldots, X_p$ (Draper and Smith, 1998). This means that the term linear models can accommodate most of the situations that arise in applications (Gallant, 1975a). From this definition of the linear model we see that a linear model has two meanings: The term linear model implies that there is a straight line relationship between a predictor and a response variable but it also means that the parameters that have to be estimated in the model, occurs linearly in the equation.

A nonlinear function is one which is nonlinear in the parameters for example:

$$y = \alpha + \beta^x$$

This equation is nonlinear in the parameter β and cannot be made linear by a transformation. These types of functions usually arise in instances where the nature of a specific scientific discipline specifies the form of the equation that the data ought to follow (Gal-

lant, 1987). This implies that the mathematical form of the relationship between the response variable and explanatory variables is known and the aim of nonlinear regression is to estimate the unknown parameters in the model (Seber and Wild, 1989). The response function which arises from the solution of a differential equation is one example of a nonlinear regression function (Gallant, 1975a). Another example of a nonlinear model is responses which are periodic in time, with an unknown period, like the following equation:

$$y = \theta_1 + \theta_2 \cos(\theta_4 t) + \theta_3 \sin(\theta_5 t)$$

It can be seen that the parameters θ_i that need to be estimated in this model occur nonlinearly in the equation and that the equation cannot be linearised in any way by transforming one of the predictor variables.

The most common criterion to fit the parameters in a regression model, whether linear or nonlinear, is to fit the parameters by minimising the sum of squares for error. Other criteria can also be used, but it is known that the least squares estimators exhibit preferable properties when certain conditions are met (Ratkowsky, 1983, p.2). In the case of least squares estimates from linear models, if the errors are independent, identically distributed normal random variables with mean zero and variance σ^2, the least square estimates are also the maximum likelihood estimates of the parameters. In addition to this, the least square estimates are unbiased and have minimum variance, indicating that when these assumptions are satisfied, least squares estimators provide the best estimates of the unknown parameters (Ratkowsky, 1983, p.3). This is not the case in a nonlinear regression model where the least squares estimators of the parameters of the nonlinear regression model exhibit these properties only asymptotically while in finite samples the least squares estimators have unknown properties (Ratkowsky, 1983, p.6). This topic will not be pursued any further here, but more information can be found in Ratkowsky (1983, ch.3–6). A good reference for ways to assess the nonlinearity in a regression equation can be found in Ratkowsky (1983, ch.2)

Assume that we have a model which is nonlinear in the parameters $\underline{\boldsymbol{\beta}}$:

$$y = f(\underline{\boldsymbol{x}}, \underline{\boldsymbol{\beta}}) + \varepsilon \tag{2.1}$$

When using least squares to estimate the parameters $\underline{\boldsymbol{\beta}}$, the objective is to estimate $\underline{\widehat{\boldsymbol{\beta}}}$ such that

$$\min \quad SSE(\underline{\widehat{\boldsymbol{\beta}}}) = \sum_{i=1}^{n} (y - f(\underline{\boldsymbol{x}}, \underline{\widehat{\boldsymbol{\beta}}}))^2 \tag{2.2}$$

where n is the number of observations,

$\underline{\widehat{\boldsymbol{\beta}}} = \left(\hat{\beta}_0, \ldots, \hat{\beta}_p\right)'$

and p is the number of predictor variables.

When the regression function is nonlinear, the minimisation of the sum of squares for error is not as simple as in the linear regression case, simply because one cannot find explicit expressions for the parameter estimates in closed form. This means that in order to estimate $\underline{\boldsymbol{\beta}}$ by least squares, one has to use an iterative procedure (Ratkowsky, 1983, p.5). In linear regression, the curve defined by SSE has only one global minimum point, where for nonlinear regression, the surface generally has a number of local minimum points in addition to a global minimum, which may or may not be unique.

2.1.3 Nonlinear estimation techniques

The aim of this section is to give an introduction to those iterative estimation techniques that are most commonly used in nonlinear regression. Instead of providing an in-depth discussion of all these techniques, I will focus on one technique, namely gradient descent, as this method forms the basis for the delta method, which is the most popular method for estimating the weights in a multilayer perceptron neural network. The Newton-Raphson

algorithm will also be discussed as this is a very popular method in estimation of nonlinear regression and many of the other methods follow very similar approaches, meaning that this method serves as an introduction to the other methods.

From (2.2) we see that the objective is to minimise the $SSE(\underline{\boldsymbol{\beta}})$ where $f(\underline{\boldsymbol{x}}, \underline{\boldsymbol{\beta}})$ is a function which is nonlinear in the unknown parameters $\underline{\beta}$. The iterative methods used to minimise the $SSE(\underline{\beta})$ all proceed by assuming initial values for the unknown parameters and then iteratively updating the parameters until the parameters converge to a minimum of the objective function, provided that a minimum in the vicinity of the initial values exists (Kennedy and Gentle, 1980, p.426). In the case of nonlinear regression, the objective function is $SSE(\underline{\boldsymbol{\beta}})$, which may have many local minima and also a possible unique global minimum. There is no guarantee that the procedure chosen will always converge to a global minimum and it may well converge to a local minimum. One has to choose a reliable method that we can expect to converge to a local minimum based on an initial guess about a value of $\underline{\boldsymbol{\beta}}$ which minimises SSE (Kennedy and Gentle, 1980, p.426). There is no best method in this case as every method has its advantages and constraints. Deciding on which method to use is a function of the characteristics of the problem at hand.

Some of the most well known techniques for estimating the unknown parameters in non-linear least squares are gradient descent (steepest descent), conjugate gradient descent, Gauss-Newton, modified Gauss-Newton, Levenberg-Marquatt and Newton-Raphson. The focus here will be on gradient descent as this is typically the same as the delta rule in back propagation neural networks which will be discussed in section 2.3.4. This will provide a good basis for when we investigate the delta rule later and this method also gives a good introduction to numerical optimisation algorithms since it is easy to understand. Comprehensive information on the other methods can be obtained in Thisted (1988, ch.4), Seber and Wild (1989, ch.14), Kennedy and Gentle (1980, ch.10).

Gradient method

The basic idea of gradient descent, also known as steepest descent, is to start at an initial value of the parameter $\underline{\boldsymbol{\beta}}$, denoted by $\underline{\boldsymbol{\beta}}^{(0)}$, and then update $\underline{\boldsymbol{\beta}}$ each iteration by:

$$\underline{\boldsymbol{\beta}}^{(t+1)} = \underline{\boldsymbol{\beta}}^{(t)} + \gamma \underline{\boldsymbol{d}}$$

where γ is the step size and $\underline{\boldsymbol{d}}$ is the direction of steepest descent evaluated at the current point $\underline{\boldsymbol{\beta}}^{(t)}$. It can be shown that the direction of greatest increase in the objective function at a point, say $\underline{\beta}^{(t)}$, is equal to the gradient of the objective function in $\underline{\boldsymbol{\beta}}^{(t)}$ (Fleming, 1965, p.61). This implies that the direction of greatest decrease in the objective function is equal to the negative gradient evaluated at that point.

In a nonlinear regression context, the objective function to minimise is $SSE(\underline{\boldsymbol{\beta}})$. The gradient descent method for nonlinear regression is:

$$\underline{\boldsymbol{\beta}}^{(t+1)} = \underline{\boldsymbol{\beta}}^{(t)} - \gamma \bigtriangledown SSE(\underline{\boldsymbol{\beta}}^{(t)})$$

$$\text{where} \quad \bigtriangledown SSE(\underline{\boldsymbol{\beta}}^{(t)}) = \frac{\partial SSE(\underline{\boldsymbol{\beta}})}{\partial \underline{\boldsymbol{\beta}}}\Big|_{\underline{\boldsymbol{\beta}}=\underline{\boldsymbol{\beta}}^{(t)}}$$

Gradient descent for nonlinear regression is based on the observation that if a real-valued function $SSE(\underline{\boldsymbol{\beta}})$ is defined and differentiable in a neighborhood of a point $\underline{\boldsymbol{\beta}}$, then the SSE will decrease fastest if one goes from $\underline{\boldsymbol{\beta}}$ in the direction of the negative gradient, $SSE(\underline{\boldsymbol{\beta}}^{(t)}) \geq SSE(\underline{\boldsymbol{\beta}}^{(t+1)})$,for $\gamma > 0$, γ a small enough number. This implies that we should proceed in the direction $- \bigtriangledown SSE(\underline{\boldsymbol{\beta}})$. With this in mind one starts at an initial value $SSE(\underline{\boldsymbol{\beta}}^{(0)})$, and then follows the path of steepest descent until a minimum is reached.

We are expecting that

$$SSE(\boldsymbol{\beta}^{(0)}) \geq SSE(\boldsymbol{\beta}^{(1)}) \geq SSE(\boldsymbol{\beta}^{(2)}) \geq \ldots$$

will continue to converge to a desired local minimum. The value of the step size γ is allowed to change at every iteration. Several methods are available to adjust γ at each step. *Line search* is a method which aims to choose the optimum value for γ (Seber and Wild, 1989, p.597). Another popular approach is to choose γ to be a decreasing value at each iteration e.g. halving it at each iteration. Using one of these methods to determine the value for the step size usually ensures better convergence of the algorithm. Note that the (negative) gradient at a point is orthogonal to the contour line going through that point as this direction will give the fastest decrease in $SSE(\underline{\boldsymbol{\beta}})$.

Newton-Raphson method

One of the most popular methods for optimisation in nonlinear least squares is the Newton-Raphson method. This method is known as a second derivative method because the second derivatives of the objective function to be minimised are also computed.

A quick overview of the method follows: Suppose we want to find the p estimated values of $\underline{\boldsymbol{\beta}}$, say $\underline{\hat{\boldsymbol{\beta}}}$, that minimises the objective function

$$SSE(\underline{\boldsymbol{\beta}}) = \sum_{i=1}^{n} (y - f(\mathbf{x}, \underline{\boldsymbol{\beta}}))^2$$

for the n observations.

Let

$$\underline{\boldsymbol{g}} = \frac{\partial SSE(\underline{\boldsymbol{\beta}})}{\partial \underline{\boldsymbol{\beta}}} = \left(\frac{\partial SSE(\underline{\boldsymbol{\beta}})}{\partial \beta_0}, \frac{\partial SSE(\underline{\boldsymbol{\beta}})}{\partial \beta_1}, \cdots, \frac{\partial SSE(\underline{\boldsymbol{\beta}})}{\partial \beta_p} \right)$$

where p is the number of predictor variables.

Let $\boldsymbol{H}$ denote the matrix having entries

$$h_{ij} = \frac{\partial^2 SSE(\underline{\boldsymbol{\beta}})}{\partial\beta_i \partial\beta_j}$$

Let $\underline{\boldsymbol{q}}^{(t)}$ and $\boldsymbol{H}^{(t)}$ be those values evaluated at $\underline{\boldsymbol{\beta}}^{(t)}$. For the Newton-Raphson algorithm, a Taylor series expansion of the second order is used to approximate $SSE(\underline{\boldsymbol{\beta}})$ in the vicinity of $\underline{\boldsymbol{\beta}}^{(0)}$ at the t^{th} iteration of the process.

Let $Q^{(t)}(\underline{\boldsymbol{\beta}})$ be the Taylor series expansion of $SSE(\underline{\boldsymbol{\beta}})$ up to the second order at the point $\underline{\boldsymbol{\beta}}^{(t)}$. Hence

$$Q^{(t)}(\underline{\boldsymbol{\beta}}) = SSE(\underline{\boldsymbol{\beta}}^{(t)}) + q^{(t)}(\underline{\boldsymbol{\beta}} - \underline{\boldsymbol{\beta}}^{(t)}) + \frac{1}{2}(\underline{\boldsymbol{\beta}} - \underline{\boldsymbol{\beta}}^{(t)})H^{(t)}(\underline{\boldsymbol{\beta}} - \underline{\boldsymbol{\beta}}^{(t)})$$

To optimise, which in this case will mean to minimise, the value of $Q^{(t)}(\boldsymbol{\beta})$ at $\underline{\boldsymbol{\beta}}^{(t)}$, we differentiate $Q^{(t)}(\underline{\boldsymbol{\beta}})$ w.r.t. $\underline{\boldsymbol{\beta}}$, set the derivative equal to zero and then solve for $\underline{\boldsymbol{\beta}}$. The value of $\underline{\boldsymbol{\beta}}$ that is thus obtained becomes the next iterative value of $\underline{\boldsymbol{\beta}}$, i.e. $\underline{\boldsymbol{\beta}}^{(t+1)}$.

A single update of the Newton-Raphson algorithm is then:

$$\underline{\boldsymbol{\beta}}^{(t+1)} = \underline{\boldsymbol{\beta}}^{(t)} - \left(\boldsymbol{H}^{(t)}\right)^{-1} \underline{\boldsymbol{q}}^{(t)}$$

Other methods

Even though gradient descent is one of the oldest methods of determining a minimum of a function, it is not used that often anymore as it is notoriously slow to converge. This is especially true when the shape of the objective function resembles long flat valleys,

as this causes gradient descent to have a zigzag trajectory (also called hemstitching) to the bottom of the valley (the minimum of the function). This causes the algorithm to take lots of short steps until it eventually reaches the minimum of the objective function. Conjugate gradient descent is a method that was developed to rectify this by ensuring that there is only one step taken in each search direction in a scaled space. This improves the performance of the algorithm and a minimum of the function is reached in fewer steps. Detailed information on how this algorithm works can be found in Shewchuk (1994).

Because the Newton-Raphson method uses second derivative information in calculating each iteration, more computational power and memory is necessary to calculate each iteration than steepest descent. This method normally displays a quadratic convergence rate near a minimum (Kennedy and Gentle, 1980, p.442). One problem with Newton-Raphson is that the Hessian (second-derivative) matrix may not be positive definite at each iteration. A slight modification to the Newton-Raphson was proposed by Levenberg and Marquardt independently in which a ridge modification is made to the Hessian matrix to ensure that it is positive definite (Levenberg, 1944; Marquardt, 1963) . This leads to the Levenberg-Marquardt algorithm, which is one of the most widely used methods of estimating the parameters in a nonlinear regression. The Hessian matrix may also be difficult to obtain at each iteration or it may become ill-conditioned, causing problems with the estimation of the inverse. With quasi-Newton methods, the Hessian matrix is approximated at each iteration. Some of the methods used to estimate the derivatives are the *secant* method and also the *finite differences* method. For more information on these methods consult Seber and Wild (1989, pp.605–612).

Gauss-Newton is another popular method used in nonlinear regression. The basic outline of this method is that, instead of approximating the objective function, viz.

$$SSE(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y - f(\underline{\mathbf{x}}, \underline{\boldsymbol{\beta}}))^2$$

by a second order Taylor polynomial like in the Newton-Raphson, Gauss-Newton approximates the nonlinear regression function $f(\mathbf{x}, \boldsymbol{\beta})$ itself, by a first order Taylor polynomial. The $SSE(\underline{\boldsymbol{\beta}})$ is minimised similarly to the linear model case. Modifications to this method was proposed by Hartley (1961).

2.2 Nonparametric regression

In traditional nonlinear and linear regression the objective is to fit a model of the form

$$y_i = f(\underline{\beta}, \underline{x}_i) + \varepsilon_i$$

where y_i is the value of the i-th dependent variable, with corresponding predictors variables $\underline{x}_i' = (x_{i1}, \ldots, x_{ip})$ and $\underline{\beta}$ is the parameters of the model to be estimated. The errors are assumed to be independently and normally distributed with constant variance. In linear regression the model is linear in the parameters and in nonlinear regression the parameters enter the model nonlinearly. The function $f(\cdot)$, which specifies the form of the relationship between the dependent variable and the predictors, is known in advance in a well defined analytical form and this reduces the regression problem to estimating the parameters of the model fitted. This can mean that tradition linear and nonlinear regression is very restricted, especially when the functional form between the predictors and the response is very erratic.

A nonparametric regression model removes this limitation by not restricting the functional form between the response and predictor variables. A nonparametric regression model is defined as:

$$\begin{aligned} y_i &= f(\underline{x}_i) + \varepsilon_i \\ &= f(x_{i1}, \ldots, x_{ip}) + \varepsilon_i \end{aligned}$$

where it is assumed that $f(\cdot)$ is any continuous, smooth function. The errors are also assumed to be independently distributed with a zero mean and constant variance.

The objective of nonparametric regression is to estimate the regression function $f(\cdot)$ directly rather than estimating the parameters in the model (Fox, 2002). For some non-

parametric models, the parameters obtained are not unique. This is often one of the major disadvantages of nonparametric models in that the parameters in a nonparametric regression model cannot be easily interpreted like in a linear regression model. This typically limits the use of nonparametric models in cases where interpretation of the model is important.

The function $f(\cdot)$ in the nonparametric regression model can be fitted using least squares and the regression function is usually estimated using smoothers or splines. Smoothers are used as an estimator of the conditional expectation

$$f(x) \approx E(y|x)$$

Smoothing methods in nonparametric regression are used to obtain pointwise estimates of the regression function. The way in which this is most often done is by using a local regression approach in which a point estimate of the regression function is obtained at say x_0, by carrying out a weighted least squares regression. This local regression is usually in the form of a linear or polynomial regression. In this weighted least squares regression the points are weighted by their distance from the focal point x_0 in the sense that points further away from x_0 carry less weight (possibly a weight of 0) than the points closer to x_0. This technique of obtaining a point estimate of the regression function are then repeated at several different points along the range of the inputs. Each of the estimated values are then connected to obtain a nonparametric regression curve.

The weighting of the points around x_0 is most often done by using a kernel function. This kernel function is a symmetric function around zero. The kernel function awards the largest weight to the point at x_0 (or close to X_0), and then the weight decreases symmetrically on each side to 0. The choice of kernel function does not influence the fit of the regression curve significantly (DiNardo and Tobias, 2001). Each kernel function

has a bandwidth parameter, which can be fixed or variable. This bandwidth parameter influences the size of the neighborhood around the point x_0. A larger value for the bandwidth parameter causes more observations to be weighted in the local regression and hence gives the regression model a more global fit. On the other hand, a small value for the bandwidth parameter, causes a more local fit around the point x_0. A fixed value for the bandwidth means that the size of the neighborhood around the focal point x_0 stays constant. A generalisation of this is a variable bandwidth parameter, where an equal amount of observations is included in the local regression around x_0. The reason that this is called a variable bandwidth, is that the size of the neighborhood will be small when there is many observations close to x_0, and this neighborhood size will become bigger as there are fewer observations around x_0.

There is a trade-off between choosing too large a value for the bandwidth parameter and choosing the bandwidth parameter too small. A large value of the bandwidth parameter causes the fit to be oversmoothed and introduces bias in the regression estimates, while choosing a too small value for the bandwidth parameter, causes the regression curve to be very bumpy and increases the variance of the estimate. This is referred to as the model is being overfitted. The interested reader can find more information on kernel regression in Bowman and Azzalini (1997); Fox (2000); Hastie et al. (2001) and Ildiko (1995).

The other popular method of nonparametric regression involves approximating the regression function by using regression splines. The basic idea of regression splines is to find a function, denoted by $\widehat{f(x)}$, with continuous first and second order derivatives, that will minimise the following criterion:

$$SS^*(h) = \sum_{i=1}^{n} (y_i - f(x_i))^2 + h \int_{x_{min}}^{x_{max}} (f''(x))^2 \tag{2.3}$$

This criterion is called the *penalised sum of squares*. The penalised sum of squares is a function of h, where h is a smoothing constant similar to the bandwidth parameter which was discussed earlier (Fox, 2000, p.67).

The first term in the penalised sum of squares is the usual residual sum of squares while the second term that is added to this criterion is called the roughness penalty. This roughness penalty ensures that the regression curve that is fitted to the data, does not just interpolate the data, since if the data is interpolated, the normal residual sum of squares, would be equal to 0, but the regression function will generalise very poorly to unseen data. The penalised sum of squares function tries to avoid this overfitting of the regression function, by penalising the function, if the roughness penalty becomes large. This roughness penalty will become large when the second derivative, over the range of the x's, becomes large. The second derivative gives an indication of the curvature of the function at a selected point, meaning that if the roughness penalty is large, the regression function that was fitted varies rapidly over small regions of the input space. This tends to cause the regression function to fit the training data exactly instead of the underlying distribution in the data.

It can be shown that a natural cubic spline will minimize the penalised sum of squares criterion in (2.3) (Green and Silverman, 1994). The roughness penalty that was added to the penalised sum of squares in (2.3) can take on many different forms and can force the regression function to have a special structure. This forms the basis for generalised additive models of which projection pursuit regression is an example (Hastie et al., 2001, p.34). Projection pursuit regression is of special interest in this dissertation as this is closely related to a multilayer perceptron and will be discussed in the next section.

The above types of nonparametric regression methods are also called penalty methods or regularisation methods. These methods generally restrict the regression coefficients to

avoid overfitting of the regression function to the data. Another method of regularisation will be discussed when fitting a neural network in section 2.3.5. More information on using regression splines can be found in Hastie et al. (2001); Eubank (1988) and Frank (1995).

2.2.1 Projection pursuit regression

Projection pursuit regression is very closely related to neural networks in the sense that it models the response variable as a sum of functions of linear combinations of the predictor variables (Friedman and Stuetzle, 1981). We will see in section 2.3.4 that projection pursuit regression is closely related to a neural network with one hidden layer where the activation functions in the neural network are any arbitrary smooth functions. The situations in which projection pursuit regression is useful is when:

- The number of independent variables is fairly large.
- Many of the independent variables are relevant.
- Most of the predictive information lies in a low-dimensional subspace.

This is also the situation in which it is most appropriate to use a feedforward neural network. One difference between the projection pursuit regression model and feedforward neural networks is the way in which the projection pursuit regression model and its parameters are estimated. The parameters in a neural network are estimated simultaneously while the parameters in the projection pursuit regression model are estimated cyclically in groups. Projection pursuit regression will not be treated indepth in this dissertation but only an overview will be given. The interested reader can consult Bishop (1995,

pp.135-137), Thisted (1988, pp.232-234) , Hastie et al. (2001, pp.347–350) and Friedman and Stuetzle (1981).

Suppose we have p predictor variables, $X_1, \ldots, X_p$, which can be combined in a vector $\boldsymbol{X} = (X_1, \ldots, X_p)'$, and a response variable denoted by Y, the projection pursuit regression model is:

$$Y = \sum_{m=1}^{M} g_m(\boldsymbol{w}_m' \boldsymbol{X})$$

where $\boldsymbol{w}_m$, $m = 1, \ldots, M$ are unit p-vectors of unknown parameters. Since $\boldsymbol{w}_m$ are unit vectors, it follows that $\boldsymbol{w}_m' \boldsymbol{X}$ is the projection of $\boldsymbol{X}$ onto the vector $\boldsymbol{w}_m$. These projections are then transformed by nonlinear functions, g_m, called activation functions. The output Y is formed by taking the sum of these M activation functions. M is the number of functions which is deemed necessary to approximate the regression surface closely. In practice, this value M also needs to be estimated from the data (Thisted, 1988, p.232). To simplify the explanation of how projection pursuit regression work, we will assume that the value of M is known and fixed.

The functions g_m are not known beforehand and these need to be estimated together with the unknown vectors of parameters $\boldsymbol{w}_m$, $m = 1, \ldots, M$ in an iterative way. The measure for how well the projection pursuit regression model fits is usually the sum of squares for error. When the objective is to minimise the sum of squares for error, the iterative estimation method proceeds as follows: The activation functions are estimated by starting with initial estimates for $\boldsymbol{w}_m$, $m = 1, \ldots, M$. This is a one-dimensional curve fitting problem and the functions are usually estimated using smoothing methods like local regression or smoothing splines (Hastie et al., 2001). These estimated functions are then used in the next step where $\boldsymbol{w}_m$, $m = 1, \ldots, M$ are estimated using nonlinear estimation techniques. This process is then repeated, iterating cyclically between estimating the activation functions and the unknown parameters, until the model fits well. This implies that the process will be stopped when the value obtained for the error function is sufficiently small.

An alternative form of the projection pursuit regression model is presented by Bishop (1995, p.136) The model is:

$$y = \sum_{m=1}^{M} b_m g_m(\boldsymbol{w}'_m \boldsymbol{x} + w_{m0}) + b_0$$

where b_m, $m = 1, \ldots, M$ are unknown parameters and b_0 and w_{j0} are constant terms. Using this model emphasizes that constant terms, called biases in the neural network literature, can also be included in the model. Bishop (1995, p.136) states that projection pursuit regression can be seen as a generalisation of a multilayer feedforward neural network, in the sense that the activation functions in the projection pursuit regression model are more flexible. This is because the activation functions are estimated in a nonparametric way in the projection pursuit regression model. A more comprehensive comparison between the multilayer perceptron and projection pursuit regression can be found in Hwang et al. (1994).

2.3 Neural networks

2.3.1 Introduction

Artificial neural networks come from the objective to develop mathematical algorithms that emulate the human brain. These mathematical algorithms are computational modelling tools that can be used to solve a wide array of complex real-world problems. In this chapter, an introduction to neural networks will be given. In section 2.4, it will be shown that neural networks can be expressed i.t.o. a generalisation of statistical models. Since most of the research and development of artificial neural networks come from the Engineering and Computer Science field, a special emphasis will be made to make this understandable to persons coming from a statistical background as most of the notation and terminology for neural networks differ from standard statistical terminology. In the appendix a table is given in which the terminology of neural networks is related to those in the statistical literature.

A neural network is characterised by the following features (Fletcher, 2002):

- The way in which the neurons are connected (architecture).
- The technique which is used to estimate the weights in the network (training method).
- The activation functions.

In this section and in the next section we will discuss different neural networks, starting from a simple feedforward single layer network, to the most popular multilayer network

called the backpropagation network. Different network architectures, estimation methods and activation functions will be introduced later on.

2.3.2 A concise history neural networks

The aim of this section is not to give a full list of the historical development of neural networks but rather point out the most significant contributions to the field. The earliest work done on artificial neural networks can be traced back to the work of McCulloch and Pitts (1943), who introduced simplified neurons. These simplified neurons was a very simplified representation of a biological neuron which could perform computational tasks. This is generally seen as the first work in artificial neural networks and this caused a spark of interest in this field.

The first learning rule for artificial neural networks was created by Donald Hebb (Hebb, 1949). The learning rule can be seen as a way of estimating the weights in a neural network. Researchers like Rosenblatt and Widrow and Hoff started to do serious research in the field of neural networks in the 1950s. Frank Rosenblatt introduced and developed a class of artificial neural networks called *perceptrons* (Rosenblatt, 1958, 1962), which will be discussed in section 2.3.3. Bernard Widrow and Marcian Hoff (Widrow and Hoff, 1960) developed a powerful learning rule for single layer neural networks in the early 1960s. This rule is known as the Widrow-Hoff learning rule, least mean squares or delta rule. This learning rule which was used for single-layer networks is the foundation on which the backpropagation rule for multilayer networks is built (Fausett, 1994, p.23). The ADALINE (ADAptive LInear NEuron or ADAptive LINEear system) was also developed by Widrow and his students. The ADALINE is not a network but rather a single neuron, which, when presented with a pattern of inputs, will produce a single output based on those inputs. This concept is described more in section 2.3.3. In 1969 Minsky and Papert

published their book *Perceptrons* in which they exposed the limitations of these perceptron models. Some of the weaknesses that they mentioned were that perceptrons could only solve simple problems which were linearly separable and also the lack of a general method to estimate the parameters in a multilayer network. This effectively ended the interest in research in this field.

It was only in the 1980s that interest in the field increased again after several important theoretical results were published. These developments included Hopfield's energy approach in 1982 (Hopfield, 1982), estimation algorithms like the backpropagation learning algorithm for multilayer perceptrons, different network architectures like Kohonen networks and the Boltzmann machine which is a nondeterministic neural network. For a full account of all the historical developments in the field of artificial neural networks consult Anderson and Rosenfeld (1988). A brief overview can be found in (Fausett, 1994; Fletcher, 2002).

2.3.3 Single layer neural networks

In this section an introduction to single layer networks will be given. The early single layer networks have their limitations as was proved by Minsky and Papert (1969), but they will still be used in this introduction as they give a clear illustration of the concepts involved in neural networks. This will then serve as a basis to generalise the networks to more complex designs.

Model of the basic neuron

The most simple single layer network is the basic neuron. In the basic neuron, the inputs are ordered in a layer. These inputs are analogous to the independent variables in a statistical context, each input corresponding to one independent variable. Weights are applied to each of these inputs and then summed. This summed value is called the *net input*. The *net input* is compared to some threshold level and the neuron then produces an output based on whether the *net input* exceeds a specified threshold level or not. This model of the neuron was first proposed by McCulloch and Pitts in 1943 (McCulloch and Pitts, 1943). Mathematically we can formulate this as (Beale and Jackson, 1990, pp.41–43):

Suppose we have p inputs, denoted by $x_1, \ldots, x_p$, to the neuron denoted by y :

$$\begin{aligned} y_{in} &= w_1 x_1 + \cdots + w_p x_p \\ &= \sum_{i=1}^{p} w_i x_i \end{aligned}$$

The step function with threshold θ is defined as :

$$f(x) = \begin{cases} 1 & \text{if } x > \theta \\ 0 & \text{if } x \leq \theta \end{cases}$$

Using this step function, the output or predicted value of the neuron, denoted by $\hat{y}$, is :

$$\hat{y} = \begin{cases} 1 & \text{if } \sum_{i=1}^{p} w_i x_i > \theta \\ 0 & \text{if } \sum_{i=1}^{p} w_i x_i \leq \theta \end{cases}$$

It should be noted that $\hat{y}$ will be used throughout this dissertation to denote the fitted value produced from the network while y will be used to denote the value of the target or dependent variable. The whole process of modelling a basic neuron is graphically illustrated in figure 2.1:

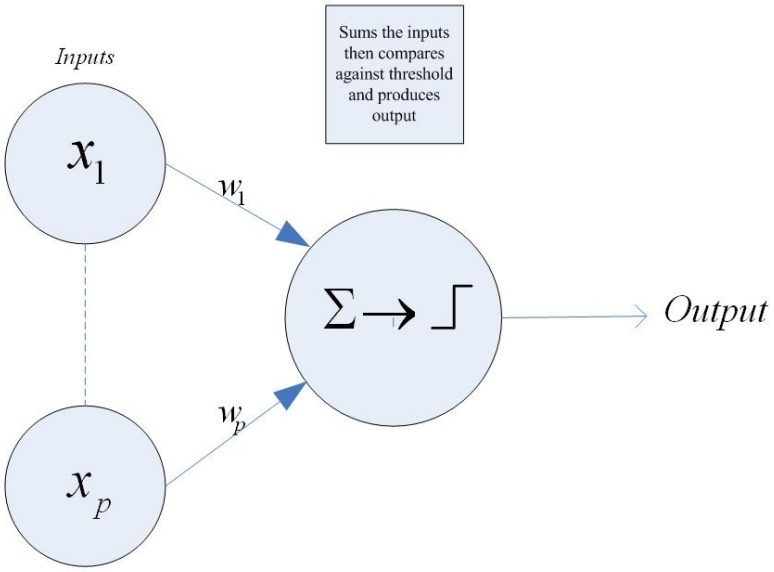


Figure 2.1: Outline of basic model

We see that the inputs and the outputs are illustrated as neurons. The input neurons do not perform any computation; they receive their signals or values from the outside world, similar to how data is obtained on the independent variables in a statistical context. Hence when we have a data set, the observations on the independent variables in the data set are the inputs. The input neurons are arranged in what is known as the input layer. Each of the neurons in the input layer are connected through a directed, weighted path to the output neuron, in the output layer. The value coming from a neuron is known as the neuron's activation in the neural network architecture. In this example the input neurons pass their values on to the output layer through the connection paths in a forward way, meaning that the output obtained is not fed back to the input layer in any way. This is referred to as a feedforward neural network (cf. 2.3.4). The direction of

the connection paths does not necessarily have to be forward. There may also be more than one output neuron in the output layer. Neurons can be connected in many different ways to define different neural network architectures. More information on other neural network architectures can be found in Boden (2001); Hertz et al. (1991).

In the model of the simple neuron as defined earlier, the threshold can in essence be subtracted from the net input and the result then compared to zero. On the basis of this outcome the network produces an output. This is just an alternative way of applying the threshold to the neuron.

A constant can also be added to the net input, to increase or decrease the net input into the neuron. This is called a bias in the neural network literature. This bias is usually included as an extra input, denoted as input x_0, for which the value is fixed to be 1. The weight on this input then determines the value with which the input to the next neuron should be offset. This weight can be positive or negative. This is similar to what is done in a regression model when a constant term is added. The equation describing the output can then be written as

$$y = f\left[\sum_{i=0}^{p} w_i x_i\right] \qquad \text{where } f(x) = \begin{cases} 1 & \text{if } x > \theta \\ 0 & \text{if } x \leq \theta \end{cases}$$

Another approach is to incorporate the threshold, θ, into the bias and then have just one constant. The equation for the simple neuron then becomes:

$$y = f\left[\sum_{i=0}^{p} w_i x_i\right] \qquad \text{where } f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

This is the most simple way to describe the model of the basic neuron which includes a bias. It should just be noted that there are many different ways in which this neuron can be constructed and that a neural network model can include a bias term and also a threshold term on the activation function. The difference being that a bias is added to the net input into a neuron while a threshold is subtracted.

Figure 2.2 gives an illustration of how a single layer network is generally represented graphically:

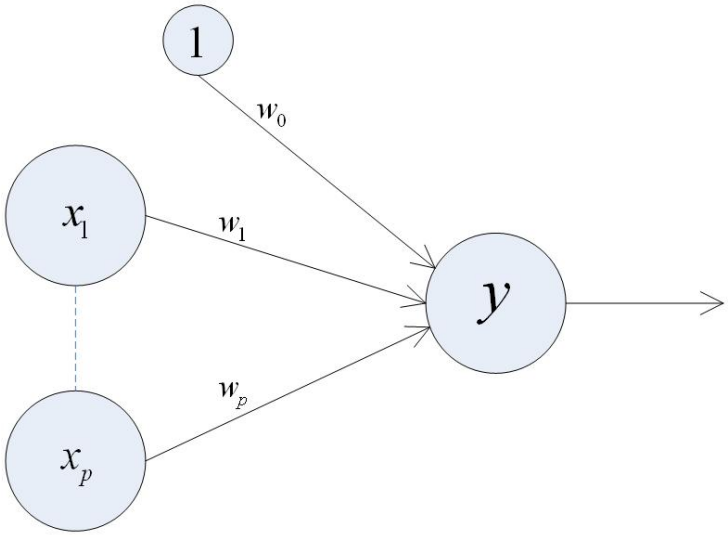


Figure 2.2: A single layer neural network

The step function which is used above is an example of an activation function in neural network literature. This function can be any function although certain activation functions provide favorable properties which will be seen later. The activation functions which are used most often is a hard limiting function (either a step or the signum function), a piecewise linear function, or a soft limiting function (for example the sigmodial or *softmax* function). An example of each of the activation functions is given next.

The step or threshold function is defined as:

$$f(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

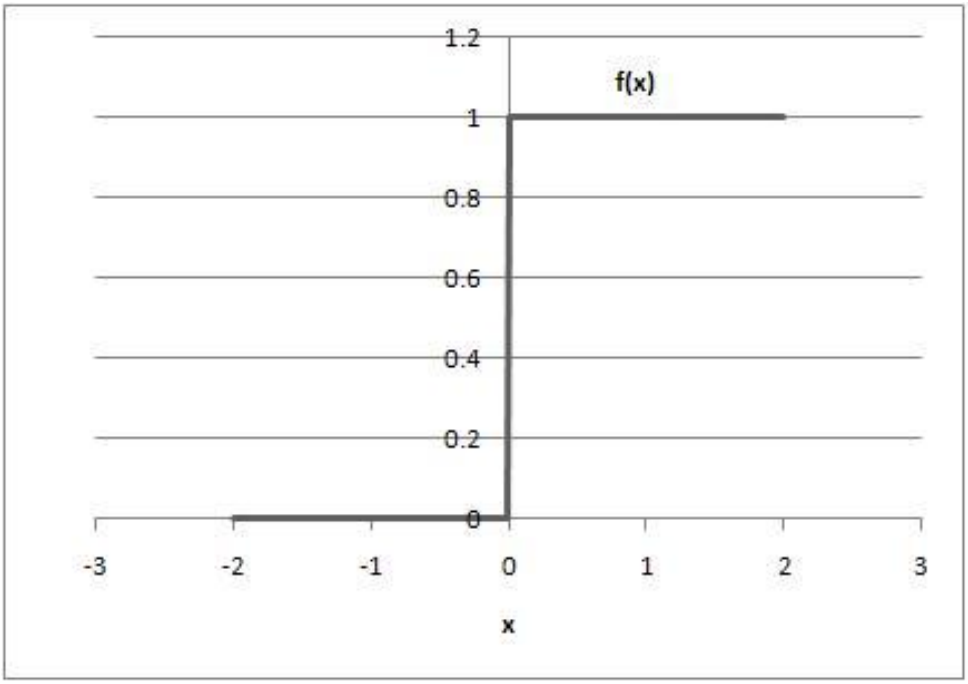


Figure 2.3: Step function

The piecewise linear function is defined as:

$$f(x) = \begin{cases} 1 \text{ if } x \geq 0,5 \\ x + 0.5 \text{ if } -0,5 < x < 0,5 \\ 0 \text{ if } x \leq -0,5 \end{cases}$$

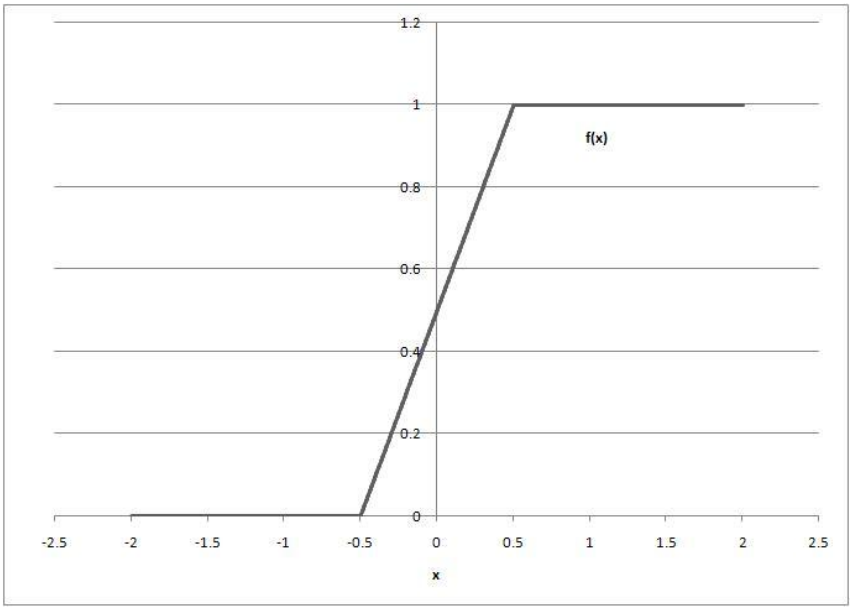


Figure 2.4: Piecewise linear function

The sigmoidal function is the most popular activation function used. One of the favorable characteristics of the function is that it is differentiable and that the derivative can be expressed in a convenient form as will be shown later. An example of a sigmoidal activation function is the logistic function defined as:

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

where a is the slope parameter. The logistic function has a squashing effect, i.e. it scales values to lie in a small range. For the logistic function, the values are scaled to lie between

0 and 1 which means it is also a favorable activation function to use when we want to model probabilities (see section 2.4.3). An illustration of a logistic function for various values of a is shown in figure 2.5.

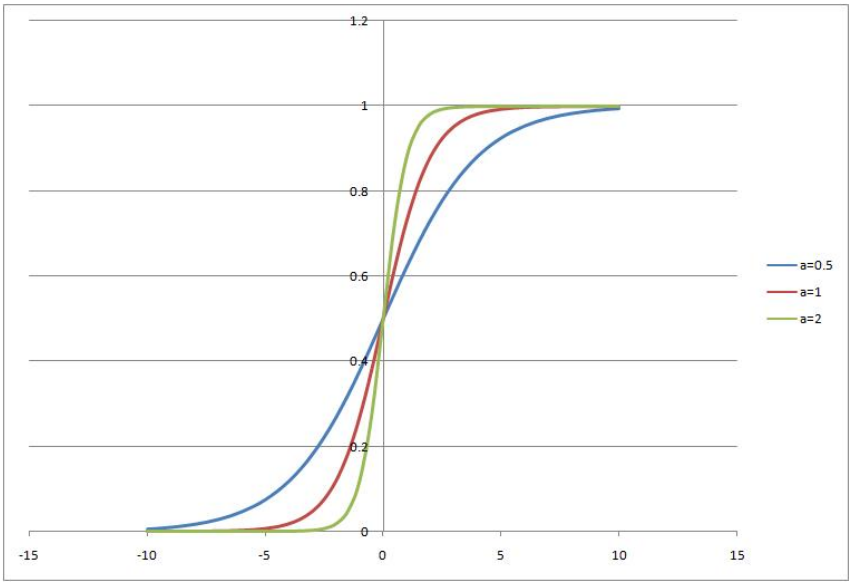


Figure 2.5: Sigmoidal function with various slope parameters

The activation function used is very similar to the link function in generalised linear models and if a linear activation function is used, the single layer network exactly resembles a linear regression model in statistics. The link between neural networks and similar methods in statistics will be discussed in section 2.4.

The perceptron

Single-layer networks with threshold activation functions were given the name *perceptrons* by Frank Rosenblatt (Rosenblatt, 1958, 1962). These perceptrons were applied to simple classification problems and usually received binary (0 or 1) or bipolar (-1 or 1) inputs. The perceptron produced an output, $\hat{y}$, by calculating the weighted sum of the inputs, including a bias, which is denoted as the *net input*. This *net input* into the neuron in the output layer is compared to a threshold and then produces a specified output based on whether the *net input* exceeds this threshold value. In mathematical terms this is denoted as

$$\hat{y} = f(net\ input) = f(\sum_{j=0}^{p} w_p x_p) \tag{2.4}$$

where $f(\cdot)$ denotes a threshold function and $x_0 = 1$ signifies the inclusion of a bias term. An example of a threshold function that can be used is (Fausett, 1994, p.59):

$$f(net\ input) = \begin{cases} 1 & \text{if } input > \theta \\ 0 & \text{if } -\theta \leq input \leq \theta \\ -1 & \text{if } input < -\theta \end{cases}$$

The threshold θ, can be any arbitrary but fixed value. This is not the only threshold function available and we will look at another threshold function which can be used later in this section. The weights in this perceptron were calculated with the perceptron learning rule. The perceptron learning rule is an iterative algorithm for calculating the parameters (i.e. the weights) in the network. The basic idea of the algorithm is that each observation is presented to the network one at a time. The network then calculates the output from the given input and the weights are adapted to reduce the error made by the network. This process is often termed *that the network learns from experience*, although it is nothing more than parameter estimation.

This algorithm is suitable for either binary or bipolar inputs and the output from this perceptron is bipolar. A fixed threshold, denoted by θ, and adjustable bias, denoted by w_o is assumed for this algorithm. The algorithm is not particularly sensitive to the starting values of the weights. To ensure better convergence of the algorithm a learning rate, denoted by α, can be added to the algorithm. The algorithm will be described in more detail now for a simple classification problem, where the objective is to classify an observation to one of two classes based on a given input. (Fausett, 1994; Beale and Jackson, 1990):

Assume each training observation consists of p inputs, $\mathbf{x}_i = (x_{i1}, \ldots, x_{ip})$, and corresponding response, y_i, where the response is coded as:

$$y_i = \begin{cases} 1 & \text{if the observation is from class 1} \\ -1 & \text{if the observation is from class 2} \end{cases}$$

Assume that there are n training observations, $i = 1, \ldots, n$.

Perceptron learning rule

Step 0: Choose initial values for the weights $\mathbf{w}^0 = (w_0, \ldots, w_p)$. (For simplicity the weights may all be set to 0.)

Choose a learning rate α $(0 < \alpha \leq 1)$. (For simplicity, the learning rate can be set to 1.)

Step 1: Present a training pattern of p-inputs and response, $(\mathbf{x}_i; y_i)$, $i = 1, \ldots, n$, and compute the fitted value, $\hat{y}_i$, from the network with:

$$in_i = \sum_{j=0}^{p} w_j x_{ij};$$

$$\hat{y}_i = f(in_i)$$

where $f(\cdot)$ is the activation function used.

Step 2: Update the weights if an error ($\hat{y}_i \neq y_i$) was made for this particular observation by using:

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \alpha y_i \mathbf{x}_i$$

If no error was made for this observation the weights do not change.

Step 3: Steps 1 and 2 are repeated for all the training patterns, each time computing the output from the perceptron for the pattern and then adjusting the weights if the pattern is not correctly classified. This process is continued until the weights do not change anymore or if the weights do not change significantly for each iteration.

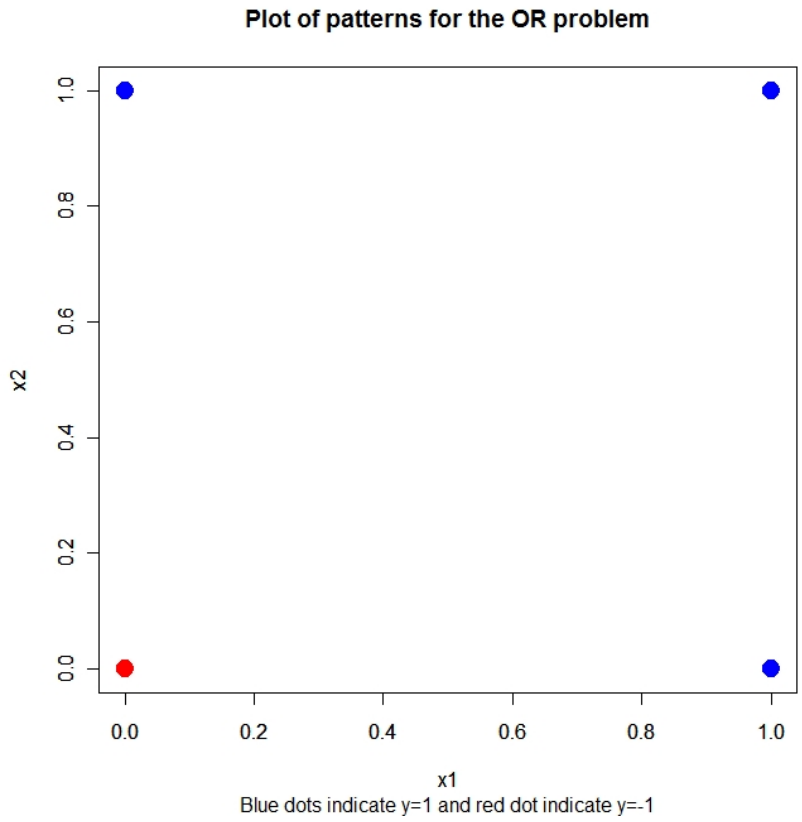
From this algorithm we see that since the response variable is coded as a bipolar variable (-1 or 1), the weights are changed in the direction of the response variable if an error was made. Note that only the weights on the connections from the inputs, where not all the inputs are zero, will be changed with this procedure. To see how the perceptron leaning rule works we will first consider a simple example.

Example: Solving a simple classification problem using the perceptron learning rule Before we continue with more complex single-layer networks and learning rules, it may be beneficial to give a very simple example of a problem that can be solved using the perceptron learning rule that was just discussed. This will give insight into how these single layer neural networks work. This should also give a better understanding of the notation used.

The problem that will be explained here is called the OR problem in computer science. The perceptron takes two binary inputs (x_1 and x_2) and needs to classify each pattern into one of two possible classes. More specifically, the patterns are given as:

x_1	x_2	y
0	0	-1
0	1	1
1	0	1
1	1	1

A plot of these observations are presented is figure 2.3.3. From this plot of the data we see that this problem is perfectly linearly separable meaning that a linear boundary will perfectly separate the blue dots ($y = 1$) from the red dot ($y = -1$).



The objective is to solve this simple problem by using the perceptron learning rule that was discussed earlier in this section. We will use the following activation function for this

problem:

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

We start by initialising all the weights in the perceptron to 0, i.e. $w^0 = (w_0, w_1, w_2) = (0, 0, 0)$. Note that a bias is included in the perceptron, which means that an extra weight, w_0, and input $x_0 = 1$ needs to be included. A learning rate of $\alpha = 1$ is chosen.

If we present the first pattern $\boldsymbol{x}_1 = (x_0, x_1, x_2) = (1, 0, 0)$ we get:

$$\begin{aligned} in_1 &= w_0 \times x_0 + w_1 \times x_1 + w_2 \times x_2 \\ &= 0 \times 1 + 0 \times 0 + 0 \times 0 \\ &= 0 \end{aligned}$$

The output from the perceptron is:

$$\hat{y}_1 = f(0) = 1$$

Now we have that $y_1 = -1 \neq 1 = \hat{y}_1$ meaning that the weights should be adjusted. The adjustment of the weights are calculated as:

$$\begin{aligned} \boldsymbol{w}^{(1)} &= \boldsymbol{w}^{(0)} + y_1 \boldsymbol{x}_1 \\ \begin{pmatrix} w_0^{(1)} \\ w_1^{(1)} \\ w_2^{(1)} \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + (-1) \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \end{aligned}$$

After the first pattern has been presented the weights in the perceptron are:

$$\boldsymbol{w}^{(1)} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}$$

Now using these new weights, the second pattern, $\boldsymbol{x}_2 = (1, 0, 1)$, is presented to the perceptron. The output from this pattern is calculated as:

$$\hat{y}_2 = f(-1 \times 1 + 0 \times 0 + 0 \times 1) = f(-1) = -1$$

We have that $\hat{y}_2 \neq y_2$ and hence the weights should be adjusted. After adjusting the weights we have that:

$$\boldsymbol{w}^{(2)} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

For the third pattern the output is $\hat{y}_3 = 1$. and hence we have that $\hat{y}_3 = y_3 = 1$. This means that the weights are not adjusted at this iteration, i.e. $\boldsymbol{w}^{(3)} = \boldsymbol{w}^{(2)}$. Presenting the fourth pattern, to the perceptron we find $\hat{y}_4 = 1$ and hence the weights are not adjusted for this pattern and $\boldsymbol{w}^{(4)} = \boldsymbol{w}^{(3)}$.

Now that we have presented all the patterns to the perceptron we cycle through the patterns again by starting from the first pattern again. Each cycle through all the patterns is referred to an an epoch in neural network literature. If we present the first pattern again to the perceptron using the weights

$$\boldsymbol{w}^{(4)} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

we find that $\hat{y}_5 = f(0 \times 1 + 1 \times 0 + 0 \times 0) = f(0) = 1$. Because the predicted value is not correct the weights should be adjusted and we find:

$$\boldsymbol{w}^{(5)} = \begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}$$

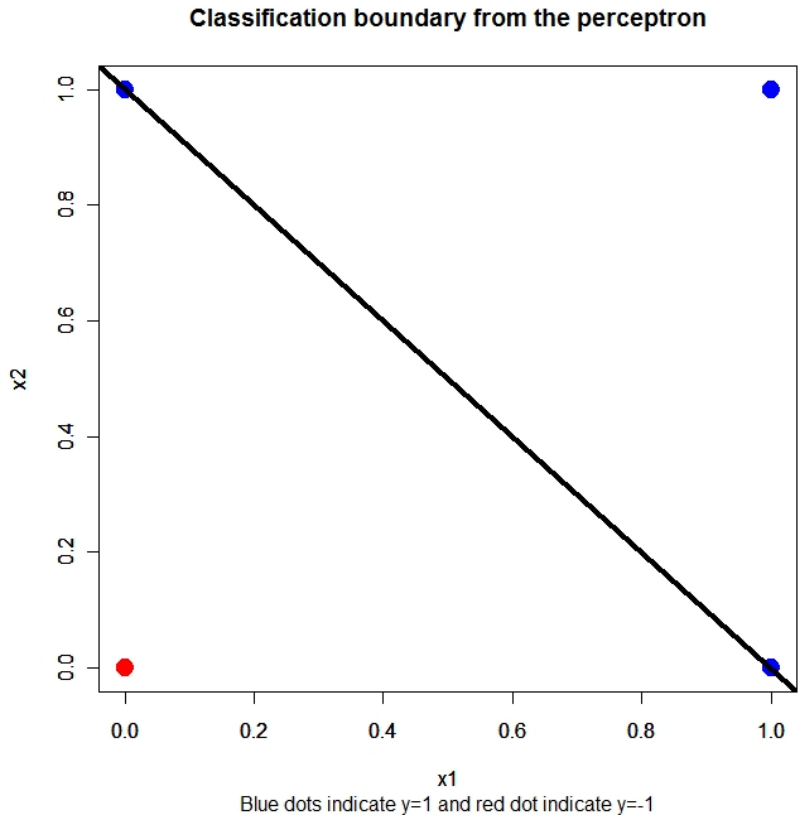
We will continue in this fashion until the weights do not change from iteration to iteration. After 13 iterations, we find the last change in the weights and after that all the patterns are classified correctly using those weights. The weights which provide this solution is:

$$\boldsymbol{w}^{(13)} = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}$$

A plot of the classification boundary that is fitted by the perceptron is given in figure 2.3.3. The line in the plot is the classification boundary, with every point on or above the line being classified as a 1 and all the points below the line being classified as a -1. Note that this line perfectly separates all the observations from the 1 category (blue dots) from the observation which is labeled as -1 (red dot).

It can be proved that the perceptron learning rule will converge to a solution in a finite number of steps if the data is linearly separable, i.e. the two classes can be perfectly separated by a linear boundary. There are a number of problems with this algorithm. These problems include: The solution obtained is not unique and depends on the starting values, the algorithm can take very long to converge and if the data is not linearly separable, the algorithm will not converge (Ripley, 1996).

A number of solutions have been proposed to eliminate these problems like using kernels



to transform the input space to allow for better separability between the groups and adding extra constraints to the algorithm to ensure convergence. This forms the basis for another learning method called *support vector machines*. This topic will not be pursued further here but the interested reader can obtain more information and references on this topic in Cristianini and Shawe-Taylor (2000); Vapnik (1996).

ADALINE

The ADAptive LINEar neuron is also a single layer network very similar to the perceptron. It was developed in 1960 by Bernard Widrow and Marcian Hoff (Widrow and Hoff, 1960). The network typically also uses binary or bipolar inputs and outputs, although it is not limited to these only. Similar to the perceptron, the ADALINE arranges the inputs in a layer and the inputs are then connected in a forward direction to the output in the output

layer. The weights on the connections between the input and output layers are adjustable and the bias is the weight on the connection of an input which is permanently set to 1. The activation function is typically linear but this can be any function.

The major difference between the ADALINE and the perceptron as described above is the way in which the network is trained, i.e. the weights are estimated. In general the weights are estimated with the Widrow-Hoff rule. This algorithm is also known as least mean squares. The Widrow-Hoff rule can also be used to estimate the parameters in a network where there is more than one output neuron in the output layer.

The basic idea of the Widrow-Hoff rule is similar to the perceptron learning rule. The fundamental change is in that the weights are not only changed if an error is made in the classification, but that the weights are adapted proportional to the error made by the network. This implies that the weights will be adapted a lot if a large error is made and vice versa.

Any activation function can be used with the Widrow-Hoff rule, because as will be shown now, the weights are adapted in relation to the error made on the output before the activation function is applied. This means that we use the *net input*, denoted by in_i, into the output neuron to update the weights. For the final classification, the activation function should be applied. This algorithm can be written in more mathematical detail as in (Fausett, 1994; Beale and Jackson, 1990):

Suppose that each training observation has p-inputs, $\mathbf{x}_i = (x_{i1}, \ldots, x_{ip})$, each with corresponding output y_i , where $i = 1, \ldots, n$ is the number of training observations. For classification purposes the response variable can be coded as a binary or bipolar variable, however we will not make any assumption on the form of the inputs and outputs, making the network more general.

Widrow-Hoff rule:

Step 1: Choose starting values for the weights $\mathbf{w}^0 = (w_0^0, \ldots, w_p^0)$. Small random weights are usually used.

Set the learning rate α.

Step 2: For each training observation $(\mathbf{x}_i; y_i)$, compute the net input to the output neuron:

$$y_{in_i} = \sum_{j=0}^{p} w_j x_{ij} \quad \text{where } x_{i0} = 1$$

Step 3: Update the weights using the following update rule, $i = 1, \ldots, n$:

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \alpha(y_i - y_{in_i})\mathbf{x}_i$$

Step 3: Continue with this algorithm until a suitable stopping criterion is satisfied, e.g. the largest change in the absolute value of the weight in the previous step is smaller than a predetermined value.

The Widrow-Hoff rule changes the weights in the neural network to minimize the difference between the *net_input* to the output unit, and the response variable, y, for an observation. The aim of the algorithm is to minimise the error over all the observations. This can be done in two ways: The way in which the algorithm is described above, by minimising the error one observation at a time, which is referred to as online learning or by accumulating the errors until all the observations in the data set have been presented and then updating all the weights by using the accumulated adjustment. This is referred to as batch learning.

The learning rate should be chosen with care. If a too small learning rate is chosen, the algorithm will take long to converge. If too large value is taken the algorithm may not converge. For a single layer network with one output neuron, a guideline to set the

learning rate is to choose a value such that $0,1 \leq p\alpha \leq 1$, where p is the number of inputs or independent variables. An adaptive learning rate can also be chosen and will be discussed in section 2.3.5.

Delta rule The Widrow-Hoff rule was extended to allow continuous, differentiable and monotone increasing activation functions to be applied to the network. This learning rule is called the delta rule and was introduced by McClelland and Rumelhart (1986, 1988). The delta rule can be used to determine the weights in a single layer network, by minimising an objective function with regards to the weights in the network, similar to the method of Widrow-Hoff. The difference between the Widrow-Hoff rule and the delta rule is in that the derivative of the activation function is used to update the weights. This is why the activation function needs to be differentiable in order for the delta rule to be applied.

The delta rule uses the gradient descent optimisation. The gradient descent technique states that the objective function (sum of squares for error is usually used) decreases the fastest in the direction of the negative gradient. This means that when using the delta method, the weight changes are in the direction of the negative gradient.

The algorithm for training a single layer network with differentiable activation functions is the same as that for the Widrow-Hoff rule and only the update rule for the weights change. To derive the delta rule is mathematically simple. Using the same notation as for the Widrow-Hoff rule, the sum of squares error function, denoted by E, for a single layer network with differentiable activation function $f(\cdot)$ can be written as:

$$E[\mathbf{w}] = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - f(\sum_{j=0}^{p} w_j x_{ij})\right)^2 = \frac{1}{2}\sum_{i=1}^{n}(y_i - f(in_i))^2$$

Differentiating this expression w.r.t. the weight w_j, using the chain rule, we obtain:

$$\frac{\partial E}{\partial w_j} = -\sum_{i=1}^{n} (y_i - f(in_i))\, f'(in_i) x_{ij}$$

so that the accumulated weight update (batch learning update) for w_j , $j = 1, \ldots, p$ is:

$$w_j^{t+1} = w_j^t + \alpha \sum_{i=1}^{n} (y_i - \hat{y}_i) f'(in_i) x_{ij}$$

The weight updates does not need to be accumulated until all training observations are presented. The online version for updating the weights w_j , $j = 1, \ldots, p$ after presenting each pattern x_{ij} , $i = 1, \ldots, n$ to the network is:

$$w_j^{t+1} = w_j^t + \alpha \Delta w_j x_{ij}$$

with

$$\Delta w_j = (y_i - \hat{y}_i) f'(in_i)$$

where $0 \leq \alpha \leq 1$ is the learning rate and $j = 0, \ldots, p$. We see that the online version is exactly the same form as the batch version, the only difference being that the sum does not occur in the weight update. This means that the weights are updated after each pattern is presented to the network instead of accumulating the weight updates for all the patterns present and then updating the weights after each cycle through all the patterns (also called an epoch).

Single-layer networks with more than one output unit

The networks that have been discussed thus far all have a single output unit. We can now make the network more general by including more than one output unit in the network and in this section an introduction will be given to this type of network. These types of

networks are often used for classification problems where the objective is to classify an observation to a class, where there are more than two classes to which the observation can belong to. A single-layer network with more than one output variable is closely related to discriminant analysis in statistics. For example, each of the outputs can be used to model the probability of an input belonging to each of those classes and in section 2.4.3 we will look at these networks and how they fit into a statistical methodology.

Let $\mathbf{x}_i = (x_{i1}, \ldots, x_{ip})$, $(i = 1, \ldots, n)$, be p inputs with corresponding response variables $\mathbf{y}_i = (y_{1i}, \ldots, y_{Ki})$. To shorten the notation, the subscript which indicates the observation number will be dropped and an observation which is presented to the network will be denoted by $(\mathbf{x}; \mathbf{y})$, where $\mathbf{x} = (x_1, \ldots, x_p)$ denotes the input variables and $\mathbf{y} = (y_1, \ldots, y_K)$ denotes K response variables corresponding to the same observation. Let $f_k(.)$, denote the activation function of the k-th output neuron. These activation functions do not necessarily need to be the same. More information on the activation functions in networks where there is more than one output will be given in section 2.3.4.

This network is graphically illustrated in figure 2.6.

The weights, w_{jk} indicate that it is the connection weight from input neuron j connecting to output neuron k. Mathematically the output from output neuron k, $k = 1, \ldots, K$, when presented with an observation $\mathbf{x} = (x_1, \ldots, x_p)$ is:

$$\hat{y}_k = f_k\left(\sum_{j=0}^{p} w_{jk} x_j\right) \qquad \text{where } x_o = 1$$

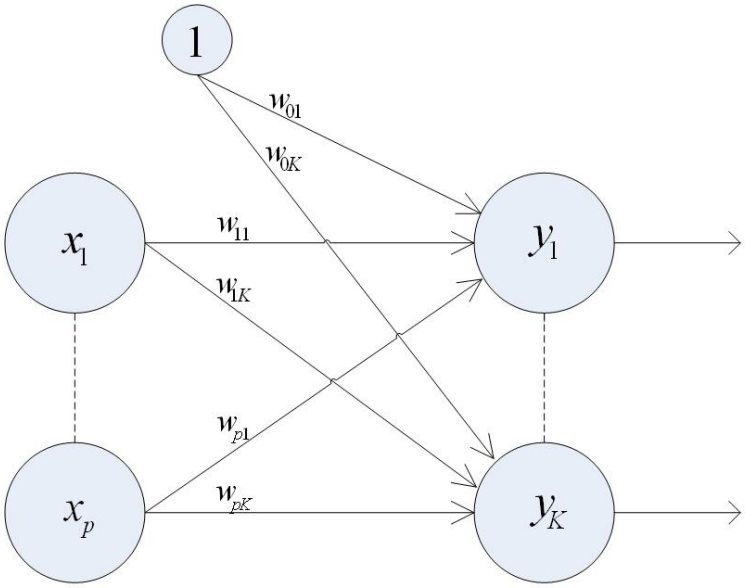


Figure 2.6: A single layer neural network with multiple outputs

Delta rule for several output units

The delta rule can be extended to estimate the weights in single layer networks which have more than one output unit. Suppose there are n observations, each of the form $\mathbf{x} = (x_1, \ldots, x_p)$, with corresponding response variables $\mathbf{y} = (y_1, \ldots, y_K)$. The net input into output neuron k from observation $\mathbf{x}$ is denoted by y_{in-k}, where $k = 1, \ldots, K$. Denote w_{jk} as the weight from the j-th input neuron, to the k-th output neuron. Denote the predicted value on output neuron k by $\hat{y}_k$. The delta rule to update the weights for this network follows similarly to the one for a network with only one output. Using the notation defined the algorithm is now:

Step 1: Choose starting values for the weights $\mathbf{w}_0^0, \mathbf{w}_1^0, \ldots, \mathbf{w}_K^0$. Small random weights are usually used.

Set the learning rate α.

Step 2: For each training observation $(\mathbf{x}; \mathbf{y})$, compute the net input to each of the output neurons:

$$y_{in-k} = \sum_{j=0}^{p} w_{jk} x_j \quad \text{where } x_0 = 1$$

Apply the activation function to each of the values obtained to get a predicted value, denoted by $\hat{y}_k$ on each of the output neurons:

$$\hat{y}_k = f(y_{in-k}) \qquad k = 1, \ldots, K$$

Step 3: Update the weights using the following update rule:

$$\mathbf{w}_k^{t+1} = \mathbf{w}_k^t + \alpha(y_k - \hat{y}_k) f'(y_{in-k}) \mathbf{x}$$

Step 4: Continue to repeat these steps for each of the observations in the data set until a suitable stopping criterion for the algorithm is satisfied.

The weight corrections can also be accumulated until every observation in the data set have been presented and then the weights are updated before each observation is then presented again to the network.

2.3.4 Multilayer networks

In this section an introduction to multilayer networks will be given. Different network architectures and estimation techniques for multilayer networks will be discussed. The single layer networks discussed in the previous section can represent a very limited range of functions. To estimate more general functions, multilayer networks can be used.

Feedforward multilayer perceptron

To give an introduction to multilayer networks we are going to start by combining two single layer networks to form a two layer network. The first network has p inputs, $x_1, \ldots, x_p$, each of these inputs are fully connected to M output neurons denoted by $z_1, \ldots, z_M$. The second network then uses these M outputs, $z_1, \ldots, z_M$, of the first network as its inputs. These M inputs of the second network are then fully connected to K output units. These output units in the second network produce the outputs of the whole model which are denoted by $\hat{y}_1, \ldots, \hat{y}_K$. Bias terms can be added to both the inputs into the first and second network, by including neurons which have activations (values) permanently set to 1. All the inputs and outputs of the two networks are arranged in layers. The inputs to the first network are called the input layer. The outputs of the first network (which also serve as the inputs to the second network) are called the hidden layer and the outputs from the second network are arranged in the output layer. Graphically this can be illustrated as:

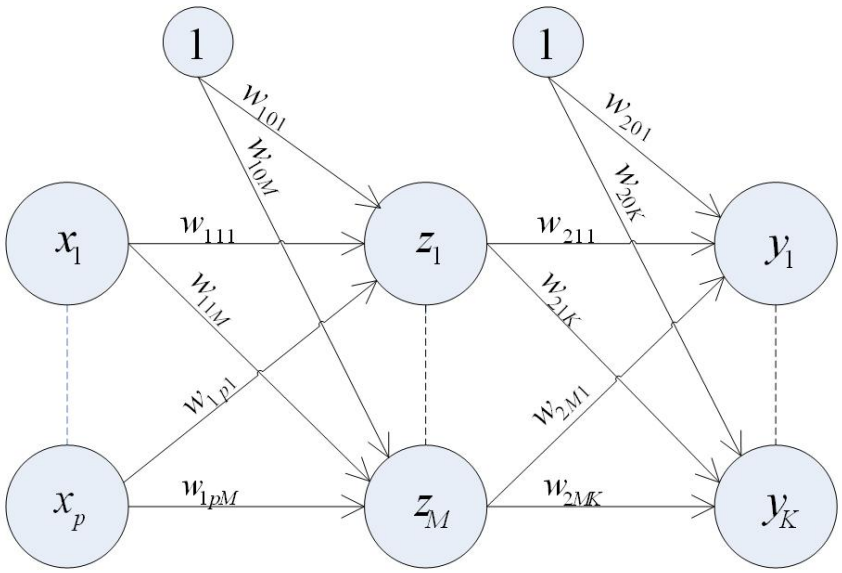


Figure 2.7: A multilayer network

In this diagram, the weights from input neuron j to hidden layer neuron m are denoted

by w_{1jm} and the weights from hidden layer neuron m to output neuron k are denoted by w_{2mk}.

This is known as a two layer feedforward neural network. The reason why it is a two layer network is because there are two layers of weights which need to be estimated in this network, and it is a feedforward network because the neurons in each layer contribute to the input of the neurons in the next layer. Two layer networks provide a huge improvement over single layer networks in that two layer networks can approximate any continuous function arbitrary closely (Bishop, 1995, p.116). It should be noted that the weights that need to be estimated in the neural network are sometimes referred to as the adaptive weights while the parameters of the neural network generally refer to the number of hidden layers, the number of neurons per layer and also the adaptive weights. Thus when we say we need to estimate the parameters of the neural network, this generally refers to estimating the number of hidden layers, the number of neurons per hidden layer and also the connective weights between the neurons.

Step by step, the outputs from a two layer network when presented with an observation $\mathbf{x} = (x_1, \ldots, x_p)$, can be calculated as follows: Suppose the network has p inputs in the input layer, M hidden nodes in one hidden layer and K output nodes in the output layer (similar design as in diagram 2.7).

1. For each of the hidden neurons, $z_1, \ldots, z_M$, compute a weighted sum of the inputs, including a bias term:

$$z_{in-m} = \sum_{j=o}^{p} w_{1jm} x_j \qquad \text{with } x_0 = 1$$

where z_{in-m} denotes the net input to hidden neuron z_m when presented with an observation $\mathbf{x}$.

2. Pass this value, z_{in-m} to the activation function, $f_m(\cdot)$, on that particular unit to get an output value from hidden neuron m, which is denoted by z_m. That is, for $m = 1, \ldots, M$ calculate:

$$z_m = f_m(z_{in-m})$$

3. For each of the neurons in the output layer, $y_1, \ldots, y_k$, calculate the net input to each of these neurons by computing a weighted sum of $z_1, \ldots, z_M$, including a bias term:

$$y_{in-k} = \sum_{m=0}^{M} w_{2km}\, z_m \qquad \text{with } z_0 = 1$$

4. Use the activation function to get a final fitted value, $\hat{y}_k$. This implies that the fitted values $\hat{y}_k$, $k = 1, \ldots, K$ are calculated by:

$$\hat{y}_k = g_k(y_{in-k})$$

Combining these steps, we see that the output on neuron k in the output layer when the two layer network is presented with an observation $\mathbf{x} = (x_1, \ldots, x_p)$ is:

$$\hat{y}_k = g_k\left(\sum_{m=0}^{M} w_{2mk}\, f_m\left(\sum_{j=o}^{p} w_{1jm} x_j\right)\right)$$

where $g_k(\cdot)$ and $f_m(\cdot)$ are the activation functions in the output and hidden layer respectively.

The above design can be generalised further to include far more complex designs in which there are more layers of hidden units. For the purpose of this dissertation, only the two layer network will be discussed since the methods for higher order networks are similar.

2.3.5 Learning in multilayer networks

Up to this point we have discussed single layer networks which may or may not have more than one output. The inputs and outputs were each arranged in a layer and there is only one layer of weights between the input and output layer which needs to be estimated. In the ADALINE, this layer of weights was estimated by the Widrow-Hoff or delta rule. The development of a method for estimating the weights in multilayer networks was one of the main reasons for the reemergence of interest in the fields of neural networks. Training a multilayer network is usually done by setting the weights in the network to minimise an objective function. Different objective functions can be used for this purpose but the sum of squares for error function is usually chosen for regression problems and the cross entropy error function for classification problems (cf. Section 2.4).

The sum of squares for error function is well-known in statistics, as most statistical regression models are fitted by this criterion. In neural networks the sum of squares error function is a function of the weights of the network similar to how the error function is a function of the parameters in a regression model. The objective is then to choose values for the weights in the network, such that the error function is a minimum. Graphically, the process of minimising the error function is illustrated in figure 2.8.

From figure 2.8, it can be seen that the error function forms a surface above a p-dimensional weight space, where the weight space in this example can be denoted by $\underline{w} = (w_1, w_2)$. The shape of the error surface in figure 2.8, will typically depict the general form of an error surface from a single layer network, with linear activation functions, where the network was fitted using a sum of squares error function, much like in a linear regression model. The objective is to choose the values of w_1 and w_2, such that the error (denoted by E), is a minimum, that is where $\nabla E = 0$. It can be seen from fig 2.8 that the error surface has more than one minimum value (at point A and B) and one of the

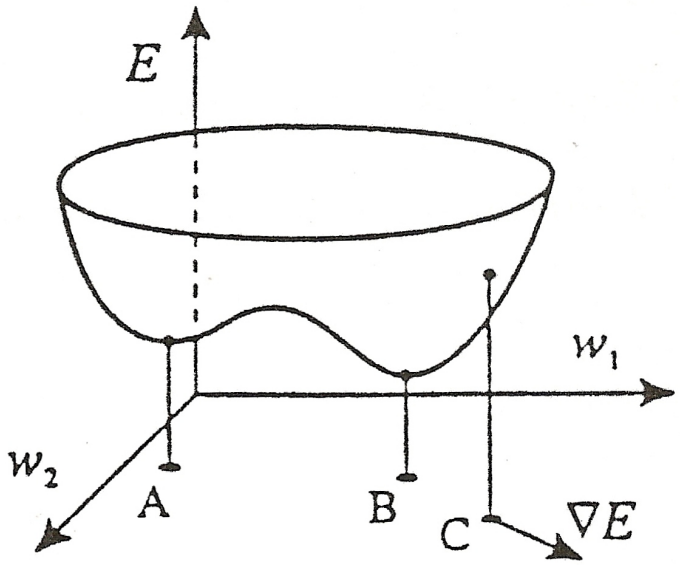


Figure 2.8: Geometrical illustration of an error surface (Bishop, 1995, p.254)

main problems with optimisation methods is that the optimisation technique can become stuck in a local minimum (like point A) instead of converging to a better solution (point B).

The error surface for a neural network will generally be a highly nonlinear function of the weights. This is especially true if the network has more than one layer of weights. This will cause the error function to have numerous points where $\nabla E = 0$ which are called stationary points. Figure 2.9 illustrates this point for an error function E against one weight w. For this graph, there are four points where the gradient of the error function would be equal to zero. Point A is a local minimum, since this point is the lowest for a small neighborhood around A but it is not the smallest value across the total error function. Point B is a local maximum. Point C is a saddle point, and some algorithms can get stuck on this flat surface for prolonged periods of time. Point D is a global maximum, which is the desired value of the error function (for an optimal network architecture cf.

Section 2.4.4).

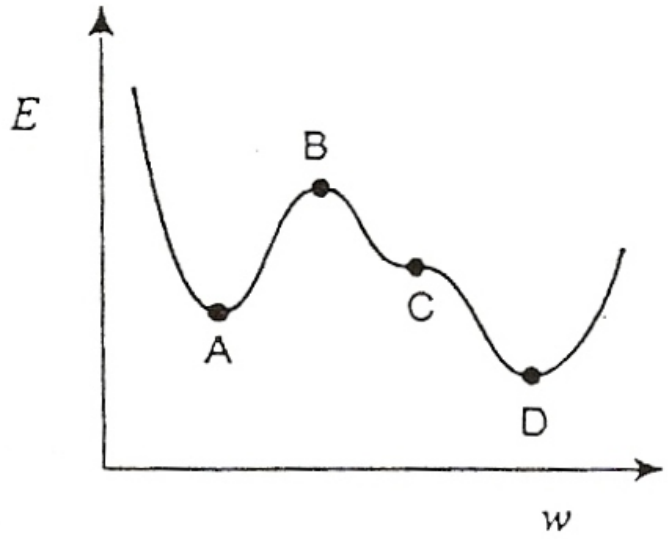


Figure 2.9: An error function with four stationary points (Bishop, 1995, p.255)

Because the error surface is a highly nonlinear function of the weights, explicit expressions for the global minimum of the error function cannot be obtained. To overcome this problem we must make use of a numerical optimisation method to train a neural network. The main objective is to use an efficient method which will locate the global minimum of the error function.

The basis for locating the global minimum of the error function for a neural network is done by using an algorithm that conducts a search through the weight space. The basic idea of this search is to start by choosing any point on the error surface, and then proceed some distance in an appropriate direction that will decrease the error function. This process continues until a minimum is reached. This can be formulated mathematically as:

$$\underline{w}^{(t+1)} = \underline{w}^{(t)} + \Delta\underline{w}^{(t)} \tag{2.5}$$

where t is the step number, $\underline{w}$ is the weight vector and $\Delta\underline{w}^t$ is the adjustment to the

weight vector at time t (Hill and Lewicki, 2006). The two questions that can be asked from this procedure is: what is the appropriate direction to move in and what distance must the step length be?

Different algorithms involve different choices for the direction in which the weights must be updated and the step length that must be used. This essentially means that different algorithms involve different ways of choosing the weight adjustment $\Delta\underline{w}^{(t)}$. We will start by looking at backpropagation in the next section and will also provide an overview of other methods which can be used for neural network training in section 2.3.5.

Error backpropagation

The gradient method forms the basis for the backpropagation algorithm, which is the most popular method for estimating the weights in a multilayer perceptron. Recall from section 2.1.3, that the gradient at any point in the weight space , i.e. $\nabla\underline{w}$, is the direction in which the gradient increases most rapidly. This is illustrated in the figure 2.8, in which the gradient is calculated at the point C. Gradient descent is then based on the principle that the negative gradient of the error function E at a point in the weight space, indicates the direction in which the error function decreases the fastest.

This local gradient will often not point directly to the minimum value and therefore gradient descent takes a number of small steps through the weight space, each time moving a small distance in the direction of the negative gradient at that specific point. The length of these steps is proportional to the learning parameter chosen. If a small step is taken in the direction of the negative gradient, the error function should decrease and this process is continued until a minimum is reached. By referring back to (2.5) we can

write the weight update for each iteration as

$$\Delta \underline{w}^{(t)} = -\alpha \nabla E|_{\underline{w}^{(t)}} \tag{2.6}$$

where the gradient of the error function ∇E is evaluated at the point $\underline{w}^{(t)}$.

The idea of backpropagation was independently discovered by several researchers. The most popular version of backpropagation as it is used in neural networks for adjusting the weights in the model was derived by Rumelhart et al. (1986) but similar ideas where presented by Bryson and Ho (1969); Werbos (1974); Parker (1985). The backpropagation rule is the name given to an algorithm which is used for evaluating the derivatives of the errors in a multilayer feedforward neural network when the objective is to minimise the total sum of squares for error. The backpropagation rule is also known as the generalised delta rule. Essentially the backpropagation rule uses steepest descent to minimise the sum of squares error function with regards to the weights in a multilayer network. This whole process is done in three steps: the feedforward of an input pattern to produce an output, the calculation of the error which is then propagated back through the network and based on this the weights are set in the network. These three steps, which are used to train a multilayer perceptron with backpropagation, will now be discussed. This is only done for a multilayer perceptron with one hidden layer but the algorithm can easily be extended to train multilayer perceptrons with more than one hidden layer.

Consider the two layer case as was introduced in the previous section. The second layer of weights can be estimated by regarding this layer as a simple perceptron where the inputs are given by the hidden layer, and then using the perceptron learning rule or any other learning rule for a single layer network. The problem is that the first layer of weights cannot be estimated using this technique since there are no target values on the hidden layer units. This is known as the credit assignment problem (Bishop, 1995, p.140).

> "The solution to this credit assignment problem is relatively simple. If we consider a network with differentiable activation functions, then the activations of the output units become differentiable functions of both the input variables, and of the weights and biases. If we define an error function such as the sum of squares error, which is a differentiable function of the network outputs, then this error is itself a differentiable function of the weights. We can therefore evaluate the derivatives of the error with respect to the weights, and these derivatives can then be used to find weight values which minimize the error function, by using gradient descent, or one of the more powerful optimization methods" (Bishop, 1995).

This idea will be discussed in more detail in the next section.

The backpropagation algorithm

Suppose we have a two layer feedforward network, with p input nodes, $x_1, \ldots, x_p$, in the input layer. Each node in the input layer is fully connected to M hidden nodes, $z_1, \ldots, z_M$, arranged in one hidden layer. Each hidden node is fully connected to K output nodes, $y_1, \ldots, y_K$. Let w_{1jm} denote the weights from input x_j to hidden node z_m and w_{2mk} denote the weight from hidden unit z_m to output node y_k. Denote the net input into hidden node z_m, when the network is presented with observation $\mathbf{x}$ by z_{in-m} and let z_m be the output from this hidden node when the activation function $f_m(\cdot)$ is applied to the net input. Denote the net input into output node y_k, when the network is presented with observation $\mathbf{x}$, by y_{in-k}. Apply an activation function $g_k(\cdot)$ to the net input and denote the fitted value by the network on output node y_k by $\hat{y}_k$. Suppose that the activation functions $f_m(\cdot)$ and $g_k(\cdot)$ are differentiable. The notation that is used for this two-layer neural network is illustrated in figure 2.10.

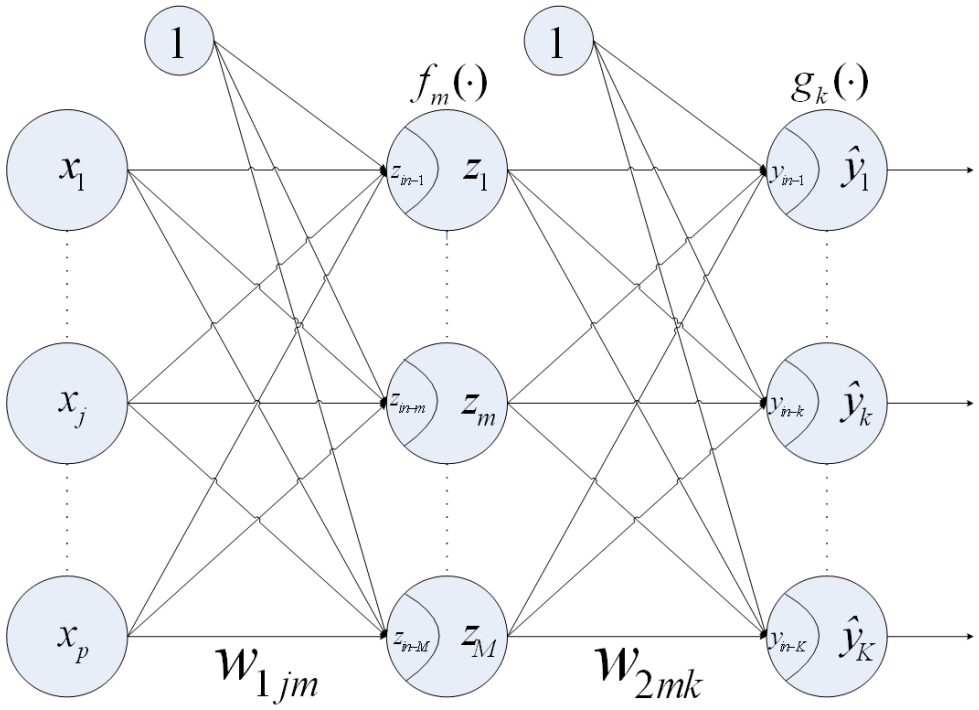


Figure 2.10: Representation of a general 2-layer neural network

We will now proceed to derive the backpropagation rule for a general two layer neural network which is trained by minimisation of the error sum of squares function. The result obtained from this derivation will enable us to obtain the gradient descent weight updates for the different types of networks that will be trained in chapter 3 of this dissertation.

Our goal is to determine suitable weights, using gradient descent, which will minimise the following sum of squares error function:

$$E[\underline{w}] = \frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - \hat{y}_{ik})^2 \tag{2.7}$$

Training the neural network means that we minimise the error over all the output units and all the observations in the training sample with respect to the weights in the neural network. We see that this error function is written as the sum over all the observations in the training data set. We can drop the subscript i to arrive at an instantaneous error function for training the neural network. Hence we need to choose the weights in the

neural network as to minimise

$$\begin{aligned} E[\underline{w}] &= \frac{1}{2}\sum_{k=1}^{K}(y_k - \hat{y}_k)^2 \\ &= \frac{1}{2}\sum_{k=1}^{K}\left(y_k - g_k\left(\sum_{m=1}^{M} w_{2mk} z_m\right)\right)^2 \\ &= \frac{1}{2}\sum_{k=1}^{K}\left(y_k - g\left(\sum_{m=1}^{M} w_{2mk} f_m\left(\sum_{j=1}^{p} w_{1jm} x_j\right)\right)\right)^2 \end{aligned} \tag{2.8}$$

over each of the training observations. This is a continuous differentiable function of every weight which means that we can use gradient descent to minimise this error function with respect to the weights in the network.

The gradient descent algorithm states that we start at an initial estimate of the weights and then move some distance in the direction of the negative gradient to obtain a new estimate of the weights in the neural network. This procedure is continued until there is little change in the weights from iteration to iteration. This means that the weight update rule for the gradient descent algorithm is:

$$w_{xyz}^{(t+1)} = w_{xyz}^{(t)} - \alpha\frac{\partial E}{\partial w_{xyz}} \tag{2.9}$$

where x specifies the number of the layer in which the weight occurs, y and z is the input and output respectively which the weight connects, t is the iteration number and α is the learning rate.

For hidden to output units, we have:

$$\begin{aligned}
\Delta w_{2mk} &= -\alpha \frac{\partial E}{\partial w_{2mk}} \\
&= -\alpha \frac{\partial E}{\partial y_{in-k}} \frac{\partial y_{in-k}}{\partial w_{2mk}} \\
&= \alpha (y_k - \hat{y}_k) g_k'(y_{in-k}) z_m \\
&= \alpha \delta_{2k} z_m \qquad \text{where } \delta_{2k} = g_k'(y_{in-k})(y_k - \hat{y}_k)
\end{aligned} \tag{2.10}$$

For the input to hidden layer weights, we must differentiate with respect to w_{1jm}, by again using the chain rule, we obtain:

$$\begin{aligned}
\Delta w_{1jm} &= -\alpha \frac{\partial E}{\partial w_{1jm}} \\
&= -\alpha \frac{\partial E}{\partial z_m} \frac{\partial z_m}{\partial w_{1jm}} \\
&= \alpha \sum_{k=1}^{K} (y_k - \hat{y}_k) g_k'(y_{in-k}) w_{2mk} \cdot f_m'(z_{in-m}) x_j \\
&= \alpha \sum_{k=1}^{K} \delta_{2k} w_{2mk} f_m'(z_{in-m}) x_j \\
&= \alpha \delta_{1m} x_j \qquad \text{where } \delta_{1m} = f_m'(z_{in-m}) \sum_{k=1}^{K} w_{2mk} \delta_{2k}
\end{aligned} \tag{2.11}$$

We see that the update equations for the weights connecting the input layer to the hidden layer (2.11) is of the same form as the update equation of the weights that connect the hidden layer to the output layer (eq 2.10). The general form of this equation is the learning rate times the delta, which is calculated at the node to which the particular weight connects to, times the activation of the input node from which the particular weight connects. The weight update equations basically propagate the errors backwards through the network, to determine the suitable weights. This is where the term backpropagation comes from. This result can be extended to derive weight update equations for networks

with any number of hidden layers simply by further application of the backpropagation rule. More detail on how this can be done and also on the backpropagation algorithm just derived can be found in Fausett (1994); Beale and Jackson (1990); Hertz et al. (1991). Now that we have derived the weight update rules, we implement the backpropagation algorithm as follows:

1. Set initial values on the weights (small random numbers can be used).

   Feedforward phase:

2. For each training observation $(\mathbf{x}, \mathbf{y})$, calculate the predicted values from the network:

$$\hat{y}_k = g_k\left(\sum_{m=0}^{M} w_{2mk}\, f_m\left(\sum_{j=0}^{p} w_{1jm}\, x_j\right)\right) \quad k = 1, \ldots, K$$

   Backpropagation of error:

3. For each output node $(y_k, k = 1, \ldots, K)$ in the output layer calculate the error information term, also called the delta for that node:

$$\delta_{2k} = (y_k - \hat{y}_k) g_k'(y_{in-k})$$

   The change in weights for weights connecting to the output layer is:

$$\Delta w_{2km} = \alpha \delta_{2k} z_m$$

   Each node in the hidden layer $(z_m, m = 1, \ldots, M)$, sums the delta inputs (from the units in the output layer):

$$\delta_{in-m} = \sum_{k=1}^{K} \delta_{2k}\, w_{2mk}$$

The delta for the hidden nodes is then:

$$\delta_{1m} = \delta_{in-m} f'_m(z_{in-m})$$

The change in weights for weights connecting to the hidden layer is:

$$\Delta w_{1jm} = \alpha \, \delta_{1m} \, x_p$$

Update the weights:

4. The update rule for the weights connecting to the output layer is:

$$w_{2mk}^{(new)} = w_{2mk}^{(old)} + \Delta w_{2mk}$$

The weights connecting to the hidden layer is updated by:

$$w_{1jm}^{(new)} = w_{1jm}^{(old)} + \Delta w_{1jm}$$

5. Continue this process until the weights converge or another suitable stopping criterion is satisfied.

The algorithm above updates the weights after each observation is presented to the network. Batch learning can also be used in backpropagation by accumulating the weight updates for an entire training epoch. If online learning is used to update the weights, it allows for wider examination of the surface of the error function which makes the training rule stochastic. The choice of which of the two ways of learning should be used varies between different problems but in practice it is found that online learning seems to perform better (Hertz et al., 1991, p.119).

Backpropagation enhancements

The backpropagation training rule has been the topic of a lot of research in past years in the field of neural networks and the algorithm illustrates the method of training neural networks clearly but it is often criticised that it is slow to converge. This is largely due to the nature of the error surface. Error surfaces are generally full of flat spots, steep regions and local minima and these characteristics are even more pronounced when dealing with classification problems or small samples. Many enhancements to the backpropagation method of training a neural network have been suggested to improve the speed of convergence, avoid that the algorithm converges to a local minimum and to estimate the weights in the neural network such that it gives the best possible generalisation capability to unseen data. Only an overview of some of the methods to improve the speed of convergence for the backpropagation training rule will be given.

The learning rate that is used in the backpropagation algorithm determines how fast the network will converge to a solution. The problem with a constant learning rate, is that if the learning rate is chosen to very small, the error function should decrease at each iteration, but convergence will be very slow since a large number of iterations will be needed to reach the minimum.

On the other hand, if the learning rate is chosen to be large, convergence will often be quick initially but the algorithm may overshoot the minimum and end up oscillating between values. This is especially true if the minimum is located in a deep valley with steep sides. This will cause the algorithm to oscillate from the one side of the minimum to the other side and the minimum may never be reached. This point is illustrated in figure 2.11.

Different values of the learning rate α is best suited to different regions of the error

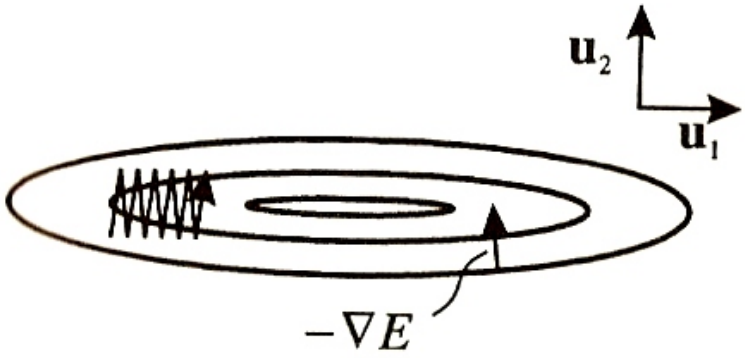


Figure 2.11: Illustration of a three dimensional error surface as viewed from the top. We see that the gradients along the different directions in this figure differs drastically. The single arrow shows that for most points on the error surface, the negative gradient at that point does not point directly to the minimum. On the left hand side, it is shown that gradient descent can take a long time to converge to the minimum. In this particular case, the fixed step gradient descent oscillates across the error surface (Bishop, 1995, p.265).

surface and this shows that the algorithm will generally benefit from a learning rate that begins with a large value, with the value becoming smaller as the algorithm approaches a minimum. This is where adaptive learning rates can help and increase the rate of convergence of the backpropagation algorithm. The easiest way to include an adaptive learning rate, which will also ensure that the algorithm converges even though it may not be the optimal solution, is to choose the learning rate as

$$\alpha = \frac{1}{t}$$

where t denotes the iteration number. This implies that the learning rate will become smaller with each step and eventually the learning rate will so small that the weights will change very little from iteration to iteration.

A more effective way of choosing the learning rate is to make it part of the training process. This consists of checking whether a specific weight update did indeed reduce the

error. If the error increased in a specific step, it means that the step that was taken was too large and overshot the minimum implying that the learning rate should be reduced. On the other hand, if several steps in a row have lead to a decrease in the error function, it means that the learning rate is too small and that the algorithm will take long to converge implying that the learning rate could be increased to be more effective. Hertz et al. (1991) states that it is better to increase the learning rate by adding a constant value and decrease the learning rate geometrically. This will enable the learning rate to become smaller rapidly if needed to. This means that the learning rate is chosen as follows:

$$\alpha_{new} = \begin{cases} \alpha_{old} + a & \text{if } \Delta E < 0 \text{ consistently} \\ b\alpha_{old} & \text{if } \Delta E > 0 \end{cases}$$

where a and b are appropriate constants. More information on choosing adaptive learning parameters can be found in (Hassoun, 1995; Hertz et al., 1991).

A modification to increase performance in the backpropagation learning rule is to include a momentum term to the weight updates. This is a very effective way to increase the speed of learning especially when part of the training data differs a lot from the majority of the training data. Momentum causes the current weight update, to be a combination of the current gradient and the gradient at the previous update. The reason for this is that it is desirable when an unusual training observation is presented to the network, that the weights should not be influenced as much. On the other hand we want the network to learn the weights fairly quickly when it is presented with training observations that are relatively similar to each other.

By adding a momentum term to the weight updates, the learning rate of the algorithm slows down when an unusual training observation is presented to the network. This means that the rate of convergence is often accelerated when a momentum term is added to the weight updates because the algorithm keeps learning in the average downhill direction

making the updates resistant to wild oscillations. The modified gradient descent formula which includes momentum is given by

$$\Delta \mathbf{w}^{(t)} = \alpha \, \nabla E|_{\mathbf{w}^{(t)}} + \eta \Delta \mathbf{w}^{(t-1)}$$

where $0 < \eta < 1$ is called the momentum rate. This formula shows that the weight update now becomes a combination of the update at the current iteration and the update that was used at the previous iteration (Bishop, 1995). This concept is illustrated in figure 2.12.

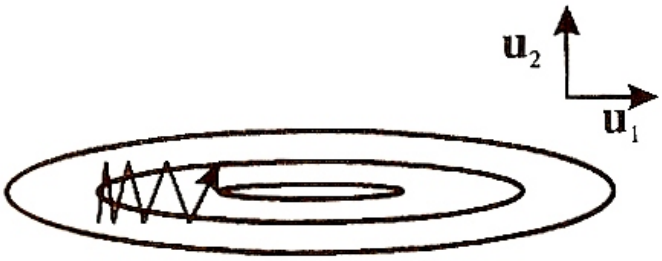


Figure 2.12: Illustrates the use of a momentum term when using gradient descent. We see now that the steps move in the average downhill direction, which means that the minimum is reached in less steps than without the momentum term (Bishop, 1995, p.269).

This is one example of implementing momentum in the weight updates. There are also other variations like adaptive momentum rates, one of which is called *quickprop*, but these will not be discussed further here and the interested reader is referred to Fahlman (1988); Hassoun (1995); Waugh and Adams (1997).

The backpropagation rule is very sensitive to the initial values assigned to the weights as is shown in Kolen and Pollack (1991). This is due to the gradient descent nature of the algorithm. If the initial weight vectors are chosen such that the algorithm starts in the vicinity of a local minimum with steep sides, the weights will quickly converge and how good the solution is will depend on how deep the local minimum is relative to the global minimum. On the other hand, if the algorithm starts on a flat part of the error surface,

convergence can be slow. The initial weights are often chosen to be small zero-mean random values which is found to work well in practice.

The backpropagation rule is not the only method for estimating the weights in a neural network and many other methods and techniques have been developed. An overview of some of these techniques will be given in the next section.

Other estimation methods

Gradient descent is easy to understand and although it is still widely used in neural network training today, a lot of research has been done on implementing more efficient algorithms for network training. Optimisation methods can be categorised according to the type of information that they use and are generally be classified into three groups (Fiesler and Beale, 1997):

(i) Search methods: These methods employs a search technique across the whole weight space. The error function is evaluated at different points in the weight space and the point which gives the lowest value of the error function is then used. This technique is not used a a lot as they are very slow but they can be used to supplement other methods, for example they can be used to obtain initial values to be used with the other methods. Search methods will not be discussed any further.

(ii) First-derivative methods: These methods use first-derivative (gradient) information of the error function to make weight adjustments. Gradient descent is a very good example of a first derivative method and other examples are conjugate gradient descent and quasi-Newton descent. The basic idea of these methods is to use gradient information to compute a weight adjustment in a direction that will lead to a decrease in the value of the error function. They are generally fast and do not need a

lot of information to be computed at each weight adjustment.

(iii) Second-derivative methods: These methods use first and second order derivative information of the error function. They are computationally very expensive since these methods need to calculate the second-order derivative matrix (Hessian matrix) at each weight adjustment, but they will generally be the fastest to converge to a minimum.

We will not go into all the mathematical detail of these methods but will rather aim to provide a brief overview of them. As was stated, the area of network training is a major area of research in neural networks and a list of references where more information on these methods can be obtained will be given at the end of this section.

We will start with first derivative methods which only make use of the first order derivatives of the error function. Backpropagation, which was discussed in section 2.3.5 is the most widely used first derivative method but the concept can be enhanced to form the foundation of more sophisticated first derivative methods. Recall from section 2.3.5 where it was stated that network training is an exploration of the error surface in which we start at some point on the error surface, pick an appropriate direction to move in and then move some distance in that direction. For the backpropagation rule it was shown (cf. Section 2.3.5) that the direction which is moved in, is the negative gradient of the error function and the distance is determined by the learning parameter.

A better procedure might be to move in the direction of the negative gradient and then find the point in that direction that will minimise the error function. This procedure is called line search and forms the basis of several methods which are more powerful than gradient descent. Minimisation of the error function along a search direction is a one dimensional minimisation problem and approaches on how to implement this can be found in Bishop (1995).

By proceeding to move in the direction of the negative gradient after minimisation along the initial negative gradient direction may not be the optimal choice. This may cause the algorithm to oscillate on successive steps and can cause the algorithm to take long to converge. A better approach to selecting the direction after minimisation along the initial direction is called conjugate gradient descent or non-interfering directions. The concept of conjugate gradients is that once minimisation was done in some direction, a minimisation in another direction may spoil this other minimisation. Now if the directions are non-interfering and linearly independent, this will reduce the need for multiple minimisations in the same direction and will converge to a solution in fewer steps. This is the method of conjugate gradient descent and more information can be found in Bishop (1995); Battiti (1992).

We now move on to give a quick discussion of second derivative methods. These methods can be further subdivided into Newton methods and secant methods. Second order methods uses local quadratic estimation of the error function around some point by use of a Taylor series expansion. Consider a Taylor expansion of $E(\underline{w})$ up to second order:

$$E(\underline{w}) \approx E(\underline{w}') + (\underline{w} - \underline{w})'\underline{b} + \frac{1}{2}(\underline{w} - \underline{w}')'\mathbf{H}(\underline{w} - \underline{w}') \tag{2.12}$$

where $\underline{b}$ is defined to be the gradient of E evaluated at $\underline{w}'$

$$\underline{b} = \left.\frac{\partial E(\underline{w})}{\partial \underline{w}}\right|_{\underline{w}=\underline{w}'} \tag{2.13}$$

and $\mathbf{H}$ is the second order derivative matrix (Hessian) defined by

$$(\mathbf{H})_{ij} = \left.\frac{\partial^2 E}{\partial w_i \partial w_j}\right|_{\underline{w}=\underline{w}'} \tag{2.14}$$

The derivative of (2.12) with respect to $\underline{w}$ is given by

$$\frac{\partial E(\underline{w})}{\partial \underline{w}} = \underline{b} + \mathbf{H}(\underline{w} - \underline{w}') \tag{2.15}$$

and if we set this equal to 0 we obtain:

$$\frac{\partial E(\underline{w})}{\partial \underline{w}} = \underline{b} + \mathbf{H}(\underline{w} - \underline{w}') \stackrel{set}{=} 0 \tag{2.16}$$

$$\Rightarrow \underline{w} = \underline{w}' - \mathbf{H}^{-1}\underline{b} \tag{2.17}$$

This forms the basis of the second derivative methods. The second derivative methods make explicit use of the Hessian matrix. The term $-\mathbf{H}^{-1}\underline{b}$ in expression (2.17) is known as the Newton direction or the Newton step. Since (2.12) is a quadratic approximation to the error surface, (2.17) should be applied iteratively. This will give rise to the following weight update formula:

$$\underline{w}^{\text{new}} = \underline{w}^{\text{old}} - \mathbf{H}^{-1}\underline{b} \tag{2.18}$$

where the gradient vector $\underline{b}$ and the Hessian matrix $\mathbf{H}$ is evaluated at the current point in the weight space $\underline{w}^{\text{old}}$. Notice that this algorithm also takes on the form of starting with an initial value on the error surface and then taking a step in a suitable direction that will expectantly decrease the error function. For a quadratic error surface, the Newton step will always point to the minimum of the error surface (Bishop, 1995). The major disadvantage of the Newton method is that the Hessian matrix $\mathbf{H}$ has to be re-evaluated and inverted at every iteration and thus for networks with a large number of weights. This makes the method computationally very intensive. Other disadvantages of the Newton method is that it may converge to a saddle point or a local maximum. This will typically happen if the Hessian is not positive definite. The Newton method can however be modified to become a practical method of optimisation in neural networks (Bishop, 1995).

Quasi-Newton methods fundamentally work in the same way as the Newton-method in (2.17), but instead of calculating the Hessian directly, it iteratively builds up a good approximation to the inverse of the Hessian matrix. By approximating the Hessian the method is not computationally as expensive as the Newton method. More information on using quasi-Newton methods for training neural networks can be obtained in Battiti (1992) and Bishop (1995).

All the methods that we have considered up to now followed a search direction. A *model-trust region* approach is another way of training neural networks. The *model-trust region* assumes that the error surface is some well-behaved shape (for example a parabola) in the vicinity of the current point. The Levenberg-Marquardt algorithm is an example of a *model-trust region* approach in which it is assumed that the error surface is parabolic in the region of the current point. The Levenberg-Marquardt algorithm is generally the fastest of all the training algorithms for neural networks but it has severe limitations, one of them being that it can only be employed on networks with a single output, that is trained by the sum of squares criterion (Hill and Lewicki, 2006). Battiti (1992) and Bishop (1995) gives more information on the Levenberg-Marquardt algorithm and how it can be implemented.

2.4 Neural networks as statistical modelling tools

2.4.1 Comparisons between neural networks and statistics

There have been a lot of dispute in the recent years whether neural networks are indeed intelligent models or if they are just generalisations of statistical models. There is considerable overlap between the fields of neural networks and statistics and for many neural networks models there is an equivalent or similar statistical technique. Examples of neural networks with their statistical equivalents are (Sarle, 1997):

- Single layer feedforward neural networks are closely related to generalised linear models.
- Multilayer perceptrons with one hidden layer are closely related to projection pursuit regression.
- Probabilistic neural networks is the same as kernel discriminant analysis.
- Kohonen networks are very similar to k-means cluster analysis.

These are not the only neural networks models that are closely related to statistical models and a comprehensive list of neural networks with a competing statistical method can be found in Sarle (1997).

One of the major claims from users of neural networks is that neural networks require no distributional assumptions about the data. Bishop (1995) showed that neural networks involve the same distributional assumptions as statistical models, but these assumptions are often ignored by persons using neural networks. In contrast to this, statisticians study the importance of the distribution assumptions and also the consequences leading from

whether the assumptions are satisfied or not. Studying the distributional assumptions is obviously a very important aspect of modelling and this can be the difference between a model that fits and generalises well to unseen data and using an inappropriate model for the training data at hand. An example of a very important assumption which is almost never checked by users of neural networks is that of whether the error terms have a constant variance.

Neural networks are characterised by the algorithm that was used to train the network, for example the backpropagation network, because the weights in the network are estimated using backpropagation. The criterion that is used to fit the model is not important for people coming from the field of neural networks. Statisticians however consider the different training algorithms just as different ways of implementing the same model or to be more correct, different ways to estimate the parameters in the regression model. However if a different criterion is used to fit the model, statisticians view this as a different estimation method with different statistical properties (Sarle, 1997).

In section 2.3.5 it was shown that minimisation of the sum of squares for error function is done by using a numerical optimisation algorithm, the most often used being the gradient descent method leading to the backpropagation neural network. Many of the methods that are often used to fit nonlinear regression models like the Levenberg-Marquardt and conjugate gradient algorithms can also be used to estimate the weights in a feedforward neural network.

Regression and classification problems where both independent and dependent or target variables are present in the training data are called supervised learning in the neural network literature. Unsupervised training occurs where only independent or feature variables are present with no dependent variables and the objective is to explain how the data is organised or clustered. A well known technique for unsupervised training in statistics is

cluster analysis while Kohonen networks and adaptive resonance theory (ART) are examples of unsupervised neural networks. In sections 2.4.2 and 2.4.3 we will show how regression can be expressed as a neural network and how neural networks can be used for classification and regression problems encountered in statistics.

2.4.2 The regression problem

Regression is described as the problem of modelling a continuous dependent variable as a function of continuous, and possibly categorical independent variables, where the categorical independent variables can be coded as dummy variables. This means that regression defines a function that maps all the inputs of an observation, x_i, $i = 1, \ldots, p$ to an output y, where this output is a continuous variable. This relationship between the dependent and independent variables are modelled as a mathematical function, with a number of adaptive parameters which needs to be estimated from the training data. A regression model can be written in the form

$$\hat{y} = f(\underline{x}, \underline{w}) \tag{2.19}$$

where $\underline{w}$ denotes the parameters that needs to be estimated.

In statistics, linear and nonlinear regression are most often used for this purpose. When using parametric regression, the functional form $f(\cdot)$ is determined beforehand and the parameters of the chosen model is estimated such that the model fits the training data well and generalises well to unseen data. This model enables us to obtain a predicted value for given values or levels of the independent variables.

A neural network can be considered as a particular choice of the mathematical function $f(\cdot)$ in (2.24). Neural networks provide a very general basis for representing nonlinear

mappings from input to output variables and can be used to approximate the regression function.

Suppose that a response Y is given by the value of a deterministic function, say $h(\cdot)$, of the input variables $\underline{X}$ with an added error term which follows a normal distribution with zero mean and constant but unknown variance σ^2. This can be written as:

$$Y = h(\underline{X}) + \varepsilon \qquad \text{where } \varepsilon \sim N(0, \sigma^2)$$

We now seek to approximate the function $h(\cdot)$ by a neural network function of the form $f(\underline{X}, \underline{w})$ where $\underline{w}$ is a set of weights. This set of weights in the neural network is estimated from a training sample data set.

Bishop (1995, pp.195–197) shows that maximum likelihood estimates of the weights $\underline{w}$ under the assumption that the error term is normally distributed with zero mean and constant variance corresponds to minimising the error sum of squares function

$$E = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

or if an observation consists of more than one response variable, the sum of squares for error function takes the sum over all the outputs of the network

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{K}(\hat{y}_{ik} - y_{ik})^2$$

This is an important result which motivates the use of the sum of squares for error function for training a neural network which will be used for regression. Furthermore Bishop (1995, pp.201–206) shows that for a sufficiently general neural network, which is trained by minimisation of the sum of squares function, the outputs from the neural network approximates the conditional average of the response variable for given values of

the inputs $\underline{x}$. This can be written as

$$\hat{y} = f(\underline{x}, \underline{w}) = E(y|\underline{x})$$

Sufficiently general in this context means that the neural network must be complex enough to approximate the regression function closely. This result is of practical importance:

> The importance of neural networks is that they provide a practical framework for approximating arbitrary nonlinear multivariate mappings and can therefore in principle approximate the conditional average to arbitrary accuracy (Bishop, 1995).

Regression as a neural network

We will now show how a simple regression model can be written as a neural network. The linear regression model in statistics is defined as:

$$\hat{y} = f(\underline{x}, \underline{\hat{\beta}}) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p \tag{2.20}$$

This can very easily be illustrated graphically as a single layer neural network (figure 2.13). The activation function used in this neural network is the linear activation function given by $f(x) = x$. This need not be the case and a single layer layer network can also be extended to accommodate nonlinear regression by choosing a nonlinear activation function. One example of this is the generalised linear model in statistics, which is defined as:

$$h(y) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p \tag{2.21}$$

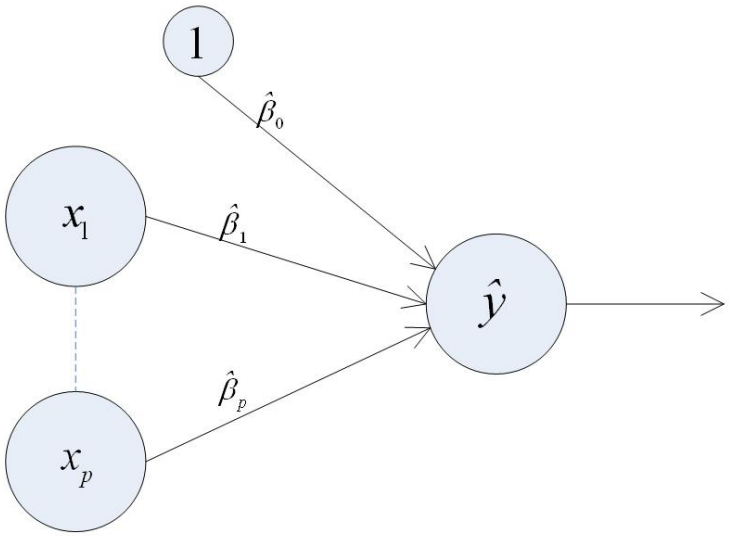


Figure 2.13: Linear regression as a single layer neural network

where $h(.)$ is called the link function. There are various link functions which can be used. The choice of link function depends on how the data is distributed. One example of a link function is the *logit* function defined by $h(y) = \frac{y}{1-y}$. For more information on generalised linear models consult McCullagh and Nelder (1983).

If we write (2.21) as:

$$\hat{y} = h^{-1}(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p) \tag{2.22}$$

we see that we can also express the generalised linear model as a single layer neural network with activation function equal to the inverse of the link function (Warner and Misra, 1996).

This shows that simple linear and nonlinear regression models can be expressed as single layer neural networks by using the appropriate activation function. The main difference between neural network models and statistical models is in the way the parameters are estimated. Regression models are most often fitted by using the sum of squares criterion.

Other error functions can also be used - for example the root-mean square error. The sum of squares error is motivated by the principle of maximum likelihood for normally distributed response data. Generalised linear models are also fitted by using the principle of maximum likelihood for a variety of distributions which are members of the exponential class (Sarle, 1994). These lead to likelihood functions which are nonlinear in the parameters and a numerical optimisation technique like Newton-Raphson can be used to estimate the parameters.

The weights in a multilayer perceptron are estimated by directly minimising the chosen error function with some numerical optimisation algorithm. The function that is most often used for this purpose in regression problems is the sum of squares error function. In section 2.4.2 it was stated that the use of the sum of squares error function can be motivated from the principle of maximum likelihood when the error term is distributed as a normal distribution with a zero mean and constant variance. This means that the sum of squares error function is most appropriate when the conditional distribution of the target variable, given the input variables, are normally distributed, typically with a constant but unknown variance and that each of the training cases is independent. This is very similar to the linear regression model. However, since we mostly use neural networks for prediction and not for statistical inference, the sum of squares error function can be, and is often, used for training any neural network that is going to be used for regression. Therefore, the assumption of normality of the error terms can be relaxed and we only need the error terms to have a distribution which is symmetric around zero. This point will be further discussed in section 2.4.4.

2.4.3 Neural networks for classification

Classification is the task of assigning observations to a number of distinct categories or classes. Classification in a statistical pattern recognition context can have two distinct meanings. In the unsupervised learning case, the observations in the data set only contain independent variables, hence there is no label to know to which true class the observation belong. When this is the case, the aim is to establish whether there are any classes or clusters in the training data. In the supervised learning case, classification means that we are given measurements on continuous (and possibly categorical) independent variables and we know for certain to which group these inputs belong. The aim is then to derive a method or rule which can be used to classify an observation that does not form part of the training data into a group. The training data contains both dependent and independent variables, where the dependent variable is often a group label (Michie et al., 1994, p.6). The supervised training case will be discussed in this section and for the rest of the dissertation we will refer to supervised training for statistical classification just as classification while unsupervised training means clustering of data. In statistics, classification is known as discrimination. Examples of applications of classification is:

- Classify an applicant for credit as being a good or bad risk. The classification can be based on variables such as age, income and marital status and these types of applications are found in credit scoring.
- Identify handwritten postal codes on an envelope. The objective is to scan each number and then use information available on the pixels in the scanned image to classify the digit as a number ranging from 0 to 9. This is an example of pattern recognition.
- In the medical field, a diagnosis of a disease can be made based solely on symptoms and other measures, like blood pressure, ldl and hdl levels of the patient.

The classification problem in general

In a statistical context, the problem of classifying an input pattern to a specific class is often termed pattern recognition. In this section we will give a general approach to classification from a statistical point of view and this will help to give a clear understanding of how neural networks can be used for classification. A more thorough approach to statistical methods for classification can be found in Hand (1981).

It was stated earlier that the aim of classification is to assign observations to one of a set of discrete classes C_k, $k = 1, \ldots, K$. Classification can also be seen as problem of defining a function, to map an input pattern $\underline{x}$ in the data set, to an output y, where y specifies the class to which the particular inputs belong to (Fiesler and Beale, 1997).

Bayesian decision theory forms the foundation of statistical classification methods such as discriminant analysis (Zhang, 2000). An overview of Bayesian decision theory will be given which follows in a similar way as in Bishop (1995, pp.17–27). The most general description for statistical classification using a Bayesian framework is in terms of the probabilities that a given input pattern, say $\underline{x} = (x_1, \ldots, x_p)'$, belongs to each of the possible classes C_k. These probabilities are called the posterior probabilities and is written as $P(C_k|\underline{x})$, meaning that it is the probability of belonging to class C_k, given input pattern $\underline{x}$.

The decision to which class an input pattern $\underline{x}$ should belong to can then be argued as follows:

The probability of a classification error is:

$$\begin{aligned} P(Error|\underline{x}) &= \sum_{i \neq k} P(C_i|\underline{x}) \\ &= 1 - P(C_k|\underline{x}) \quad \text{if we assign to class } C_k \end{aligned}$$

If the objective is to minimise the misclassification rate, this then leads to the Bayesian classification rule:

$$\text{Decide } C_k \text{ for } \underline{x} \text{ if } P(C_k|\underline{x}) = \max_{i=1,\ldots,K} P(C_k|\underline{x})$$

This means that an input pattern $\underline{x}$ is assigned to the class with the largest posterior probability $P(C_k|\underline{x})$.

From Bayes' theorem, it follows that

$$P(C_k|\underline{x}) = \frac{p(\underline{x}|C_k)P(C_k)}{p(\underline{x})} \tag{2.23}$$

where $p(\underline{x}|C_k)$ is known as the class conditional density, $P(C_k)$ is known as the prior probability and $p(\underline{x})$ is the unconditional density of the inputs. The class conditional density $p(\underline{x}|C_k)$ can roughly be stated as the probability of observing an input pattern $\underline{x}$ given that the pattern belongs to class C_k. The prior probability is the probability of class membership before any observation is made on an input pattern. The denominator in (2.23) can be written as:

$$p(\underline{x}) = \sum_k p(\underline{x}|C_k)P(C_k)$$

and this term normalises the posterior probability, ensuring that all the posterior probabilities sum to one, that is $\sum_{k=1}^{K} P(C_k|\underline{x}) = 1$.

An input pattern is assigned to the class which minimises the probability of misclassification. Thus an input pattern $\underline{x}$ is assigned to class C_k if

$$P(C_k|\underline{x}) > P(C_i|\underline{x}) \qquad \text{for all } i \neq k \tag{2.24}$$

By using (2.23), and since the denominator is independent of the class C_k, (2.24) is

equivalent to

$$p(\underline{x}|C_k)P(C_k) > p(\underline{x}|C_i)P(C_i) \qquad \text{for all } i \neq k \tag{2.25}$$

A pattern classifier provides a method which enables us to assign any point in the input feature space, say $\underline{x}$, to one of K distinct classes. We can consider this as the feature space being segmented into K decision regions, denoted by $R_1, \ldots, R_K$. An input feature that falls in the region R_k is then assigned to class C_k. It should be noted that a decision region does not necessarily have to be a closed region and that one decision region can be made up of several disjoint decision regions which are all related with the same class. The boundaries between these regions are called decision boundaries. Figure 2.14 illustrates different decision regions that are possible in a two-dimensional input space.

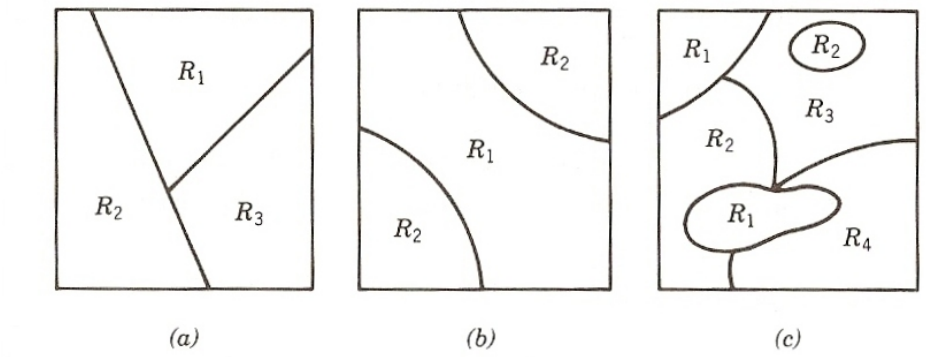


Figure 2.14: Examples of possible decision regions (Schalkhoff, 1992, p.16)

Consider an example where we have two classes, C_1 and C_2, and the input vector is two dimensional i.e. $\underline{x} = (x_1, x_2)$. The objective is to construct a decision boundary, or decision regions R_1 and R_2, which will minimise the probability of misclassification. A misclassification error will occur if we assign an input vector which belongs to class 1 to class 2 or vice versa for an input vector which belongs to class 2. The probability of an

error can then be written as (Duda and Hart, 1973):

$$P(error) = P(\underline{x} \in R_2, C_1) + P(\underline{x} \in R_1, C_2) \tag{2.26}$$

$$= P(\underline{x} \in R_2|C_1)P(C_1) + P(\underline{x} \in R_1|C_2)P(C_2) \tag{2.27}$$

$$= \int_{R_2} p(\underline{x}|C_1)P(C_1)d\underline{x} + \int_{R_1} p(\underline{x}|C_2)P(C_2)d\underline{x} \tag{2.28}$$

where $P(\underline{x} \in R_2, C_1)$ is the probability that the input $\underline{x}$ is assigned to class C_2 when the true class of the input is C_1.

If, for a given value of the input vector, we have that $P(C_1|\underline{x}) > P(C_2|\underline{x})$, then from (2.25), it follows that $p(\underline{x}|C_1)P(C_1) > p(\underline{x}|C_2)P(C_2)$. From (2.26) we see that we need to choose the regions R_1 and R_2 in such a way that the input pattern $\underline{x}$ is in R_1 because this will result in the smallest probability of a misclassification error. This is equivalent to (2.24) in which a pattern is classified to the class with the largest posterior probability of membership. This result is represented graphically in figure 2.15. In this figure it is illustrated that for a two class problem, the probability of misclassification is minimised by placing the decision boundary where the arrow is. This corresponds to where the densities cross. If the vertical line is used for the decision boundary, we see that the probability of misclassification becomes larger. This classification rule can also be generalised to the case where a p-dimensional input vector $\underline{x} = (x_1, \ldots, x_p)$ needs to be classified to one of K classes, $C_1, \ldots, C_K$.

The simple classification rule in (2.24) is the basis of many statistical methods for classification. Linear and quadratic discriminant analysis assumes that the class conditional densities are multivariate normal distributions, with assumed equal or unequal covariance matrices respectively. Two things should be noted about the simple Bayes classification rule in (2.24). Firstly the class conditional density function needs to be estimated in order to calculate the posterior probabilities. This can be done in a parametric or nonparamet-

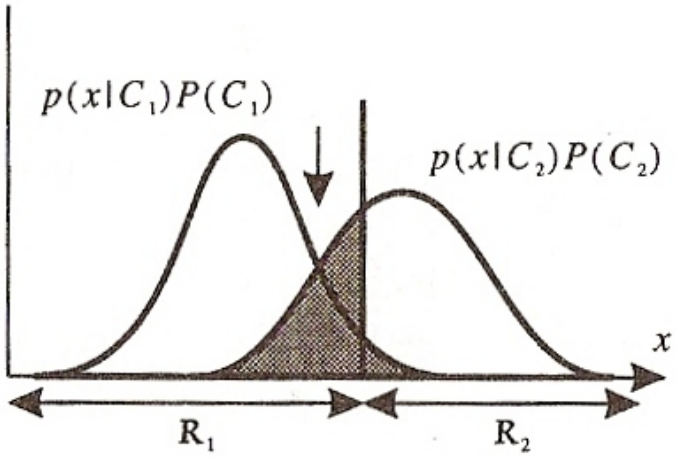


Figure 2.15: Illustration of the optimal classification rule. (Bishop, 1995, p.25)

ric way and the interested reader is referred to Michie et al. (1994); Schalkhoff (1992) for methods of density estimation. Secondly, the classification is done only to minimise the probability of misclassification and this may not always be the most suitable criterion since no provision is made for the different consequences associated with misclassification. Misclassification of observations from a certain class may be seen as more serious as misclassifications from other classes. An example of this is when a patient needs to be diagnosed as either having a particular disease or not having the disease. It is much more serious to classify a patient as not having the disease when the patient in fact has the disease then it is other way around. This is an example where the cost of misclassification of a patient that does have the particular disease is very high compared against the alternative classification. This cost of misclassification should then also be taken into account when a classifier is constructed since this will improve the decision made.

An easy way to introduce the costs of misclassification is by using a loss matrix, say L, where the elements L_{kj} specifies the cost of misclassifying an observation from group C_k

to C_j. The expected cost or loss of classifying an input vector $\underline{x}$ to group j is:

$$L_j(\underline{x}) = \sum_{k=1}^{K} L_{kj} P(C_k|\underline{x}) \qquad j = 1, \ldots, K$$

The function $L_j(\underline{x})$ is known as the conditional risk function (Zhang, 2000). Following a similar approach as for the derivation of the simple Bayes classification rule, the objective is to minimise the overall expected cost:

$$\text{Decide } C_k \text{ for } \underline{x} \text{ if } L_k(\underline{x}) = \min_{j=1,\ldots,K} L_j(\underline{x})$$

It can be shown that the optimal decision rule will be to classify an input vector to the class C_j when

$$\sum_{k=1}^{K} L_{kj} p(\underline{x}|C_k) P(C_k) < \sum_{k=1}^{K} L_{ki} p(\underline{x}|C_i) P(C_i) \quad \text{for all } i \neq j$$

An advantage of using posterior probabilities for classification is that a rejection criterion can be introduced. In general when the posterior probabilities are all relatively low or when the largest posterior probabilities are relatively similar it indicates that an observation cannot be assigned with enough certainty to a class. This means that there is a strong overlap of classes in this region and this is where we expect most of the misclassification errors to occur. If this happens, it may be better to not make a classification and rather make use of manual classification, for example a human expert can do the classification of that observation manually. If we introduce a rejection threshold this leads to the following classification rule (Bishop, 1995, p.28):

$$\text{if } \max_k P(C_k|\underline{x}) \begin{cases} \geq \theta & \text{then classify } \underline{x} \text{ to } C_k \\ < \theta & \text{then reject } \underline{x} \end{cases}$$

Modelling discriminant functions Up to this stage, the classification task have been based on modelling posterior probabilities. This involves assigning an observation to the class with the largest posterior probability of class membership. It was also shown how Bayes' theorem could be used to relate this posterior probability to class-conditional densities. These densities need to be estimated. An alternative method, which does not involve estimating the class conditional densities, can be used in which we reformulate the classification task in terms of a set of discriminant functions $y_1(\underline{x}), \ldots, y_K(\underline{x})$. A specific parameterised functional form for the discriminant function is chosen and then the parameters function is estimated from a training sample.

The way in which discriminant functions are used for classification is to classify an input vector $\underline{x}$ to class C_k if

$$y_k(\underline{x}) > y_i(\underline{x}) \qquad \text{for all } i \neq k$$

The decision boundaries can be seen as the areas where the discriminant functions are equal. For example, if regions R_k and R_i are neighboring regions, then the boundary which separates these two regions are given by the vectors in the input space where

$$y_k(\underline{x}) = y_i(\underline{x})$$

The decision rule which is based on minimising the posterior probability of misclassification as in 2.24 can be easily rewritten in terms of discriminant functions by choosing

$$y_k(\underline{x}) = P(C_k|\underline{x})$$

By using Bayes' theorem 2.23, the discriminant function can be written as

$$y_k(\underline{x}) = \frac{p(\underline{x}|C_k).P(C_k)}{p(\underline{x})}$$

This can equivalently be written as

$$y_k(\underline{x}) = p(\underline{x}|C_k)P(C_k)$$

since the denominator does not influence the classification. Now, since classification by using discriminant functions are only influenced by the size of the discriminant function, any monotonic function $g(\cdot)$ can be applied to the discriminant function without influencing the classification made. For example, if we take the logarithm of the discriminant function $y_k(\underline{x})$, then the discriminant function can be written as

$$y_k(\underline{x}) = \ln p(\underline{x}|C_k) + \ln P(C_k)$$

Linear and quadratic discriminant analysis assume that the class conditional density function $p(\underline{x}|C_k)$ is multivariate normal with class mean vector $\underline{\mu}_k$ and class covariance matrix Σ_k. The class conditional density for class C_k can be written as:

$$p(\underline{x}|C_k) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} \exp\left\{-\frac{1}{2}(\underline{x} - \underline{\mu}_k)'\Sigma^{-1}(\underline{x} - \underline{\mu}_k)\right\}$$

If the class conditional densities are independent multivariate normal distributions, then the discriminant functions can be written as

$$y_k(\underline{x}) = -\frac{1}{2}(\underline{x} - \underline{\mu})'\Sigma_k^{-1}(\underline{x} - \underline{\mu}_k) - \frac{p}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma_k| + \ln P(C_k)$$

This causes the decision boundaries to be quadratic functions in a p-dimensional space. A simplification occurs when the class covariance matrices are assumed to be equal, in which the decision boundaries reduce to linear functions in the p-dimensional input space. Bishop (1995) shows that when the class covariance matrices are assumed to be equal, i.e. $\Sigma_k = \Sigma \ \forall k$, then the discriminant functions can be written as

$$y_k(\underline{x}) = \underline{w}_k'\underline{x} + w_{k0}$$

where

$$\underline{w}_k' = \underline{\mu}_k' \Sigma^{-1}$$

$$w_{k0} = -\frac{1}{2}\underline{\mu}_k' \Sigma^{-1} \underline{\mu}_k + \ln P(C_k)$$

These discriminant functions are linear in the components $\underline{x}$ and hence the boundaries between the decision regions are also linear. A linear discriminant function is optimal for normally distributed data with equal covariance matrices. More information on linear and quadratic discrimination analysis can be found in Bishop (1995); Johnson and Wichern (2002) and Hand (1981).

Neural networks and the classification problem

Using a neural network for classification can proceed in two ways. A neural network can be set up to represent a nonlinear discriminant function enabling the network to directly provide a classification when presented with an input pattern. The most general way however is to model the probability of class membership for each input pattern $\underline{x}$.

Some of the advantages of using a neural network to approximate the posterior probability of class membership rather than using the network as a discriminant function includes the following (Bishop, 1995, p.223):

Outputs sum to one: Since the outputs of the network can be interpreted as posterior probabilities, they should sum to one. This can help to ascertain whether the network is modelling the posterior probabilities with sufficient accuracy. A check of this is to average the outputs from the neural network for a specific class over all the input observations, and this should be close to the corresponding prior probability, which can be obtained as the proportion of training sample observations which belong to this same class. This

means that:

$$P(C_k) = \int P(C_k|\underline{x})p(\underline{x})d\underline{x} \simeq \frac{1}{n}\sum_i P(C_k|\underline{x}_i)$$

Different prior probabilities Sometimes the prior probabilities that are obtained from the training data differ from what is expected or from the population prior probabilities. The output from a neural network which models posterior probabilities can very easily be adjusted to compensate for this difference in prior probabilities and without having to retrain the neural network. The way in which this is done is by dividing the network outputs by the prior probability which is calculates from the training data and then multiplying this result with the correct prior probability to which you want to adjust to. The results should then be normalised to ensure that the outputs sum to one Bishop (1995, p.223).

Minimise risk As was discussed in section 2.4.3, the optimal classification rule may not necessarily always be the one that minimise the probability of misclassification. This is especially true if there are different costs associated with various misclassifications. The objective is then rather to obtain a classification rule that will minimise the overall cost. The posterior probabilities that are obtained from the neural network can be combined with a loss matrix to arrive at a minimum cost decision. This can also be achieved without having to retrain the original network.

Rejection threshold A rejection threshold can be used to reduce the probability of a classification error as was discussed in section 2.4.3. This is easy to introduce when a neural network is set up to model posterior probabilities and is achieved by only allowing the neural network to make a classification if the posterior probability exceeds a specified

threshold level.

We will now show how a neural network can be set up for classification such that the outputs from the network can be interpreted as posterior probabilities. We will start off with the two-class classification problem, and the first error function that will be used is the well known sum of squares for error. Then an alternative error function, called the cross-entropy, will be introduced. This latter error function will later be extended to classification with more than two classes.

It can be shown that training a neural network by minimising the error sum of squares criterion, results in outputs that approximate the conditional average of the response data (Bishop, 1995, pp.212–220):

$$\hat{y}_k(\underline{x}) = E(y_k|\underline{x}) = \int y_k p(y_k|\underline{x}) dy_k$$

We can use the 1-of-K coding for the response variable, for example, let $K = 4$. This means that we have four classes into which we can classify different input observations. Then, by using the 1-of-K coding scheme for the dependent variable, the dependent variable will be coded as follows:

Category	y_1	y_2	y_3	y_4
Category 1	1	0	0	0
Category 2	0	1	0	0
Category 3	0	0	1	0
Category 4	0	0	0	1

Table 2.1: Example:1-of-4 coding scheme.

Now, if we train a neural network for classification, by minimising the sum of squared error criterion over all the network outputs, where the 1-of-K coding scheme have been used, we obtain

$$\hat{y}_k(\underline{x}) = P(C_k|\underline{x})$$

This implies that these network outputs can be interpreted as Bayesian posterior probabilities (Richard and Lippmann, 1991; White, 1989).

The network outputs $\hat{y}_k, k = 1, \ldots, K$, can be shown to sum to unity since the response variables y_k sum to one for each observation when coded by the 1-of-K coding scheme. Bishop (1995, pp.200–201) shows that the network outputs will satisfy the same linear constraint indicating that the sum of the outputs for each observation will also be one.

The major problem when using the sum of squares error function with classification problems, is that the probabilities are not guaranteed to lie between $(0, 1)$ as they should. It was stated in section 2.4.2, that the sum of squares error function is most appropriate when the conditional distribution of the response variable, $Pr(Y|\underline{X} = \underline{x})$, is assumed to be normal and the estimates of the parameters under this assumption corresponds to the parameters derived from the maximum likelihood principle. It can be seen that the sum of squares error is appropriate for regression problems. For classification problems, the response variables are binary variables and have a Bernoulli distribution. This means that although the sum of squares error function can be used to train a network for classification problems, a more appropriate error function should be used.

We will now show that the most appropriate error function for classification problems are called the cross-entropy error function which corresponds to maximum likelihood when the response data follow a Bernoulli distribution. We will start of with the two class problem and then in section 2.4.3 we will show that this error function can also be extended to allow for more than two classes.

Classification of two classes using neural networks

For the two class problem, an observation should be classified to one of two possible classes. One of two approaches can be followed for this problem. We can train a neural network with two outputs, using the 1-of-C coding scheme for the response variables. This method corresponds to what is done in section 2.4.3 and therefore it will not be discussed now. The other method is to use only one network output $\hat{y}$ and code the response variables as a 1 if the input belongs to class C_1 and a 0 if the input belongs to C_2.

Consider a two-class problem, where we use only one response variable y as was just shown. We would like to construct an error function, such that the output from the network $\hat{y} = f(\underline{x}, \underline{w})$ should represent the posterior probability that the observation is classified into class C_1 which is denoted as $P(C_1|\underline{x})$. This means that the posterior probability for class C_2 is then given by $P(C_2|\underline{x}) = 1 - P(C_1|\underline{x})$. This can be combined into a single expression such that the probability of observing either of the response values is

$$p(y|\underline{x}) = \hat{y}^y(1-\hat{y})^{1-y}$$

which is a case of a Bernoulli distribution. If we assume that the observations are independently drawn from a Bernoulli distribution with probability of success $\hat{y} \approx P(C_1|\underline{x})$, the likelihood function can be written as:

$$L = \prod_{i=1}^{n} (\hat{y}_i)^{y_i}(1-\hat{y}_i)^{(1-y_i)}$$

By taking the negative natural logarithm of this likelihood function we arrive at the

cross-entropy function

$$E = -\sum_{i=1}^{n}\{y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)\}$$

The absolute minimum of this function will occur when $\hat{y}_i = y_i$ for all i.

Bishop (1995) showed that the appropriate activation function to be used with the cross entropy error function for the output of the network, is the logistic sigmoidal activation function which is given by

$$g(a) = \frac{1}{1 + \exp(-a)}$$

This will ensure that the network outputs can be interpreted as probabilities. This is true for a neural network with any number of layers. As long as the output layer uses a logistic sigmoid activation function, the outputs will lie between 0 and 1 and can be considered as probabilities.

This shows that when we train a neural network for classification purposes by minimising the cross-entropy function

$$E = -\sum_{i=1}^{n}\{y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)\} \tag{2.29}$$

the estimates for the weights $\underline{w}$ corresponds to maximum likelihood estimates and by using a logistic sigmoid activation function for the output layer, the outputs from the network $\hat{y}$ can be interpreted as posterior probabilities of class C_1 membership, i.e. $\hat{y} = P(C_1|\underline{x})$.

Classification of more than two classes

We now consider the case where an input has to be classified into one of K groups. The objective is to arrive at an error function which can be used when the number of classes

is greater than two. Consider a neural network which has K outputs $\hat{y}_k$, $k = 1, \ldots, K$. Let the response variables $y_1, \ldots, y_K$ be coded using the 1-of-K coding scheme. We wish to set up the neural network such that the probability of belonging to the k-th class, given input $\underline{x}$, should correspond to the k-th output from the neural network, that is $P(C_k|\underline{x}) = \hat{y}_k$. The conditional distribution of the response variable $\underline{y}_i = (y_{i1}, \ldots, y_{iK})$ given an input $\underline{x}_i$ can be written as

$$p(\underline{y}_i|\underline{x}_i) = \prod_{k=1}^{K} (\hat{y}_{ik})^{y_{ik}}$$

(Bishop, 1995). Assuming that the input patterns are independently drawn, we can form the likelihood function

$$L = \prod_{i=1}^{n} \prod_{k=1}^{K} (\hat{y}_{ik})^{y_{ik}}$$

and by taking the negative logarithm of the likelihood function leads to an error function of the form

$$E = -\sum_{i=1}^{n} \sum_{k=1}^{n} y_{ik} \ln \hat{y}_{ik} \tag{2.30}$$

Again we must choose an appropriate output unit activation function to match this error function. Bishop (1995) shows that the softmax activation function is the appropriate activation function to use in this case. The softmax activation function has the form

$$\hat{y}_k = \frac{\exp(y_{in-k})}{\sum_{\forall k'} \exp(y_{in-k'})}$$

where y_{in-k} denotes the net input into the k-th output neuron. The softmax activation function has the properties that $0 \leq \hat{y}_k \leq 1$ and $\sum_{k=1}^{K} \hat{y}_k = 1$ as required for probabilities.

A neural network for approximating the posterior probabilities of class membership where the number of classes is greater than two, can be set up by using the error function in (2.30) for training a network which uses a softmax activation function for the output units.

Under these conditions a sufficiently general neural network trained on a large data set will approximate the posterior probabilities of class membership (Bishop, 1995).

Discriminant analysis

In section 2.4.3 it was shown that in order to minimise the probability of misclassification, an observation should be assigned to the class with the largest posterior probability. The posterior probability of class membership can be determined from the class-conditional densities by using Bayes' theorem (cf. section 2.4.3). These class-conditional densities then need to be estimated from the data at hand. Another approach which avoids the need of estimating the class-conditional densities, is to use discriminant functions in which the form of the discriminant function is chosen beforehand and the parameters of the function is estimated from the training data. These discriminant functions enables us to provide a classification directly when provided with an input observation.

In this section we will discuss the way in which a neural network can be used to model discriminant functions. We will see that a single layer network can be used for linear discriminant functions including logistic discrimination which will be discussed in section 2.4.3. We will also explore the types of discriminant functions that a multilayer perceptron can model.

We will begin by considering the two class classification problem. For this problem, we need to define a discriminant function $y(\underline{x})$ such that an observation will be assigned to class C_1 if $y(\underline{x}) > 0$ and to class C_2 when $y(\underline{x}) < 0$. The simplest choice for the discriminant function is to choose a function which is linear in the inputs $\underline{x}$. This discriminant function can then be written as

$$y(\underline{x}) = w_0 + w_1 x_1 + \cdots + w_p x_p \tag{2.31}$$

Bishop (1995) shows that the decision boundary $y(\underline{x}) = 0$, can be interpreted as a $(p-1)$ dimensional hyperplane in the p dimensional input space and that the normal distance from the origin to the hyperplane is given by

$$\frac{\underline{w}'\underline{x}}{\|\underline{w}\|} = -\frac{w_0}{\|\underline{w}\|}$$

An example of a linear discriminant function in a two dimensional input space is given in figure 2.16. Here the decision boundary is found by finding the values in the input space $\underline{x} = (x_1, x_2)$ which correspond to $y(\underline{x}) = 0$. We can see here that the weights of the decision boundary $\underline{w} = (w_1, w_2)$ determine the orientation of the decision boundary, while the bias w_0 determines the position of the decision boundary in the input space.

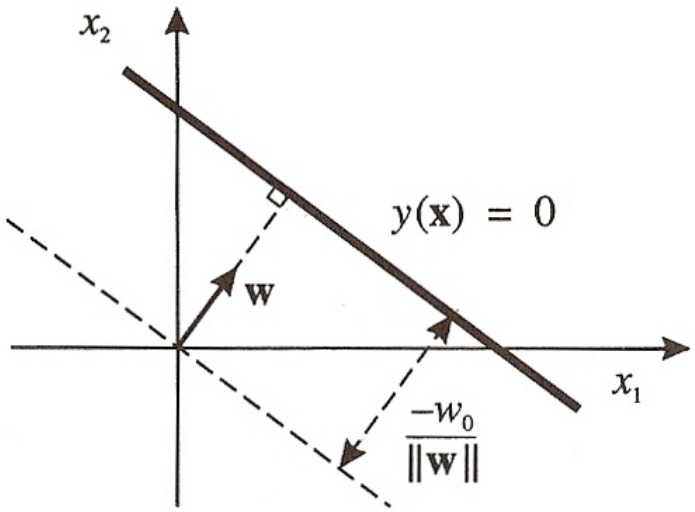


Figure 2.16: A linear decision boundary (Bishop, 1995, p.79)

A linear discriminant for a two class problem can be expressed as a single layer network with a threshold activation function (Sarle, 1994), with one output similar to what is shown in figure 2.2. A simple way to construct a neural network for discriminant analysis is to code the response variable y as follows:

$$y = \begin{cases} 1 & \text{if observation is from } C_1 \\ -1 & \text{if observation is from } C_2 \end{cases}$$

By using this response variable, we can then train a single layer network by using back-propagation. Once the weights in the network have been estimated we can then use it to classify a new observation $\underline{x}$, after it has been presented to the network. A classification is made to class C_1 if $\hat{y}(\underline{x}) > 0$ and to class C_2 otherwise. This is the same as using a threshold activation function for the output neuron. A neural network that has been set up in this way will construct a linear decision boundary to separate the two classes.

Extending this network to accommodate more than two classes can be done by using one discriminant function $y_k(\underline{x})$ for each of the K classes. The simplest way to do this is to let each of these K discriminant functions be linear in the components of $\underline{x}$. Each discriminant function can then be expressed as follows:

$$\begin{aligned} y_k(\underline{x}) &= \underline{w}_k'\underline{x} + w_{k0} \\ &= w_{k0} + w_{k1}x_1 + \cdots + w_{kp}x_p \\ &= \sum_{j=1}^{p} w_{jk}x_j \quad k = 1, \ldots, K. \end{aligned}$$

This can be expressed as a single layer neural network with K outputs as in figure 2.17. Using this network, a classification to class C_k is made if

$$y_k(\underline{x}) > y_k(\underline{x}) \qquad \forall\ k \neq j$$

A single layer neural network with a threshold activation function can only represent linear decision boundaries which makes its use very limited. Linear discriminant functions can be generalised by using nonlinear transformations of the inputs $x_1, \ldots, x_p$. This expands the range of decision boundaries that can be represented. A multilayer perceptron can

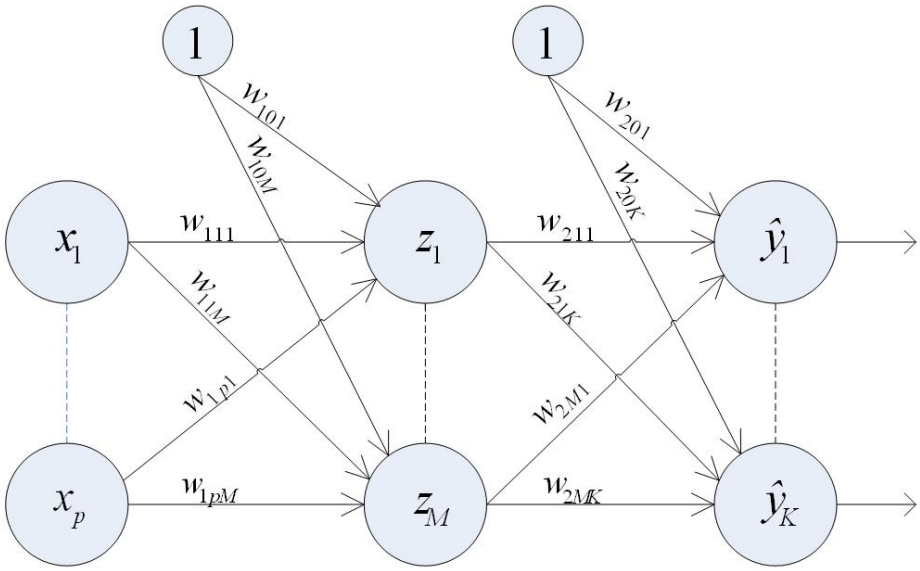


Figure 2.17: Using a neural network to model discriminant functions

be used to model generalised discriminant functions. Hornik et al. (1989) showed that a neural network with a sufficient number of hidden nodes can approximate any function to arbitrary accuracy. Figure 2.18 shows an example of the possible regions that a neural network, with a different number of hidden layers can map. All these neural networks use a threshold activation function for the output layer. It should be noted that the networks in this figure should be read from the bottom upwards instead of left to right as we have used up to now. The network in (a) is a single layer network. The network in (b) consists of one hidden layer and the network in (c) consists of two hidden layers. The networks in these examples only take two inputs in order to easily display the the classification areas on a two dimensional graph.

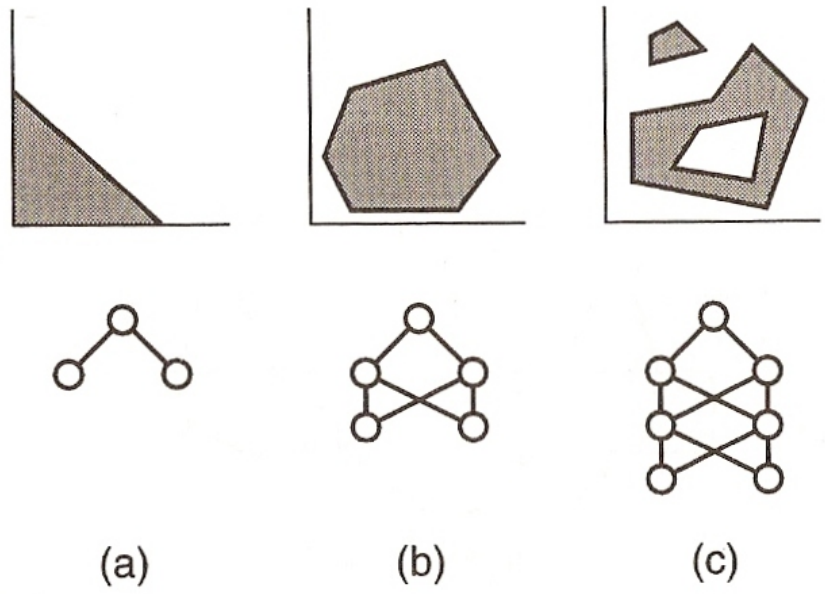


Figure 2.18: Possible decisions regions generated by feedforward neural networks

Logistic discrimination

We will start with an overview of logistic regression in statistics and then show how this can be linked to a single layer neural network with a sigmoidal activation function, which is often called the logistic perceptron.

A logistic regression model is typically used in statistics to model a binary or dichotomous dependent variable. We will now define the multiple logistic regression for a binary dependent variable as is given by McCullagh and Nelder (1983). Let Y be a variable which can take on the values 1 or 0. This can be used to identify whether a certain observation belongs to one of two classes, a value of 1 indicating that the observation belongs to the one class and 0 if the observation belongs to other class. The multiple logistic regression

model has the form

$$log\left(\frac{Pr(y=1|\underline{x})}{Pr(y=0|\underline{x})}\right) = log\left(\frac{Pr(y=1|\underline{x})}{1-Pr(y=1|\underline{x})}\right)$$
$$= \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p \tag{2.32}$$

The term $\frac{Pr(y=1|\underline{x})}{Pr(y=0|\underline{x})}$ is known as the odds. The logistic model indicates that we are modelling the *log(odds)* (also known as the *logit*) as a linear function of the inputs $\underline{x}$.

(2.32) can be rewritten as:

$$Pr(y=1|\underline{x}) = \frac{\exp\left(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p\right)}{1 + \exp\left(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p\right)} \tag{2.33}$$
$$= \frac{\exp\left(g(\underline{x})\right)}{\exp\left(1 + g(\underline{x})\right)} \tag{2.34}$$

By rewriting (2.32) as (2.34) we can see that the probability that an observation belongs to class C_1 for given values of the independent variables, that is $Pr(Y=1|\underline{x})$, is a function that is nonlinear in the parameters $\beta_0, \ldots, \beta_p$. These parameters in the logistic regression model is estimated by using the principle of maximum likelihood.

It can be shown that the likelihood function for the two class logistic regression model can be written as

$$l(\underline{\beta}) = \sum_{i=1}^{n} \{y_i \log p_i + (1-y_i)\log(1-p_i)\}$$
$$= \sum_{i=1}^{n} \{y_i(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p) - \log(1 + \exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p))\} \tag{2.35}$$

(Hastie et al., 2001).

The values of the response variables y_i are coded as 0 or 1 and $p_i = P(Y=1|\underline{x}_i)$. Maximisation of this likelihood function with regards to the unknown parameters $\beta_0, \ldots, \beta_p$

requires the use of numerical optimisation. The Newton-Raphson iterative algorithm is often used for this purpose and details on how to estimate the parameters in the logistic regression model using the Newton-Raphson algorithm can be found in Hastie et al. (2001).

The binary logistic model can be extended to a K class problem. This is achieved by choosing a reference class and then modelling the *log(odds)* of the $K-1$ other classes, each with respect to the reference class, as a linear function of the inputs $x_1, \ldots, x_p$. The K-class logistic regression model has the form

$$\begin{aligned}
\log \frac{Pr(G=1|\underline{x})}{Pr(G=K|\underline{x})} &= \beta_{10} + \beta_{11}x_1 + \cdots + \beta_{1p}x_p \\
\log \frac{Pr(G=2|\underline{x})}{Pr(G=K|\underline{x})} &= \beta_{20} + \beta_{21}x_1 + \cdots + \beta_{2p}x_p \\
&\vdots \\
\log \frac{Pr(G=K-1|\underline{x})}{Pr(G=K|\underline{x})} &= \beta_{(K-1)0} + \beta_{(K-1)1}x_1 + \cdots + \beta_{(K-1)p}x_p
\end{aligned} \tag{2.36}$$

where $Pr(G=k|\underline{x})$ denotes the probability that an observation belongs to class C_k (Hastie et al., 2001). The model specified in (2.36) uses the last class as the reference class, but any of the K available classes can be used for this purpose without affecting the resulting classifications from the model. Expressions for the probabilities of class membership can be found from (2.36) and have the form

$$\begin{aligned} Pr(G=k|\underline{x}) &= \frac{\exp(\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{kp}x_p)}{1+\sum_{l=1}^{K-1}\exp(\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p)} \quad ,k=1,\cdots,K-1 \\ Pr(G=K|\underline{x}) &= \frac{1}{1+\sum_{l=1}^{K-1}\exp(\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p)} \end{aligned} \tag{2.37}$$

The probabilities of class membership lies between 0 and 1 and the sum of all the probabilities is one. The parameters in the K-class logistic regression model are estimated by using the principle of maximum likelihood similarly to the 2-class problem.

Logistic regression models offer an advantage in that it can be used for statistical inference. This implies that the model can be used to assess the effects of the different input variables on the response variable by interpreting the model coefficients. Used as a data analysis tool, typically many logistic regression models are fit to the data and the simplest model which yields a good fit is chosen. A logistic regression model can also contain interaction terms between the independent variables. More information on using logistic regression for data analysis and inference can be found in Hosmer and Lemeshow (1989).

The logistic regression model is equivalent to using a single layer neural network with a logistic activation function (Sarle, 1994). This is commonly known as the logistic perceptron. The output from the logistic perceptron is written as

$$\hat{y} = f(\sum_{j=0}^{p} w_i x_i)$$

where $x_0, \ldots, x_p$ denotes the inputs and $w_0, \ldots, w_p$ denotes the weights and $x_0 = 1$. $f(a) = \frac{1}{1+\exp(-a)}$ denotes the logistic activation function. The weights in the logistic perceptron can be interpreted in a similar fashion as in the logistic regression model (Schumacher et al., 1996).

The weights $\underline{w} = (w_0, \ldots, w_p)$ in the logistic perceptron are determined by minimising the sum of squares error function

$$E(\underline{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

or by minimising the cross-entropy error function

$$E = -\sum_{i=1}^{n}\{y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)\}$$

The error functions can be minimised by using the backpropagation method (cf. Section 2.3.5) or any other numerical optimisation method. By using the cross-entropy error function the estimates of the weights correspond to the maximum likelihood estimates (cf. Section 2.4.3).

The main difference between the logistic perceptron model and logistic regression model is that the weights in the logistic perceptron are usually obtained by backpropagation, i.e. method of steepest descent, but is not restricted to this method of numerical optimisation and the Newton-Raphson algorithm can also be used to estimate the unknown weights. This implies that the logistic perceptron is equivalent to the logistic regression model when the cross-entropy error function is minimised by the Newton-Raphson method.

The logistic perceptron can be extended to K classes by choosing a reference class and then fitting separate logistic perceptrons for each class versus the reference class. This approach is used often but the problem with it is that the probabilities does not sum to one (Cherkassky et al., 1994). A more appropriate extension of the logistic perceptron to the problem of K classes is the log-linear model for posterior probabilities.

Consider a feedforward neural network with linear activation functions. The target variable can be coded using the 1-of-K coding scheme and the neural network provides an

output $\hat{y}_k$ for each of the K classes. Ripley (1994) gives the error function, corresponding to the log likelihood, for network training as

$$E = \sum_{k=1}^{K} y_k \log \frac{y_k}{P(C_k|\underline{x})}$$

where this is summed over all the observations. The posterior probabilities are given by

$$P(C_k|\underline{x}) = \frac{\exp \hat{y}_k}{\sum_{k=1}^{K} \exp \hat{y}_k} \qquad k = 1, \ldots, K$$

and a classification is made to the class with the largest output $P(C_k|\underline{x})$ or equivalently in this case $\hat{y}_k$. This log-linear approach is named softmax (Bridle, 1989). More information on how to fit a neural network using softmax can be obtained in Cherkassky et al. (1994); Ripley (1994).

2.4.4 Generalising capability of neural networks

The main aim of neural networks should not be to model or memorise the training data exactly, but rather to model the underlying generator of the data, i.e. to build a statistical model of the process or distribution from which the data in the training data set originates from. This is very important if the neural network is going to be used to make predictions on data that was not part of the training data set. The neural network literature describes this as that the network must be able to generalise well to unseen data.

Training data can be regarded as being generated by some deterministic function of $\underline{x}$ with added noise. This can be written as

$$y = h(\underline{x}) + \varepsilon$$

where $h(\cdot)$ is a deterministic function and ε is a zero-mean error term which is added to the response data. The main goal of neural networks is to model the function $h(\cdot)$. Overfitting occurs when we allow the function that is fitted to the training data to become too flexible and then interpolate the training data instead of modelling the underlying process $h(\cdot)$, i.e. the model is fitted to the noise in the data.

Consider a simple example of fitting a polynomial to only 12 training observations (Hertz et al., 1991). Say the data was generated by a second order polynomial with added Gaussian noise having mean zero and a constant variance. The number of coefficients of the polynomial fitted to the data determines the complexity or flexibility of the model. If we fit a simple model, for example a straight line in this case, the model will not generalise well because it is a poor approximation to the true underlying function. This model exhibits a high bias. On the other hand, if we fit a polynomial of high order, the model will be too flexible and will start to interpolate the training data. In the extreme case of where we fit a polynomial of the same order as the number of training cases, the model will fit the training data perfectly, but this model will generalise poorly since we have fitted the model to the noise in the training data and not the true underlying function. This model exhibits a high variance. An example which illustrates this point is shown in figure 2.19.

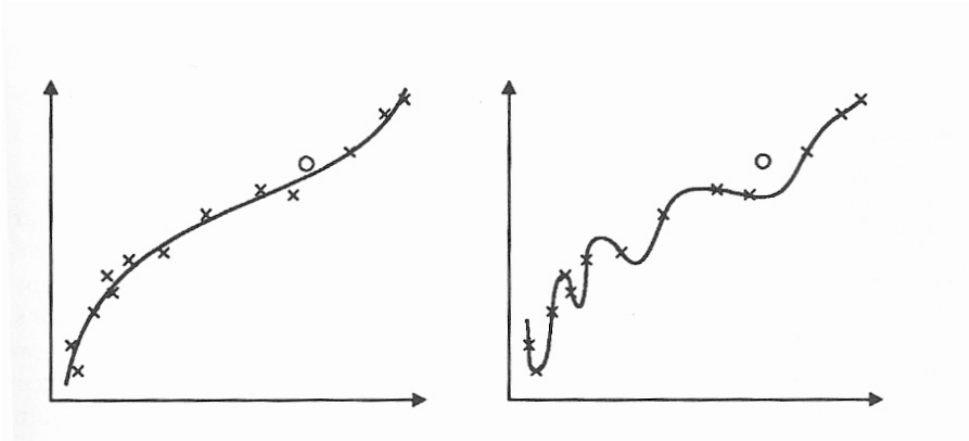


Figure 2.19: (a) A model that approximates the underlying distribution well. (b) An overfitted model (Hertz et al., 1991, p.147)

In this example, the sample data set was split into two parts. A training sample represented by the *crosses* and a test sample represented by the *circle* in figure 2.19. The model is fitted using the training data, and the test data is used as a measure of performance of the model. In the first graph we see that this model fits well and will generalise well to unseen data. In the second graph, the overfitted polynomial fits the training data perfectly, but will have poor performance on the unseen test data. This is due to that the model complexity is too high and it gives a poor approximation to the true underlying generator of the data.

This point can be best explained by the bias-variance trade-off. The bias-variance trade-off states that the mean squared error of prediction, can be decomposed into a bias and a variance component. Mathematically we can write this as:

$$MSE = E[(y - \hat{y})^2] = \underbrace{(y - E[\hat{y}])^2}_{\text{bias}^2} + \underbrace{E[(\hat{y} - E[\hat{y}])^2]}_{\text{variance}} \tag{2.38}$$

Variance and bias are complementary quantities and the goal is to select a model which gives the best balance between bias and variance, leading to the smallest prediction error when used for generalisation.

To enable neural networks to generalise well to unseen data, the complexity of the neural network must be controlled. This can be done through two complementary approaches: determine the correct network architecture and avoid overfitting of the network (De Veaux and Ungar, 1994a). Both these approaches will be discussed in more detail.

The complexity of a neural network is determined by the network architecture and in particular by the number of hidden layers in the network and also the number of hidden units within each of these hidden layers. The complexity of the optimal neural network model is determined by the number of training observations, the amount of noise within

the data and how complex the function is that we are trying to approximate (Svozil et al., 1997). Generally there is no simple way to optimise the network architecture. A multilayer neural network with one hidden layer and a large enough number of hidden units is a universal approximator of any continuous function (Hornik et al., 1989; Hornik, 1991). This means that a neural network with one hidden layer is generally sufficient, although the complexity of the problem may justify the use of a second hidden layer in some cases. The other problem is choosing the number of hidden units. One approach which can be followed is to try fitting neural networks with different number of hidden neurons. For each of these neural networks an estimate of the generalisation error can be obtained and the network with the lowest generalisation error can be used. An alternative way of deciding on the amount of hidden neurons to use, is to start with a large amount of hidden neurons and then prune the neurons until the network performance starts to suffer. This is referred to as a pruning technique. The opposite version of this is a growing technique in which you start with a very simple neural network architecture and then make it more complex. More information on growing and pruning techniques for neural networks can be found in Bishop (1995).

Another way to control the complexity of the network at hand is to use a sufficiently general neural network, meaning a neural network with a large number of hidden units, and then use an approach to prevent the network from overfitting. Overfitting is also sometimes called overtraining the network. Estimation of the weights in a neural network is done through training, which corresponds to minimisation of an error function on the training set. For a neural network with many hidden units, the error rate on the training data set will continue to decrease as the number of training epochs increases and the error rate will become zero when the network perfectly fits the training data. This results in an overfitted network with a poor generalising capability to unseen data. This means that we cannot use the error on the training data as a measure of network performance since for a neural network with many hidden units, the training error is a function of the

number epochs.

The first method which can be used to avoid overfitting is called early stopping. The easiest way to implement this is to split the data into two samples: a training sample and a validation sample. The neural network is trained using the training sample and after each training epoch, the error made by the network is computed using the validation sample. This is called the validation error. As training increases, this will cause the training error to become smaller, but at some stage the validation error will start to increase. This is indicative that the model is being fitted to the noise in the training data and hence the model is being overfitted and training should be stopped. The training and test error of a neural network with an increasing number of hidden nodes, or with a large number of hidden nodes and an increasing number of training epochs, is shown in figure 2.20. We see that the training error continues to decrease with network training, while the test error will reach a minimum and then increase after that point, indicating that a network is being overfitted.

It should be noted that validation error does not provide us with a good estimate of generalisation error (Svozil et al., 1997). A solution to this is to divide the data into three parts: the training data, validation data and testing data. Calculating the prediction error on the testing data will provide a good estimate for generalisation error. This is due to the fact that the testing data set is not used in the training of the model. The problem is that this method reduces the amount of data that is available for network training.

A more elaborate version of this split sample technique is called cross-validation in which all the data is used for model training and a good estimate of generalisation error can be obtained. This technique comprises of dividing the data into k equally sized subsets. One of the k subsets is then left out and the network is trained on the remaining data. The omitted subset is used to compute prediction error. This is repeated k times, each time

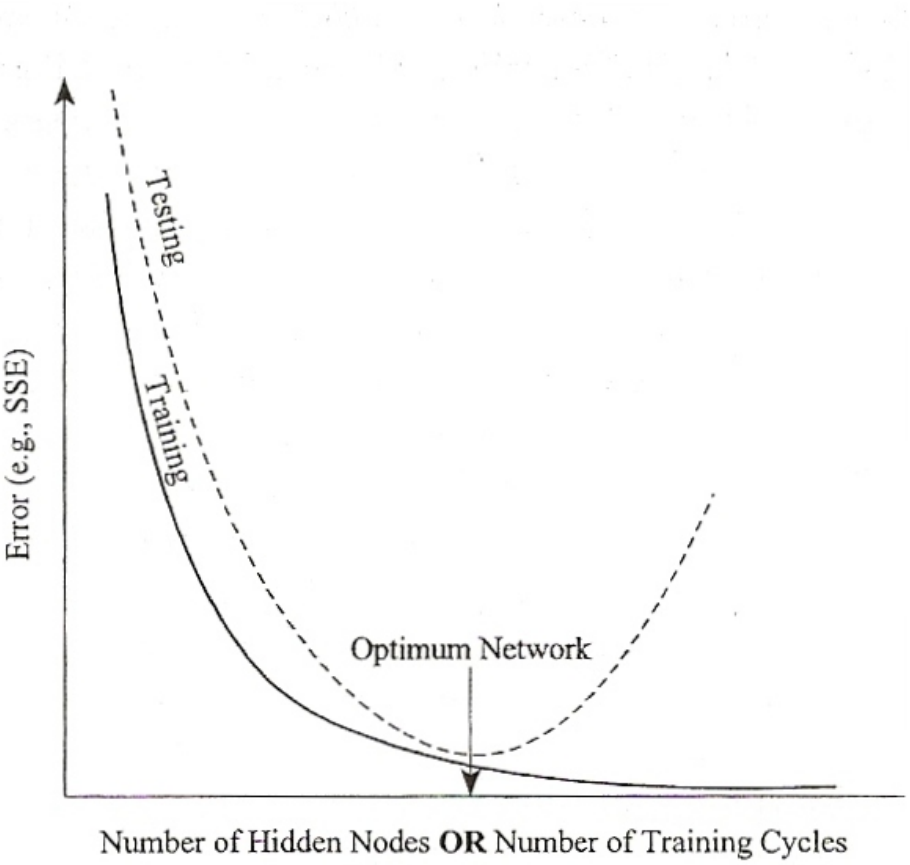


Figure 2.20: Using the split sample approach (Basheer and Hajmeer, 2000)

leaving out a different one of the k subsets. After repeating this procedure k times, the values for prediction error can be combined to obtain an estimate for the generalisation error of the network. For a discussion of the disadvantage of using cross-validation for neural networks consult (Svozil et al., 1997).

Regularisation is another method which can be used to avoid overfitting of a neural network. This method involves using a neural network with large number of hidden neurons and the network is trained by minimisation of a penalised error function. This penalised error function consists of the usual error function for example the sum of squares for error, but added to this is a penalty term. The penalised error function has the form

$$\tilde{E} = E + \upsilon\Omega$$

where Ω is the penalty term and v is called the regularisation coefficient. This penalty term is chosen in such a way that it encourages smoother network mappings. One of the simplest forms for the penalty term is to take the sum of squares of the weights in the neural network. This is known as weight decay and is most often used as a method of regularisation in neural networks. The penalty term when using weight decay can be written as

$$\Omega = \frac{1}{2} \sum_{\forall i} w_i^2$$

where this sum is over all the weights that are present in the network and the $\frac{1}{2}$ in front of the summation is to simplify computation (Bishop, 1995). In a statistical linear model this method of regularisation is called ridge regression and is a method of coefficient shrinkage (Hastie et al., 2001).

The objective of the penalty term as it is chosen when using weight decay is to penalise large absolute values of the weights. Bishop (1995); Svozil et al. (1997) gives detailed discussions of why large weights (in absolute value) decrease generalisation performance of the network, but a basic general reason is that it increases the variance of the network outputs.

Other forms of the penalty function exist which does not involve penalising the size of the weights, as in weight decay. One such method is to choose the penalty term in such a way to prevent fitting a network function with relatively high degree of curvature (Bishop, 1995). This is done by penalising a function with high curvature, since this increases the variance of the network outputs resulting in an overfitted neural network. The penalty term in this case will consist of the second derivative of the network function, since the second derivative of a function gives an indication of curvature.

Bishop (1995, pp345–346) claims that the method of weight decay will give similar results to early stopping when the sum of squares for error function is used as the criterion for

network training.

2.4.5 Conclusion

When neural networks first arrived onto the scene, lots of exaggerated claims were made about the capabilities of neural networks. Some of the claims made were that neural networks can be used without any experience and that they can be used as an automatic modelling tool without having the need to learn more complicated statistical methods (Sarle, 1994; De Veaux and Ungar, 1994a). These claims are not true for most applications of neural networks for data analysis. Removing outliers from the data set before network training and investigating the error distributions will increase the performance of the neural network. This means that exploratory data analysis is still an important part of the data modelling process and that neural networks cannot be made into a fully automated process and does still need some human interaction (De Veaux and Ungar, 1994a).

For many neural networks, there is a statistical technique that can be utilised to perform the same task as the neural network. These statistical techniques include generalised linear models, nonlinear regression, nonparametric regression, discriminant analysis, projection pursuit regression, principal regression and cluster analysis (Sarle, 1994). Some of the similarities between neural network models and statistical models were discussed in this section. For more information on relating neural network models to statistical models consult (Sarle, 1994),

The question that is asked very often is, which one is better: statistical techniques or neural networks? There is no clear-cut answer to this question. Both neural networks and statistics have their place as modelling tools and the choice of technique depends on

the problem that needs to be solved.

Neural networks, like nonparametric statistics, does not impose any functional form on the relationship between the dependent and independent variables. Multilayer perceptrons in particular can be used as a flexible nonlinear model that, given the network has enough hidden neurons and the training sample is large, can approximate any function to arbitrary accuracy (White, 1992). The disadvantage of not imposing a functional form between the dependent and independent variables is that the estimated coefficients in the model cannot be interpreted (Stern, 1996).

Neural network models outperform statistical regression with regard to prediction accuracy when the dimensionality of the problem is high, the functional relationship that needs to modeled is complex and the sample contains a large number of observations (Basheer and Hajmeer, 2000). Neural networks are also shown to perform well even when the data contains a high level of noise (Svozil et al., 1997).

When the focus is on statistical inference, regression models are more appropriate to use than neural networks. Regression models allows a statistician to describe the underlying process, generate hypotheses, provide confidence intervals, test the model assumptions and investigate the data closely. On smaller data sets with low dimensional problems, statistical techniques regularly outperform neural networks techniques and results in less complex models which are easier and faster to fit computationally (Basheer and Hajmeer, 2000).

There is a vast number of papers available in which neural networks are compared to statistical techniques. Comparative studies of regression neural networks and statistical regression methods can be found in College et al. (1995). One of the most comprehensive studies of supervised classification techniques in statistics, neural networks and machine

learning has been done by Michie et al. (1994). A thorough study between logistic regression and neural networks for logistic discrimination has been done by Dreiseitl and Ohno-Machado (2002).

Neural networks and statistics are not competing methods of data analysis. Neural networks is a nonparametric method that should rather be used to compliment other nonparametric statistical methods since they can be useful in statistical applications and can be an extremely powerful tool in a purely predictive problem. For this reason alone it should get statisticians involved in the field of neural networks.

Chapter 3

Practical Application

3.1 Introduction

In this chapter the aim will be to implement a neural network to solve a business problem. This problem is currently handled with a set of rules which have been decided upon after trial and error testing. For this reason I think that it might be worthwhile to attempt to use a neural network as an alternative to these rules. In the next section we will give an overview of the type of neural networks that we will be using and also discuss some of the theoretical aspects using results stated in section 2.3 and section 2.4. In section 3.3 we will conduct a small simulation study to show some characteristics of neural networks. In section 3.4 a thorough description of the problem at hand will be given and in section 3.5 we will describe the data that will be used. The analysis and results will follow in sections 3.6 and 3.7 respectively. In sections 3.8, 3.9 and 3.10 we will discuss various issues regarding the performance of the neural networks, implementation of the neural network and also some of the difficulties that were experienced during the analysis. We will end this chapter with some final remarks in section 3.11 and then give some future research

topics in section 3.12.

3.2 Implementing neural networks

In this section we will describe how to use a neural network for a practical problem. The aim will be to use a two-layer network and solve a regression problem and also a classification problem. We will start by giving a short description of the neural networks that will be used for these purposes. This description will include the way in which to choose the architecture of the network, the appropriate activation functions and error functions to use, and how the network will be trained. These topics have already been discussed in general for neural networks in the literature review (cf. chapter 2). In this section, we will use this knowledge to customise the networks specifically for the problem at hand. These networks will be discussed in more detail.

3.2.1 Notation used

A description of the notation that is going to be used in this chapter will be given here. This notation is the same as was used in the literature review but a complete summary of the notation will be given to make things easier. We will start with the weights: The weights that connect the input units to the hidden units will be denoted by w_{1jm}. The 1 in the subscript specifies that the weight occurs in the first layer of adaptive weights, the j subscript specifies the neuron from which the weight connect and the subscript m denotes the unit to which the weight connects. The weights in the second layer are denoted by w_{2mk} and this follows the same methodology as those in the first layer. The i-th observation is denoted by $(\underline{x}_i, \underline{y}_i)$, where $\underline{x}_i = (x_{i1}, \ldots, x_{ip})$ are the independent variables and $\underline{y}_i = (y_{i1}, \ldots, y_{iK})$ denote the dependent variables. When working with a two-layer

network, we have that the weighted sum of the inputs forms the net input into the hidden unit which are denoted by z_{in-m}. For observation i, this is written as

$$z_{in-im} = \sum_{j=0}^{M} w_{1jm} x_{ij}$$

Remember that $x_0 = 1$ denotes the bias term.The net-input is passed through the activation function and produces output

$$z_{im} = f\left(\sum_{j=0}^{M} w_{1jm} x_{ij}\right)$$

Following the same principle as above we have that the output $\hat{y}_i k$, when the neural network is presented with input $\underline{x}_i$ is given by

$$\hat{y}_{ik} = g\left(\sum_{m=0}^{M} w_{2mk} f\left(\sum_{j=1}^{p} w_{1jm} x_{ij}\right)\right)$$

To keep the notation simple, I will generally not specify the number of the observation that is presented to the neural network, i.e. subscript i will be dropped from the notation. This means that the output $\hat{y}_k$, when the neural network is presented with an input $\underline{x}$ from the training sample is given by

$$\hat{y}_k = g_k\left(\sum_{m=0}^{M} w_{2mk} f\left(\sum_{j=0}^{p} w_{1jm} x_j\right)\right)$$

The aim is to keep the notation as simple as possible. If we have a look at the notation, we will see that the subscript i is always used to refer to the observation number, subscript j is always used to refer to an input unit or variable, subscript m is used to denote a hidden unit and subscript k is used to refer to an output unit. The weights always follow the convention in which the first subscript denotes the number of the layer of adaptive weights in which the weight occurs, the second subscript denotes the input unit from

which the weight connects and the last subscript denotes the output unit to which the weight connects. When there is only one output unit, which is often the case when the neural network is to be used for regression, then the subscript k is dropped from all the notation in the network. This means that the network output is then denoted by $\hat{y}_1 = \hat{y}$ and the weights which connect the hidden-layer to the output-layer are denoted by w_{2m}. It can be seen that the notation is chosen in such a way that each subscript has a specific meaning and hence can easily be dropped when it is not needed without any confusion to what the remaining subscripts mean.

3.2.2 Neural network for regression

Our first aim will be to train a neural network for regression purposes. In section 2.3.5 we derived the backpropagation rule for a general two-layer network trained by minimisation of the error sum of squares. This derivation did not specify the form of the activation functions. From the literature, it seems that the most appropriate activation functions to use for a neural network is a sigmoid activation function for the hidden layer and linear output activation functions (Bishop, 1995).

The neural network to be used for this specific regression problem consists of one dependent variable and hence the network will only have one output node. This means that $K = 1$ and that the subscript k can be dropped from the notation used for this neural network. This network can graphically be illustrated as in figure 3.1.

The output from this neural network, $\hat{y}$, when presented with observation $\underline{x}$ is calculated as follows:

$$\hat{y} = g\left(\sum_{m=0}^{M} w_{2m} f\left(\sum_{j=0}^{p} w_{1jm} x_j\right)\right) \tag{3.1}$$

Recall from section 2.3.5 that the weight updates for a neural network trained by back-

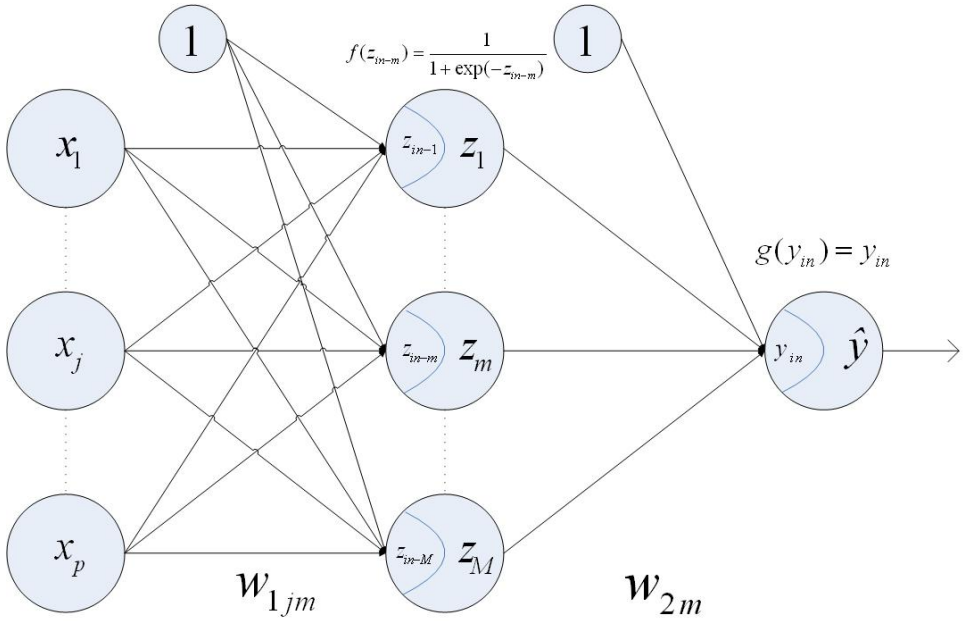


Figure 3.1: Illustration of neural network that will be used for regression.

propagation is:

$$\begin{aligned} w_{2mk}^{(t+1)} &= w_{2mk}^{(t)} + \alpha(y_k - \hat{y}_k)g_k'(y_{in-k})z_m \\ &= w_{2mk}^{(t)} + \alpha\delta_{2k}z_m \qquad \text{where } \delta_{2k} = g_k'(y_{in-k})(y_k - \hat{y}_k) \\ &= w_{2mk}^{(t)} + \alpha\Delta w_{2mk} \qquad \text{where } \Delta w_{2mk} = \delta_{2k}z_m \end{aligned} \tag{3.2}$$

for the weights connecting the hidden layer to the output layer and

$$\begin{aligned} w_{1jm}^{(t+1)} &= w_{1jm}^{(t)} + \alpha f_m'(z_{in-m}) \sum_{k=1}^{K} w_{2mk}(y_k - \hat{y}_k)g_k'(y_{in-k})x_j \\ &= w_{1jm}^{(t)} + \alpha\delta_{1m}x_j \qquad \text{where } \delta_{1m} = f_m'(z_{in-m}) \sum_{k=1}^{K} w_{2mk}\delta_{2k} \end{aligned} \tag{3.3}$$

for the weights connecting the hidden layer to the output layer. For this particular neural network that we are going to use we will use the logistic sigmoid activation function for the hidden units:

$$f_m(a) = f(a) = \frac{1}{1+\exp(-a)} \tag{3.4}$$

We know that the derivative of the logistic activation can be expressed as follows:

$$f'(a) = f(a)(1 - f(a)) \tag{3.5}$$

For the output units, a linear activation function will be used:

$$g_k(a) = g(a) = a \tag{3.6}$$

When a neural network is used for regression, the problem usually consists of a single dependent variable (i.e. $K = 1$). This means that the objective is to choose the weights in the neural network to minimise the standard sum of squares for error function

$$E[\mathbf{w}] = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

or to minimise the error made by each pattern, given by

$$(y - \hat{y})$$

over each one of the training observations.

If we use (3.2) we see that the weight update rules for the weights which connects the hidden layer to the output layer for this particular neural network, i.e. a neural network which has only one output, linear output unit activation functions and logistic hidden unit activation functions, is given by:

$$w_{2m}^{(t+1)} = w_{2m}^{(t)} + \alpha(y - \hat{y})z_m \tag{3.7}$$

Notice that this is the online version of the backpropagation rule but this rule can easily be extended by accumulating the weight updates until each observation in the training sample has been presented to the neural network and then updating the rule, hence the

weights are only updated after each epoch. The batch weight updates is as follows:

$$w_{2m}^{(t+1)} = w_{2m}^{(t)} + \alpha \sum_{i=1}^{n} (y_i - \hat{y}_i) z_{im} \tag{3.8}$$

From (3.3) the weight updates for the weights which connect the input units with the hidden units is given by

$$w_{1jm}^{(t+1)} = w_{1jm}^{(t)} + \alpha w_{2m} (y - \hat{y}) z_m (1 - z_m) x_j \tag{3.9}$$

Again we can decide to accumulate the weight updates until each observation in the training sample has been presented to the neural network, in which case the weight update will be

$$w_{1jm}^{(t+1)} = w_{1jm}^{(t)} + \alpha w_{2m} \sum_{i=1}^{n} (y_i - \hat{y}_i) z_{im} (1 - z_{im}) x_{ij} \tag{3.10}$$

Note that the subscript i is added to the above equation. This is just to make it clear that the network was presented with observation i in the sample. Thus z_{im} is the output of the m-th hidden unit when the network was presented with observation i. This then follows similarly for the output units.

3.2.3 Neural network for classification

A neural network will also be used for classification purposes. This neural network will be used to classify an input to one of K mutually exclusive classes where the number K will be greater than two. The way in which this problem is approached is by using a 1-of-K coding for the dependent variable. This means that we code the dependent variable into a number of dummy variables, with the number of dummy variables used being the same as the number of categories that the dependent variable can take. The most appropriate neural network for this problem seems to be a network with sigmoid hidden activation

functions, softmax output activation functions and number of outputs equal to the number of categories that dependent variable can take on. The sum of squares error function can be used to train this network but as was stated in section 2.4.3, the cross-entropy error function would be more suitable for this problem. This will ensure that the outputs from the network can be interpreted as posterior probabilities. The reason that I think this will be useful is that we can then assign a threshold and say that only when the posterior probability of a classification exceeds a certain threshold then we classify that specific observation. I will now proceed to give an overview of the design of the network that will be trained for classification purposes and how this network is to be trained using gradient descent.

Graphically the network can be represented as in figure 3.2.

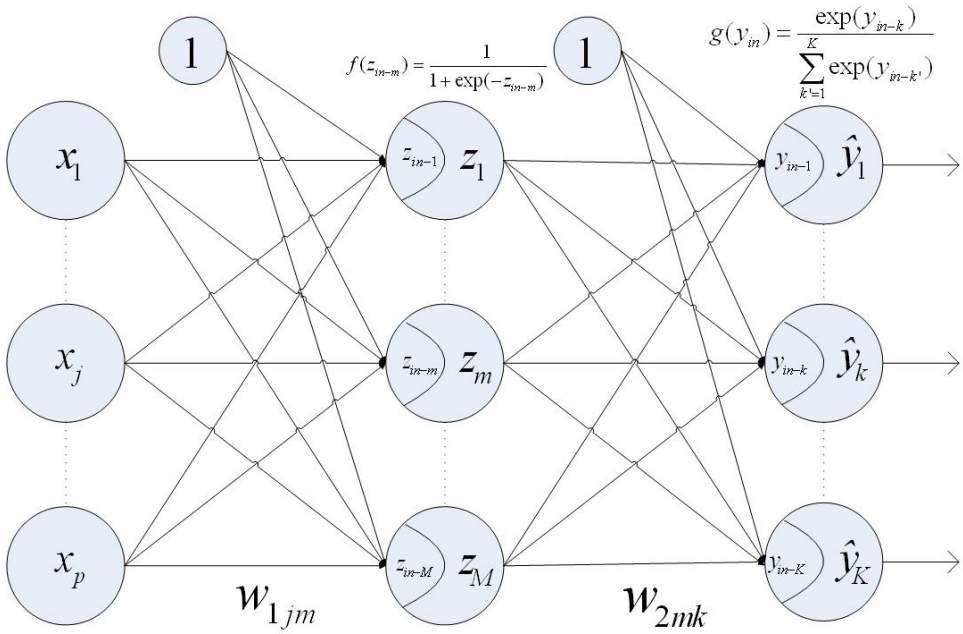


Figure 3.2: Illustration of neural network that will be used for classification purposes.

The network is trained by using a 1-of-K dependent variable coding and the output for output node k, which is denoted by $\hat{y}_k$, when presented with observation $\underline{x}$ is calculated

as follows:

$$\hat{y}_k = g\left(\sum_{m=0}^{M} w_{2mk} f\left(\sum_{j=0}^{p} w_{1jm} x_j\right)\right) \tag{3.11}$$

For this network the activation function for the hidden units is chosen as the logistic function:

$$f_m(a) = f(a) = \frac{1}{1+\exp(-a)} \tag{3.12}$$

for which the derivative can be expressed in a convenient form:

$$f'(a) = f(a)(1 - f(a)) \tag{3.13}$$

For the output units, we use the softmax activation function which means the predicted value on output k is given by

$$\hat{y}_k = \frac{\exp(y_{in-k})}{\sum_{k'=1}^{K} \exp(y_{in-k'})}$$

where $\sum_{k'=1}^{K} \exp(y_{in-k'})$ is the sum of the natural exponents of all the net inputs into the output units. Hence, the softmax function basically ensures that the sum of all the outputs add up to one. This is especially necessary when we want to interpret the outputs as probabilities.

It was shown in section 2.4.3 that the most appropriate error function to use to train a neural network for classification of more than two classes is the cross-entropy error function given by:

$$-\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} \ln(\hat{y}_{ik})$$

The gradient descent algorithm can then be used to estimate the weights in the neural network that will minimise this error function.

Berka et al. (2009) shows that the weight updates when training a neural network, with

softmax output activation functions, by minimising the cross-entropy error function is given by the following:

$$w_{2mk}^{(t+1)} = w_{2mk}^{(t)} + \alpha(y_k - \hat{y}_k)z_m \tag{3.14}$$

for the weights connecting the hidden layer and the output layer and

$$w_{1jm}^{(t+1)} = w_{1jm}^{(t)} + \alpha \sum_{k=1}^{K} w_{2mk}(y_k - \hat{y}_k)f'(z_{in-m})x_j \tag{3.15}$$

for the weights connecting the input layer and the hidden layer. These weight updates are exactly the same as those found for a neural network with linear output activation functions which is trained by minimisation of the sum of squares for error function. This is because of the combination of the error function and the output activation function which leads to the same equations for the weight updates when gradient descent is used (Bishop, 1995).

When used with a logistic activation function for the hidden units, (3.15) then becomes

$$w_{1jm}^{(t+1)} = w_{1jm}^{(t)} + \alpha \sum_{k=1}^{K} w_{2mk}(y_k - \hat{y}_k)z_m(1 - z_m)x_j \tag{3.16}$$

3.2.4 Way forward

The two neural networks just described will form the foundation of this analysis. As this is a first attempt at building a neural network, I will mainly fit these two types of neural networks to the data and from this do comparisons against the current approaches that are in use.

3.3 Simulations

Before we proceed with the business application, we will perform simulations to illustrate the different neural networks that will be used for the business application. The aim of these simulations will be to show how the neural network can be used for a regression and for a classification problem. We will also show the effect of the hidden nodes in the two-layer network. The programs that will be used for the simulations will be written in the open source software package R and can be found in the appendix. These programs are provided with a full set of comments for the interested reader.

3.3.1 Simulations illustrating the regression case

We will start by illustrating how a neural network can be used in the regression case. Data will be simulated from the following function:

$$y = 1 + \sin(0,5\,x) + \varepsilon \qquad \varepsilon \sim N\left(0;(0,2)^2\right) \tag{3.17}$$

The aim is to use this data to train a neural network that will give a predicted value $\hat{y}$ when the neural network is presented with an input x. This means that we want to train the network to give the best possible generalisation to unseen data.

From (3.17) we see that y is related to x through some function, say $h(\cdot)$, with added noise which is denoted by ε. The aim of fitting the neural network is to model the underlying generator of the data, which in this case is given by the function

$$y = h(x) = 1 + \sin(0,5\,x) \tag{3.18}$$

If we simulate 16 observations from the function given in (3.17) and plot these data points, together with the true underlying distribution, we obtain the scatterplot given in figure 3.3. From the scatterplot in figure 3.3 we see that our intent would be to fit a neural

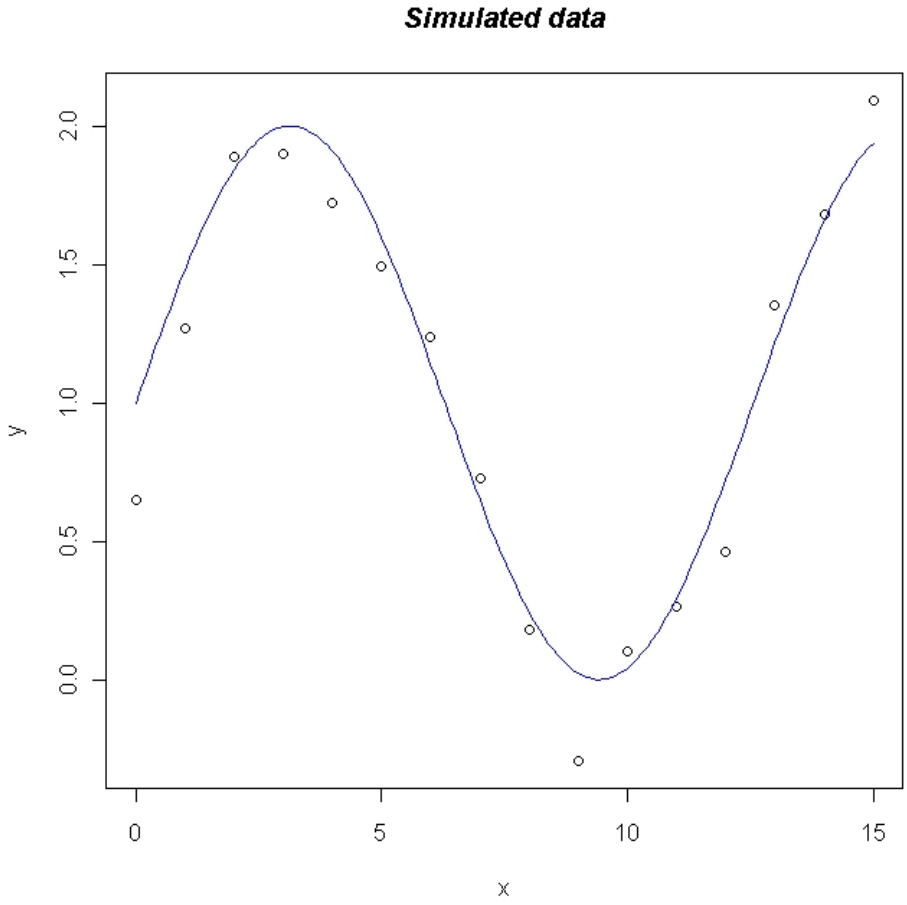


Figure 3.3: Data simulated from the function $y = 1 + \sin(0,5\,x) + \varepsilon$, where ε is a normal random variable with mean 0 and standard deviation 0,2 that is added to the y values. The blue line indicates the true underlying function, given by $h(x) = 1 + \sin(0,5x)$, that we want to model with a neural network.

network, using the simulated data points, that will closely model the underlying generator of the data given by the blue line. A neural network that will model the underlying generator of the data, denoted by the function $h(\cdot)$, closely will have the best possible generalisation to unseen data. This matter was discussed in more detail in section 2.4.4.

I will proceed by fitting a number of neural networks, each with a different number of

hidden nodes. These neural networks will be used to map an input x to an output y. This simulation exercise will be used to illustrate the function of the hidden nodes clearly. The neural networks that will be used for this purpose, will be two-layer backpropagation neural networks, with logistic hidden node activation functions and linear output activation functions. The neural network will be trained on all the simulated data points for 1,000,000 epochs. The inputs, x, will be standardised when the network is trained. For this illustration, I am not going to standardise the y values. The epoch with the lowest error, calculated on the training data, will be used to show the function of the hidden nodes in the neural network.

We will start by training a neural network, using the simulated data from (3.17), with one hidden node. The fitted neural network, the underlying generator of the data and the simulated data points are given in figure 3.4. We see in this plot, that the one hidden node is not enough to model the function $h(x) = 1 + \sin(0, 5x)$ accurately. A neural network with one hidden node can only model the logistic function and the best that the neural network can manage with this limited complexity is clearly illustrated in figure 3.17.

By increasing the number of hidden neurons to two, we again fit a neural network to the simulated data and this neural network is illustrated in fig 3.5. In this plot, we see that inclusion of the extra hidden node enables us to model one of the turning points in the function. Thus we can see that the extra hidden node increased the complexity of the function that was fitted. Intuitively we feel that if we include another hidden node, it should model the underlying function accurately.

We now proceed to increase the number of hidden nodes to three and repeat the simulation. The fitted neural network from this simulation is plotted in figure 3.6. We see from this plot that the neural network with three hidden nodes, accurately models the underlying distribution, especially considering that we have such a small sample of data

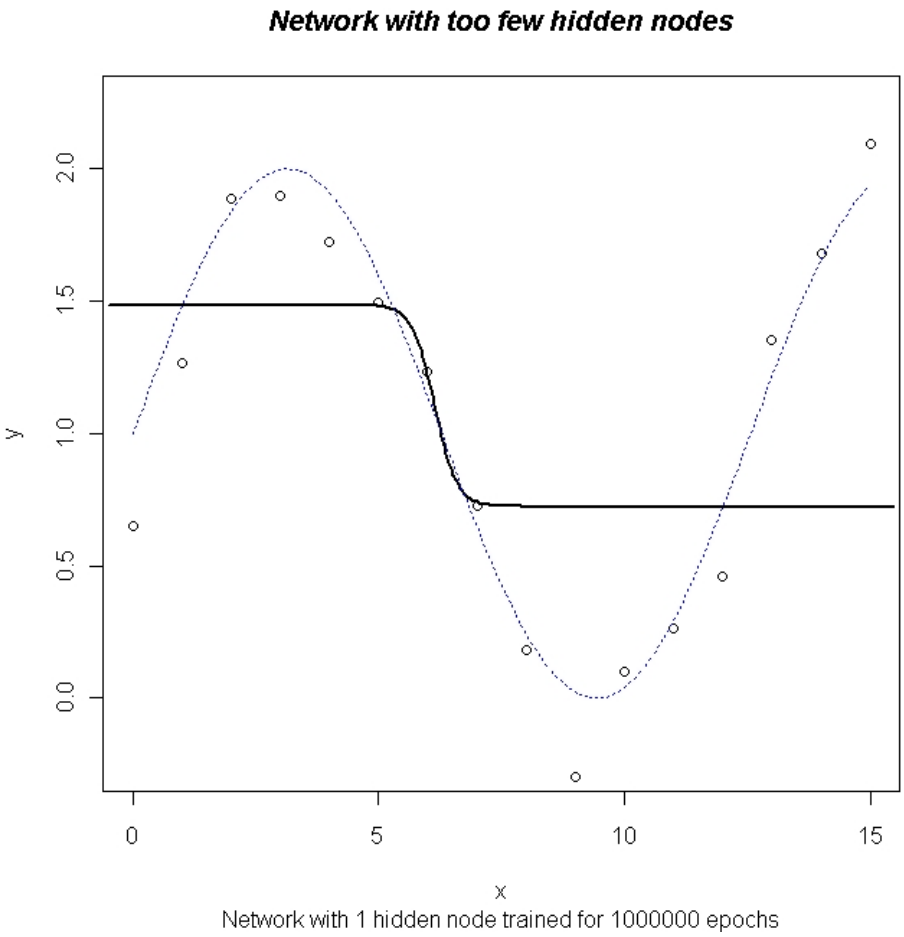


Figure 3.4: Plot of neural network with one hidden node.

points to train the network on.

We see that three hidden nodes will be the optimal number of hidden nodes to train the network on and that if we include more hidden nodes, the neural network will start to overfit the data and the neural network will be fitted to the noise in the data. To illustrate this point, a neural network with 20 hidden nodes is fitted to the simulated data. The fitted neural network is given in figure 3.7.

From the fitted neural network in figure 3.7 we see that this neural network overfits the data. We see that the large number of hidden nodes introduced too much complexity to the neural network and the neural network interpolates the data points, i.e. the neural

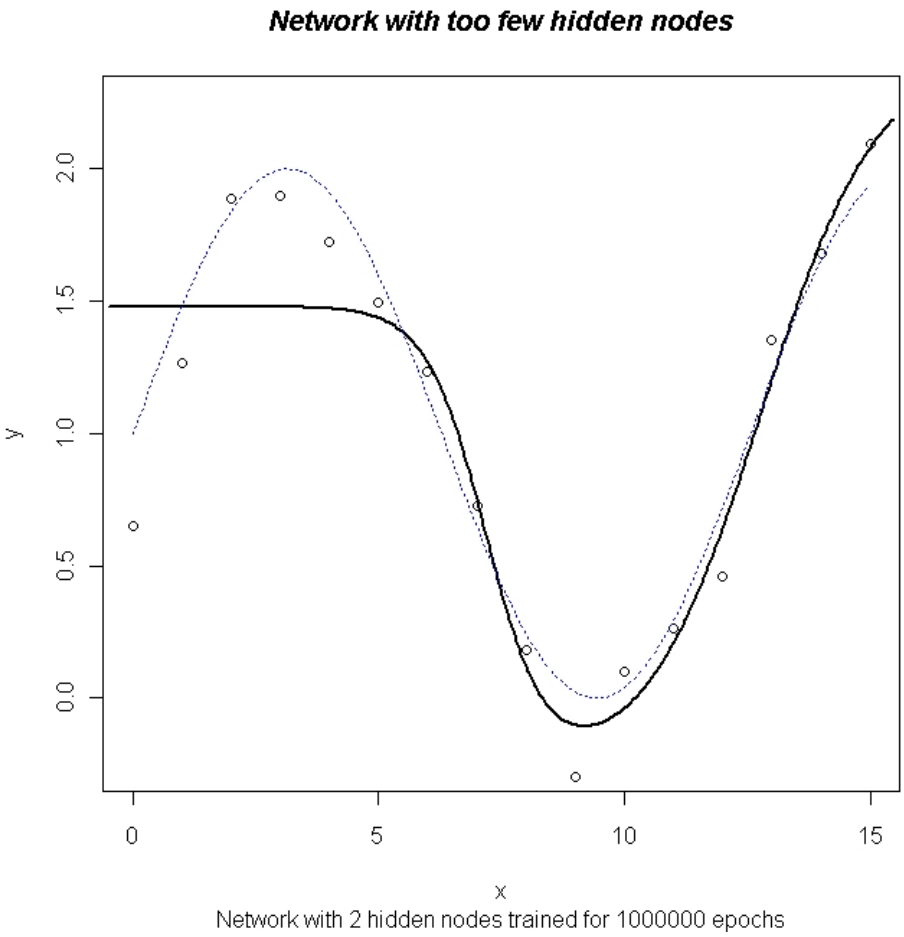


Figure 3.5: Plot of neural network with two hidden nodes.

network is fitted to the noise in the data implying that it will not provide good generalising capability to unseen data.

We see that the choice of hidden nodes is a difficult one, because if we have too few hidden nodes, the neural network may not be able to accurately model the underlying function as was shown in figures 3.4 and 3.5. On the other hand, if we have to many hidden nodes, the neural network starts to fit the training data, and hence does not model the underlying distribution of the data. This case is shown in figure 3.7. Thus we see from these simulations that the number of hidden nodes directly controls the effective complexity of the neural network. The question now is, how do we choose the number of hidden nodes in the neural network, such that the network will accurately model the

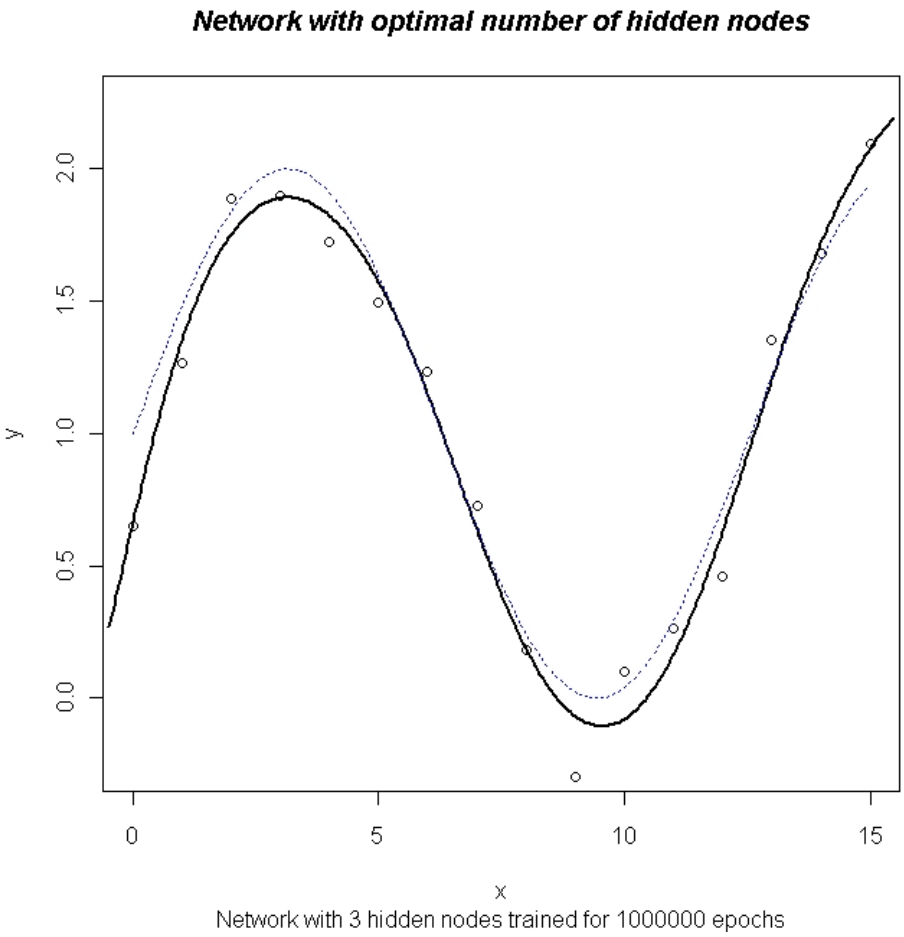


Figure 3.6: Plot of neural network with three hidden nodes.

underlying relationship in the data, and not model the sampled observations.

Through simulation we will show two easy ways, in which the effect of overfitting can be minimised. The above simulations were done on a small data set. Neural networks typically perform better when the data sets are relatively large. Large data sets also seem to limit the effect of overfitting as will be shown in the next simulation. The other factor which can be used for better neural network training is to divide the data into a modelling and a testing data set. The neural network is then trained on the modelling set, but after each epoch, the network is tested on the testing data set and the prediction error on this testing data set is then used to decide which neural network gives the best fit and will have the best generalisation capability.

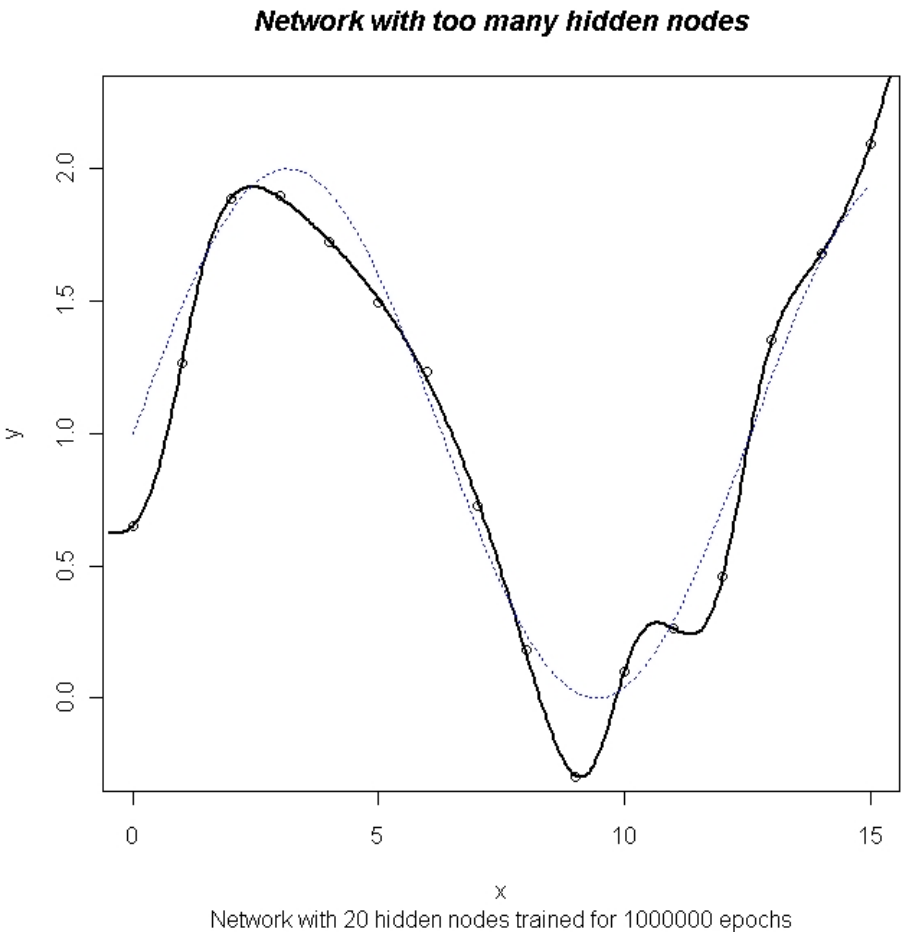


Figure 3.7: Plot of neural network with 20 hidden nodes.

An approach which seem to work relatively well when fitting a neural network is to split the data into a modelling and a testing data set and then train a neural network which have a relatively large number of hidden nodes but the neural network must be tested after each iteration on the testing set. The large number of hidden nodes ensures that the neural network will have the required complexity to accurately model the underlying relationship between the model inputs and outputs, while the concurrent testing on an independent testing set enables the learning algorithm to stop training before the neural network starts to overfit the training data.

To illustrate the effectiveness of this approach, we will again use simulations. The approach used in this simulation will be the approach that will be used for the business

application later on, because it is very difficult to determine the optimal number of hidden nodes which should be used with the real life data.

For this simulation, we will again use the small sample of 16 observations, simulated from the distribution given in (3.17). The aim is to illustrate that even with such a small sample, splitting the data into a modelling (80%) and testing (20%) data set provides benefit in limiting overfitting of the neural network. A neural network with 20 hidden nodes will be trained. The model will be trained on the modelling data set, but after each epoch, the model will be tested on the testing data set. The neural network which gives the lowest sum of squared errors on the testing set will then be used as the final neural network. The fitted network, together with true underlying relationship and the modelling and testing observations are given in figure 3.8.

We see from figure 3.8 that even with so few datapoints, splitting the data into a modelling and a testing data set provides an advantage as this seem to reduce overfitting of the neural network. The fitted neural network gives a very reasonable approximation to the true relationship between the inputs and outputs, especially considering the size of the data set. For interest sake I will increase the number of obsevations to 200, again simulating from the distribution given in figure 3.17. The data will again be split into a 80% modelling set and a 20% testing set. The fitted neural network is plotted in figure 3.9.

Figure 3.9 illustrates that the number of observations increases the accuracy of the neural network to model the underlying relationship between the inputs and outputs. It can be seen from these simulations that the approach of fitting a neural network with a relatively large number of hidden neurons, and incorporating the testing on an independent data set into the learning process, gives good performance as it ensures that the neural network has enough complexity to model the underlying relationship between the inputs and outputs

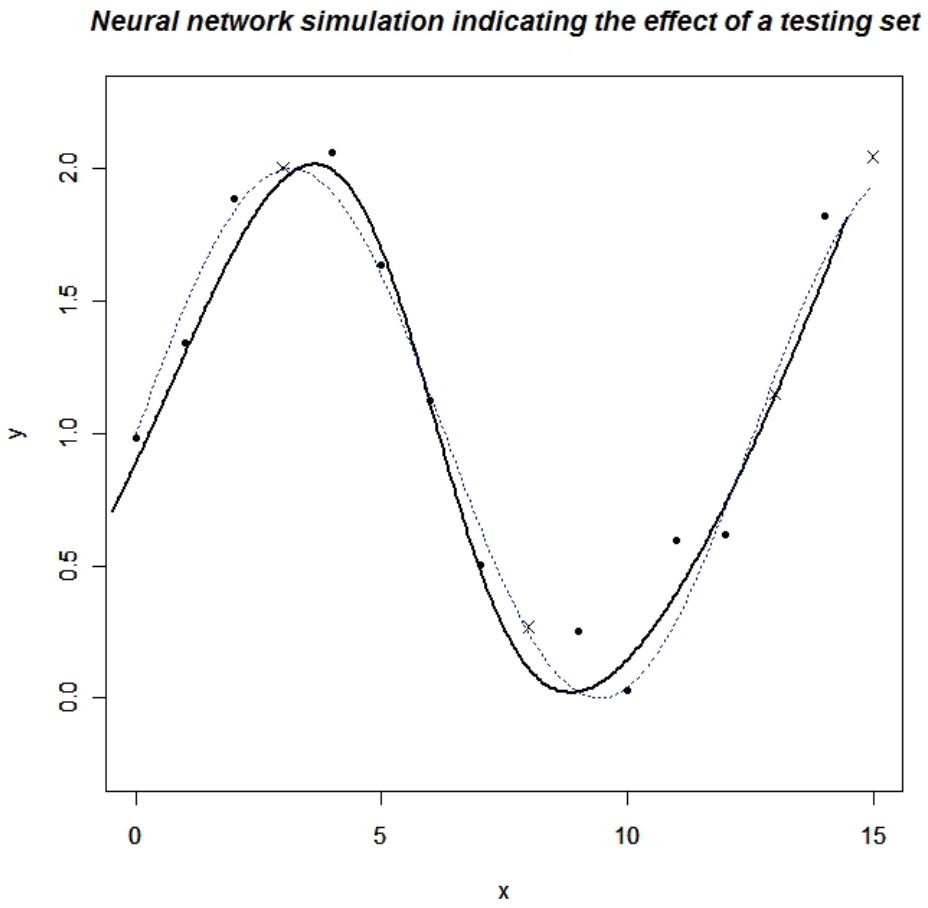


Figure 3.8: Plot of fitted neural network through simulation to indicate the effect of an independent testing set. The neural network has 20 hidden nodes and the neural network with the smallest prediction error on the testing set is used in the plot (indicated by the thick line). The dotted line indicates the true relationship between x and y. The dots indicate the modelling data and the crosses the testing data.

accurately while minimising the effect of overfitting.

3.3.2 Simulations illustrating the classification case

In the previous section, a neural network was used in a regression context. This means that the neural network was used to model the relationship between input variables and a continuous output variable. Simulations were performed to assess the effect that the number of hidden neurons have on the modelling in a regression context. The same

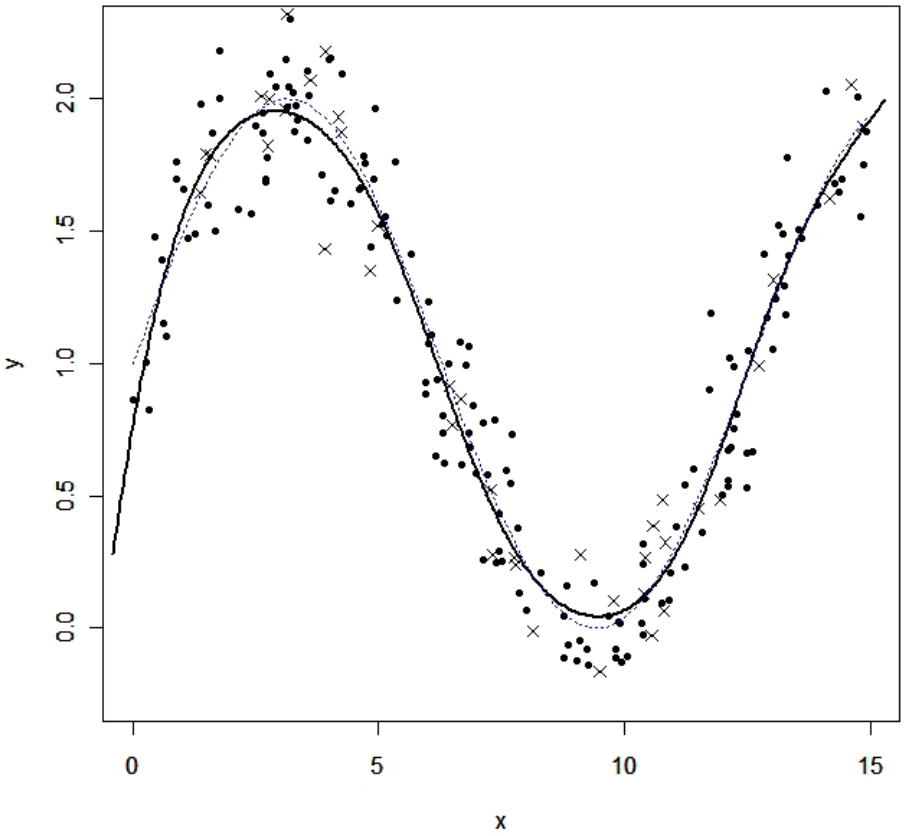


Figure 3.9: Plot of fitted neural network through simulation to indicate the effect of an independent testing set. The neural network has 20 hidden nodes and the neural network with the smallest prediction error on the testing set is used in the plot (indicated by the thick line). The dotted line indicates the true relationship between x and y. The dots indicate the modelling data and the crosses the testing data.

exercise will now be repeated in a classification framework.

Recall from section 2.4.3 that when using a neural network for classification, the objective is to classify each observation into one of a distinct number of categories. Essentially, when a neural network is used for classification, the neural network builds classification areas, and each observation is then classified into a group corresponding to the classification area into which that particular observation falls. The effect of the number of hidden nodes in the trained neural network on these classification boundaries will be shown by using simulation. Again we will also show that the most effective way of training a neural

network is by splitting the data into a modelling and testing data set and incorporating the testing with the training algorithm.

To start, we will simulate data from 5 groups, each group containing 100 observations. These data points are basically simulated from 5 different bivariate normal distributions, each with its own mean and covariance structure. A plot of the simulated data is given in figure 3.10. From this plot, we can see that there is overlap between the 5 groups, indicating that perfect seperation between the groups will not be possible.

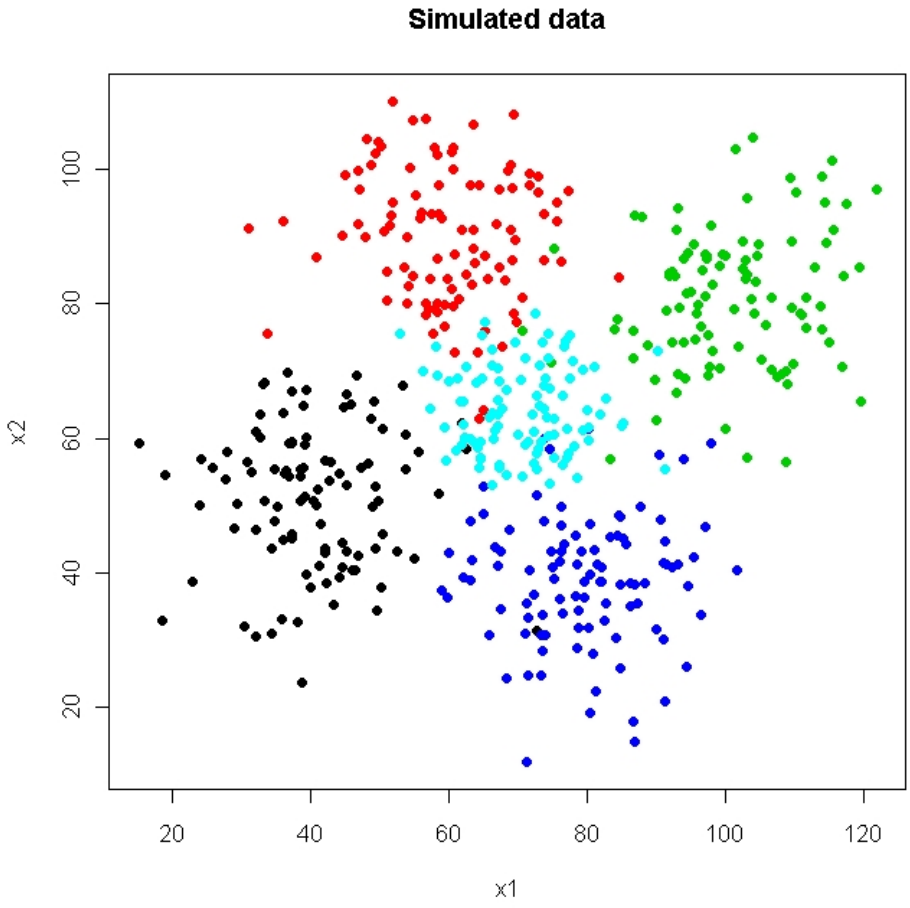


Figure 3.10: Data simulated from 5 different bivariate normal distributions. The colour of the observation indicates the group to which the observation belongs to.

These simulations will be conducted by using a neural network that is set up for classification. The inputs to this neural network will be standardised and the outputs will be

coded according to the 1-of-K coding scheme, which means that for this specific example, the neural network will consist of 3 inputs, which includes a bias term, and 5 outputs corresponding to each of the 5 groups in the data. The hidden layer activation function that will be used will be the logistic activation function and the output activation will be the softmax function, which guarantees that the outputs can be interpreted as posterior probabilities.

We start by training a neural network with one hidden node for a 100,000 epochs and choose the neural network which gave the lowest missclassification rate measured on the training data itself. The fitted classification areas from this neural network is presented in figure 3.11.

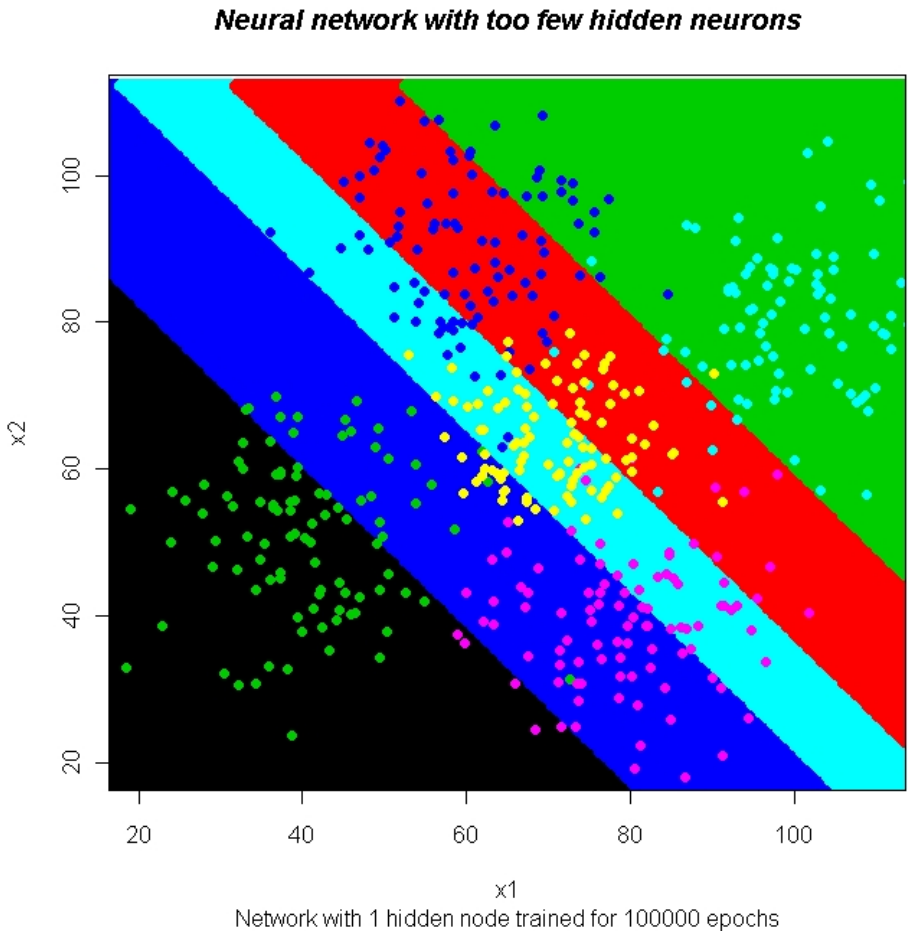


Figure 3.11: Classification areas fitted by neural network with one hidden node

From figure 3.11 we can clearly see that the one hidden neuron that is used in this neural network does not allow for sufficient flexibility and that only linear classification boundaries can be mapped by this neural network. This leads to a high rate of misclassifications. It is clear from figure 3.11 that one hidden neuron does not allow for enough complexity to accurately classify the observations and that we can lower the misclassification rate by increasing the number of hidden neurons.

For the next simulation, the number of hidden neurons will be increased to 2 to see the effect on the classification boundaries. The fitted classification areas for this simulation together with simulated data is presented in figure 3.12. We see that two hidden nodes offer a big improvement over using just one hidden node, but it still does not look like the optimal classification areas.

Increasing the number of hidden neurons to 3, we find the neural network that is plotted in figure 3.13. Three hidden nodes still does not seem to be sufficient, so we will proceed to increase the number of hidden neurons until we get an optimal fit. The results of neural networks fitted with four, five, six, seven and eight hidden neurons and the classification boundaries that each of these neural networks fits can be found in figures 3.14, 3.15, 3.16, 3.17 and 3.18 respectively.

,

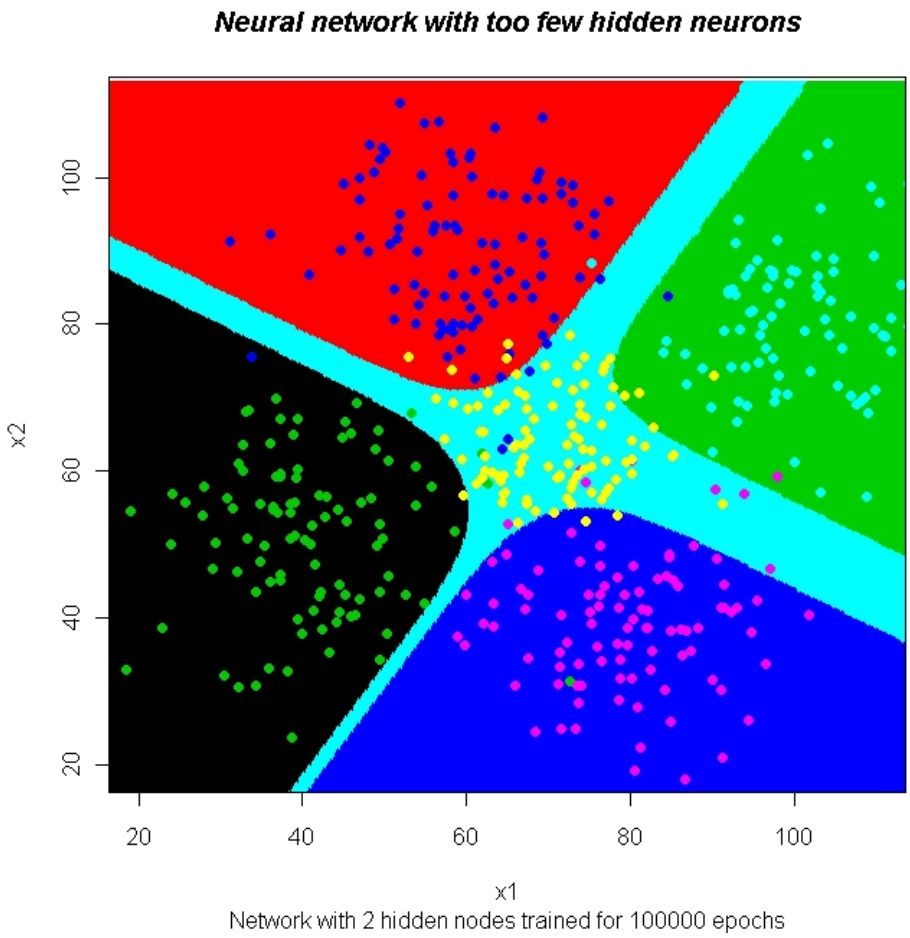


Figure 3.12: Classification areas fitted by neural network with two hidden nodes

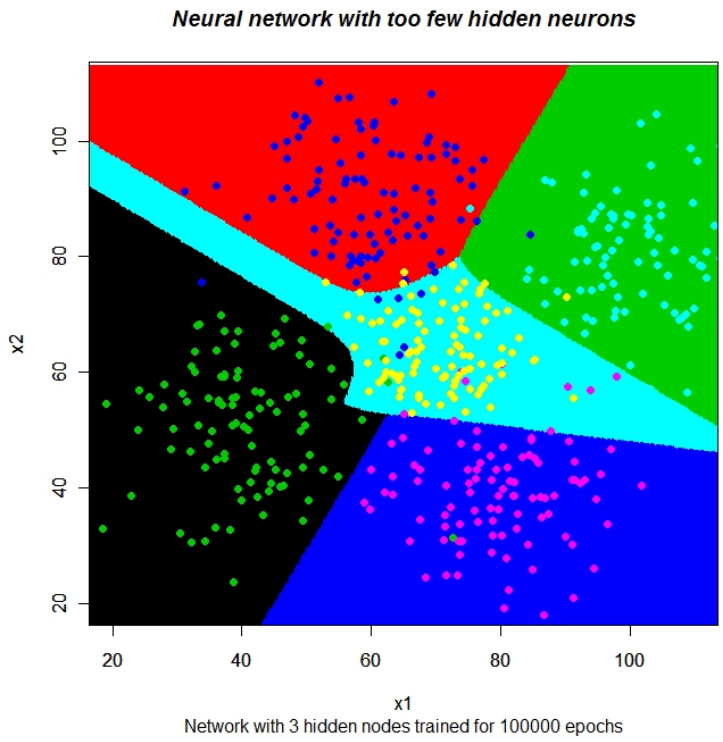


Figure 3.13: Classification areas fitted by neural network with three hidden nodes

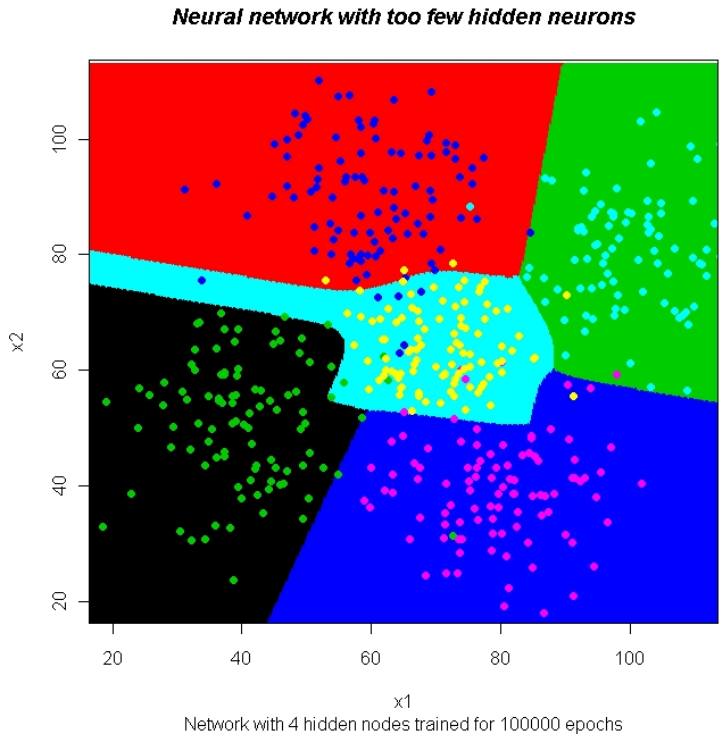


Figure 3.14: Classification areas fitted by neural network with four hidden nodes

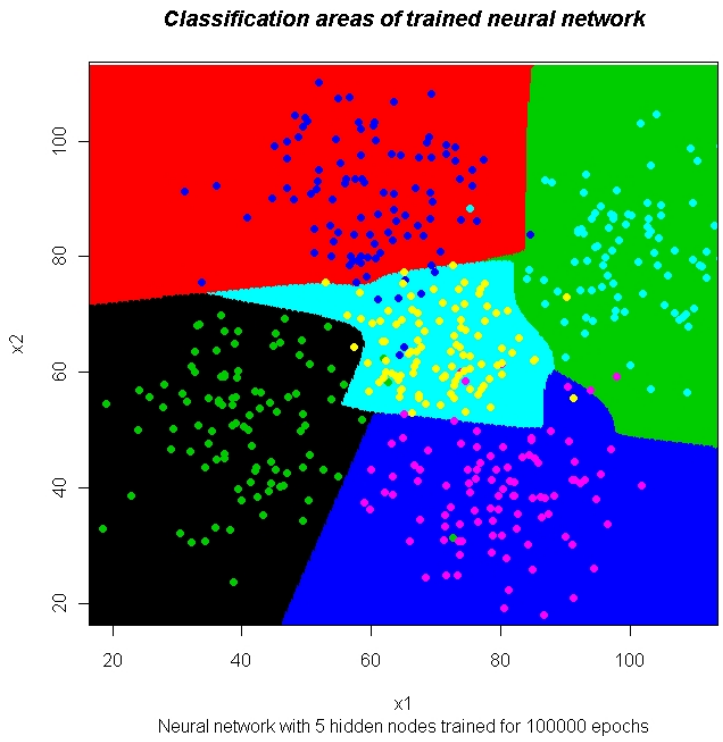


Figure 3.15: Classification areas fitted by neural network with five hidden nodes

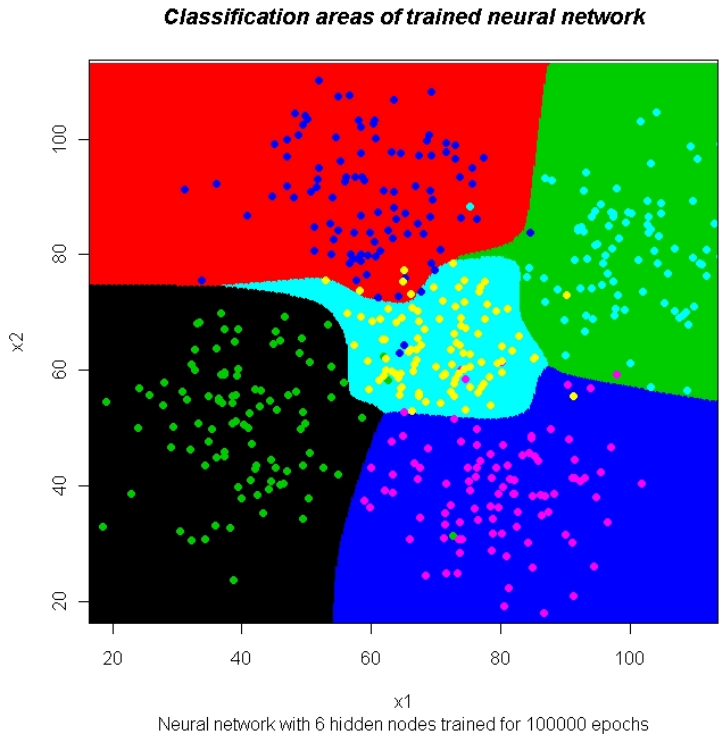


Figure 3.16: Classification areas fitted by neural network with six hidden nodes

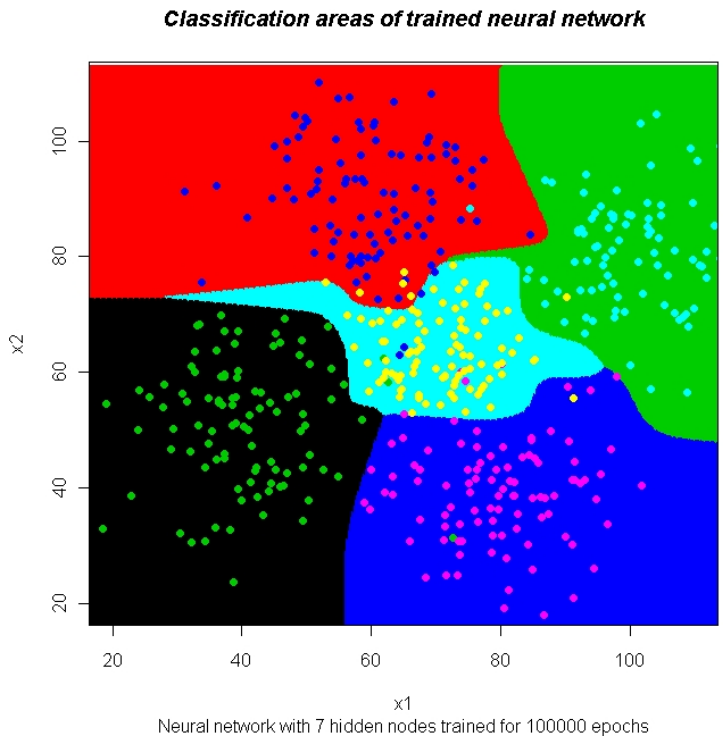


Figure 3.17: Classification areas fitted by neural network with seven hidden nodes

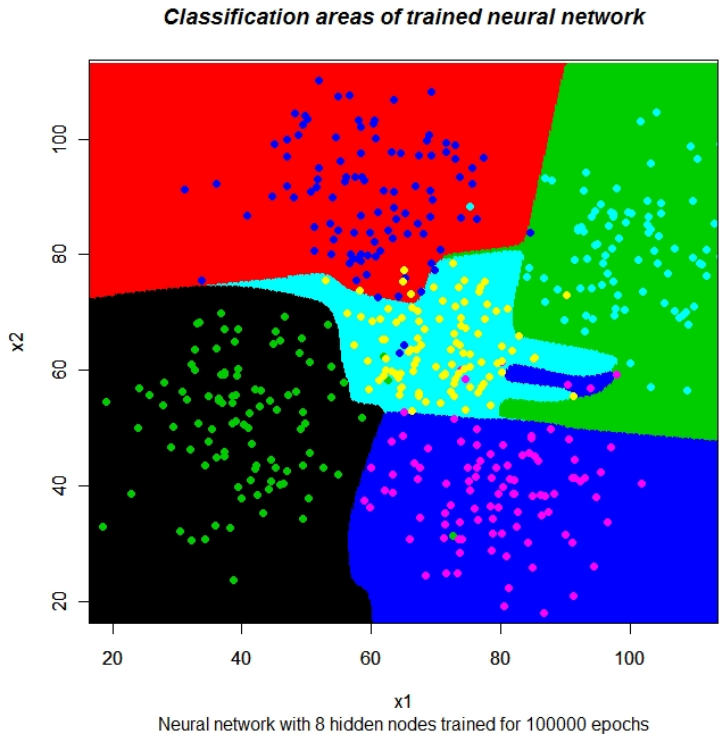


Figure 3.18: Classification areas fitted by neural network with eight hidden nodes

From the neural networks that was fitted and presented in figures 3.14, 3.15, 3.16, 3.17 and 3.18 we see that the optimum number of hidden nodes seem to be around five or six. Three or four hidden nodes seem to be too few and seven or eight hidden nodes seem to be too many, as the neural network starts to overfit the training data. For interest's sake, we will also increase the number of hidden nodes to 50 to illustrate the results. This will serve as a caution, not to just increase the number of hidden nodes because even though this network will have lower prediction error on the training set, it will overfit severely and not provide good generalising ability. The result of this simulation can be found in figure 3.19. From the plot of figure 3.19, we see that the classification boundaries become too specific to the current training data, indicating the noise in the data is being modelled instead of the true underlying distribution.

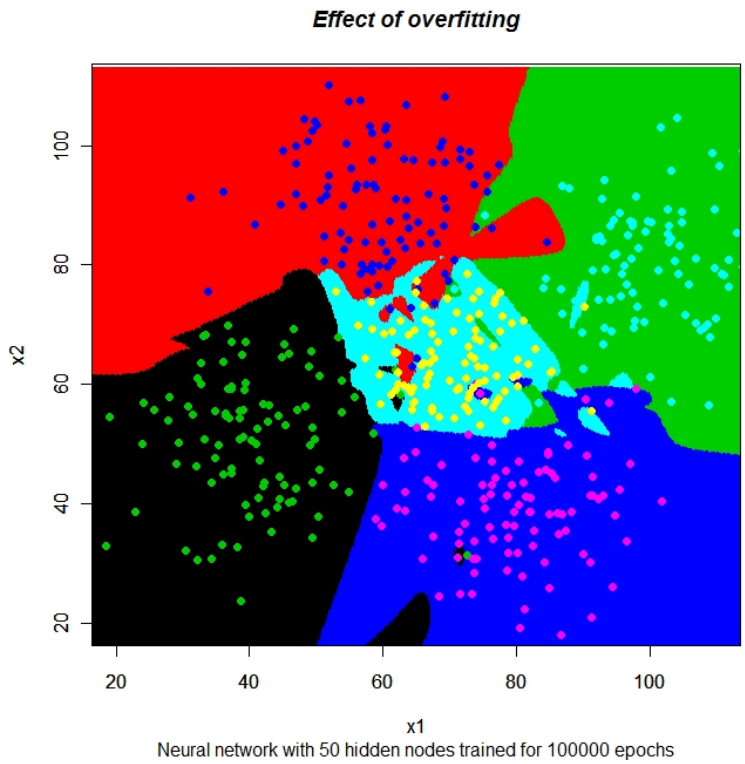


Figure 3.19: Classification areas fitted by neural network with fifty hidden nodes

In the classification context, the need for a testing set is even higher and will provide the best possible fit of a neural network. Similar to what was explained in section 3.3.1, the data will be split into a modelling and testing set. After each iteration of the learning

algorithm on the modelling data, the misclassification rate will be calculated on the testing set, and the weights from the epoch with the lowest misclassification rate on the testing set will be used as the final neural network. For this simulation, we will use a neural network with 20 hidden nodes, since the testing set will limit the overfitting of the neural network. The result of this simulation is presented in figure 3.20.

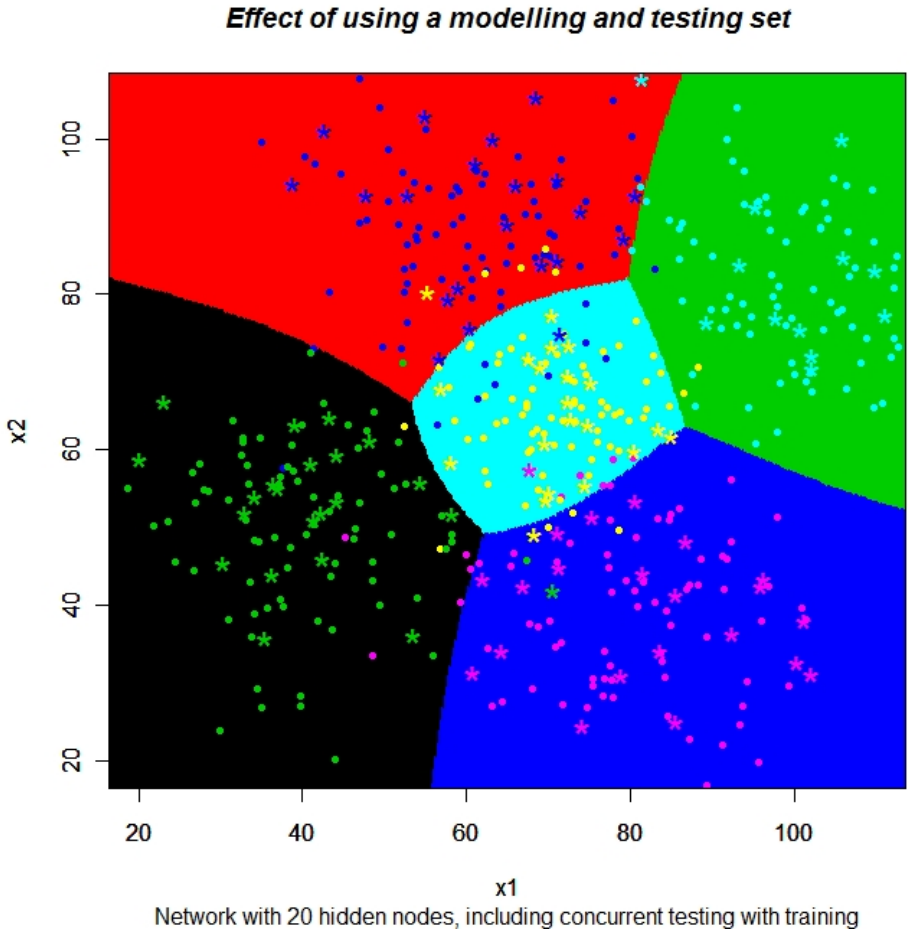


Figure 3.20: Classification areas fitted by a neural network that uses a modelling and testing set. The network was fitted using 20 hidden neurons and the weights which gave the lowest misclassification rate on the testing set were chosen. The dots indicate the modelling data and the * symbol indicate the testing observations.

From 3.20, we can see that this neural network provides a clear improvement over the previously fitted neural networks. This approach of splitting the data into a modelling and testing set and conducting the testing concurrently with the training of the neural network seem to be very effective and this will be the approach that will be followed on

the business problem that is described in the remainder of this chapter.

3.4 Description of the problem

In this section we will give an in-depth description of the problem at hand. A thorough description is needed to fully understand the problem. This is a complex problem which is not currently handled by a statistical model but by a rules-based approach. The aim of the neural network would be to see if we can build a model with comparative performance to what we currently achieve with the set of rules. We will start by discussing where the data originates from and then proceed to discuss the data set that will be used.

Lightstone is a company which employs an automated valuation model (AVM) to provide estimates for the values of residential properties in South Africa. Lightstone purchases data from the deeds office of all sales of properties in South Africa. This data is augmented by data obtained from the banks, because there is a delay in the data from the deeds office. This data is used by Lightstone to build its models and to provide an automated valuation of a property. Basically there are two main models, namely a repeated sales (RS) model and a comparable sales (CS) model which, together with various other measurements, are used in order to determine a final prediction for a property. I will not describe the whole process of how the repeated sales model and the comparable sales model work, but will rather give an overview and references of these models and their methodologies which the interested reader can consult.

3.4.1 The repeated sales model

In the repeat sales approach, house prices in a given location are determined by modelling the inflation between the last two sales of the particular property. The current value of the property is then derived by using the estimated coefficients from the regression to inflate the property's value from the previous transacted price to the current price.

The fundamental benefit of the repeat sales approach is that it captures the inherent characteristics of a property the moment that property trades in the market. For example, in the same area, a big house with good finishes will cost more than a small house with average finishes. Thereafter, the model will differentiate between the big house and the small house, without the need for any information on size or finishes of the house (which is called hedonic data).

The repeat sales model is based on a regression method, wherein the log of the price inflation between any two sale prices of a property is modelled as a function of the period between the two transactions. In this model, it is assumed that prices are driven solely by inflation (for example, we assume that two similar properties alongside each other will have the same growth in value; we do not allow for the possibility that the one owned by a pensioner may be more poorly maintained than the one owned by a younger person), even though the property inflation rate may change from year to year. Different models are tailored for separate market segments, which typically have different inflation patterns.

The repeat sales model which is fitted to the data will now be defined:

Notation

n number of properties in the area which have been sold more than twice during the time period under consideration

t_{i1} year in which the first sale of property i took place

t_{i2} year in which second sale of property i took place

$y_{t_{i1}}$ first sale price of property i

$y_{t_{i2}}$ second sale price of property i

r_k property inflation rate in year $k, k = 1, \ldots, T$ where $k = 1$ denotes the start of the period under consideration (e.g. 2003) and $k = T$ the end of this period (e.g.. 2006).

The model

Assuming that prices are driven solely by inflation, $y_{t_{i1}}$ and $y_{t_{i2}}$ are related by the formula:

$$y_{t_{i2}} = \prod_{k=t_{i1}}^{t_{i2}-1} (1 + r_k) y_{t_{i1}}$$

Then

$$Z_i = \log\left(\frac{y_{t_{i2}}}{y_{t_{i1}}}\right) = \sum_{k=t_{i1}}^{t_{i2}-1} \log(1 + r_k) = \sum_{k=t_{i1}}^{t_{i2}-1} a_k$$

This means that our regression model will take on the following form:

$$Z_i = \sum_{k=t_{i1}}^{t_{i2}-1} a_k + \varepsilon_i = \sum_{k=1}^{T} a_k \delta_{ik} + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where

$$\begin{aligned} \delta_{ik} &= 1 \text{ if } k \,\epsilon\, t_{i1}, \ldots, t_{i2} - 1 \\ &= 0 \text{ otherwise} \end{aligned}$$

Finally, recognising the fact that the first and last sales are unlikely to take place on the first day of the year (as the above formulation implies) and defining δ_{ik} instead to be the fraction of year k between the first and second sales of property i, then the above formula for Z_i will still apply, the only difference being that now $0 \leq \delta_{ik} \leq 1$. (In fact δ_{ik} could only be different from 0 or 1 in the years when the property transacted.) Note that this model is in the form of a multiple linear regression model for $Z_i = log\left(\frac{y_{t_{i2}}}{y_{t_{i1}}}\right)$ in terms of the predictors δ_{ik} , $k = 1, \ldots, T$, over the n observations.

Model with an intercept

It is found in practise that the model which includes an intercept term usually produces more accurate predictions, particularly for properties where the first and second sales are close, say less than two years apart. In this case the model becomes:

$$Z_i = a_0 + \sum_{k=1}^{T} a_k \delta_{ik} + \varepsilon_i \qquad i = 1, \ldots, n$$

Parameter estimation

Two approaches can be taken towards estimating the parameters a_k , $k = 0, 1, \ldots, T$ in the model:

1. Ordinary least squares, which assumes the error terms ε_i all have the same variance.

2. Weighted least squares. This is based on the assumption that the errors follow a normal diffusion process (i.e. the errors in the unlogged model follow a lognormal diffusion process), which in turn implies that the error variance is proportional to the period between the two sales. Thus, for example, sale prices that are four years

apart will have half the weight of sale prices that are two years apart in the fitting of the model.

Note on the data set used for fitting the model

In order to use the repeated sales model, only transactions which have transacted at least twice can be included into the modelling data set. Lightstone uses only records where the most recent sale (second sale) of the property was within the last 21 months. One can use all the records which had two or more sales, but since we are trying to give a prediction of the value of the property currently, it was found that by only using the most recent data, the regression is skewed towards current trends which gives us a more accurate prediction of what is currently happening in the market.

The model also assumes that the properties which have transacted, sold for a market related price and that the property have not undergone significant changes. Lightstone have developed various methods to flag the records which do not meet these requirements and these records are not used in the modelling data set.

Modelling Segments

Lightstone have developed segments into which properties are classified. It is found that properties which fall into different segments experience different rates of inflation and that is why the following modelling segments are used and a repeated sales model built for each of these modelling segments:

1. A: Township

2. B: Very poor and poor
3. E1: Freehold not too poor metro
4. E2: Freehold not too poor non-metro high activity
5. E3: Freehold not too poor non-metro low/med activity
6. F: Sectional title not too poor
7. G1:Freehold comfortable metro
8. G2: Freehold comfortable high activity
9. G3: Freehold comfortable non-metro low/med activity
10. H: Sectional title comfortable
11. I: Freehold wealthy
12. J: Sectional title wealthy
13. K: High density
14. L: Estates
15. Z1: Freehold unclassified properties
16. Z2: Sectional title unclassified properties

Prediction from the fitted model

Denote the predicted log-ratio for a property, based on this model, by

$$\hat{Z} = \sum_{k=1}^{T} \hat{a}_k \delta_k$$

, where the $\delta_k, k = 1, \ldots, T$ are determined by the year of its previous sale. A prediction of the value of the property at t_2, when that same property transacted for an amount y_1 at time t_1 is then given by

$$\hat{y}_2 = y_1 \exp(\hat{Z}) = y_1 \exp\left(\sum_{k=1}^{T} \hat{a}_k \delta_k\right)$$

This is the basic theory for the repeated sales model to provide a prediction for a property. For information on repeated sales methodology the interested readers can consult Bailey et al. (1963); Wang and Zorn (1997); Meese and Wallace (1997).

3.4.2 Comparable sales model

In the comparable sales approach, the property value is determined from current values of 'comparable' neighbouring properties, based on historical sales indices linked to current values. The comparable sales approach is built on three data sets:

- Freehold properties
- Sectional title properties
- Freehold properties in "residential estates"

The data set for Freehold and Sectional title carries information on three levels:

- For the specific Enumerator Area (EA) /Sectional title/Estate for (inflated) sales for the last two years
- For the specific EA /Sectional title/Estate for (inflated) sales for the last five years

- If fewer than five sales have occurred in the last two years, then the data for the EA is pooled with the closest EA of the same wealth segment. For Sectional Title properties, the data for the scheme is pooled with the data for other schemes in the same EA. No pooling takes place for Estates, beyond the Estate in which the subject property is located.

The data sets contain the distribution of sales prices (inflated to reflect current values using the coefficients developed in the repeat sales model), and property sizes, as well as the mean and median rand per square meter calculated over all the sales in the EA/Sectional Title/Estate/Pooled group.

Although the comparable sales tables are based on the deeds data it is tested on the mortgage application data to establish the accuracy of the predictions.

Rules for making a prediction

The tables are used to derive a prediction for a specific property according to the following set of rules:

Sectional Title Properties

In general, sectional title prices are dependent on the erf size of a unit within a sectional scheme, with larger units selling for more and smaller units selling for less. Thus, a comparable sales (CS) prediction for sectional title units is calculated by multiplying the unit's erf size with the average or median [1] Rand per sqm for that particular sectional

[1]The mean is used when it lies between the first and third quartiles of the inflated purchase price distribution, otherwise the median is used. This logic is used throughout.

scheme. In the case where a unit's erf size is unobtainable, either the mean or median price of units in that sectional scheme is used as the predicted value for that unit.

Rules determining which set of information to use:

- If the number of transactions in the last two years within the same sectional scheme is more than three, then this distribution will be used
- If the above is not satisfied, and the number of transactions in the last five years within the same sectional scheme is higher than three, then this distribution will be used
- If the above points are not fulfilled, and the number of transactions in the last two years for all the sectional schemes within the same EA is greater than three, then this distribution will be used
- If there are more than three transactions in the last two years for all the sectional schemes within the same EA and one adjacent EA (next to the one in which the subject property is located) of similar profile EA (i.e. same wealth classification – wealthy, poor, etc.) and the above criteria are not satisfied, then this distribution will be used
- If the above points are not fulfilled, and the number of transactions in the last two years for all the sectional schemes within the same EA and two adjacent similar EAs is greater than three, then this distribution will be used
- If the above points are not fulfilled, and the number of transactions in the last two years for all the sectional schemes within the same EA and three adjacent similar EAs is greater than three, then this distribution will be used
- If the above points are not fulfilled, then regardless of the number of transactions, the distribution of purchase prices for all the sectional schemes within the sub-place

for transactions in the last two years will be used.

Freehold properties

Since freehold property (FH) prices are more variable and not as dependent on the property's size as sectional title properties, the first quartile, mean, median or third quartile inflated price for properties within the closest possible area is most often used as the predicted value. The exception to this is when variation in purchase prices is closely linked to the variation in erf sizes; a rand per sqm calculation will then be used, provided the property's erf size is not missing.

Rules determining which set of information to use:

- If the number of transactions in the last two years within the same EA or surrounding similar EAs, if necessary, is five or more, then this distribution will be used.
- If the above is not satisfied, and the number of transactions in the last five years within the same EA is five or more, then this distribution will be used.
- If the above points are not fulfilled, and the number of transactions in the last two years within the same EA and adjacent similar is five or more, then this distribution will be used.
- If the above points are not fulfilled, the distribution of purchase prices for transactions in the last two years within the same EA and two adjacent similar EAs will be used.
- If the above points are not fulfilled, the distribution of purchase prices for transactions in the last two years within the same EA and three adjacent similar EAs will be used.

- If the above criteria are not satisfied, then regardless of the number of transactions, the distribution of purchase prices for transactions in the last two years within the sub-place (SP) will be used. A sub-place is an area which is bigger than an EA but smaller than a suburb.

- The previous sales price of the property is being compared with the price distribution for that year, to establish whether the property should receive a comparable sales estimate of the current price distribution's midpoint, or its quarter point or its three quarter point.

  For example, if in an EA a property sold in 2008 at a price which were in the lower quantile of all sales for FH properties in that EA in that year, then the Q1 comparable sales price gets allocated to the current sale. Were the previous price to have been in the top quantile of prices the Q3 price would have been allocated as the CS prediction for the property. Where there were not sufficient sales in the year of the previous sales, all sales are inflated to current values and the relative position of the property in the overall price distribution is used to determine whether Q1, mean or median or Q3 CS value should be used.

Freehold properties within estates

The estimation of a freehold property's value within an estate is derived in the same way as for freehold properties outside of estates.

Rules determining which set of information to use:

- If the number of transactions within the same estate in the last two years is more than three, then this distribution will be used

- If the above is not satisfied, and the number of transactions in the last five years within the same estate is more than three, then this distribution will be used
- If the above two points are not fulfilled, and the number of transactions in the last two years for freehold properties within the same EA (circling out to surrounding and similar EAs if volumes are insufficient) is greater than three, then this distribution will be used
- If the above three points are not satisfied, then regardless of the number of transactions, the distribution of purchase prices for all freehold properties within the sub-place will be used.

The rules stated were well tested before it was implemented and were found to be optimum when providing a prediction for a property from the comparable sales model.

3.4.3 Combining of predictions

After obtaining the predictions from the repeated sales model and the comparable sales model Lightstone proceeds to combine the predictions from both these models in a statistical way. This combining helps to overcome some of the problems that are experienced when the models are used separately. To describe the advantages of combining these two models let's first look at the advantages and disadvantages of each model separately.

We will start with the advantages of the repeated sales model:

- If no hedonic data, i.e. data on the characteristics of the property, is available, the repeat sales model would still be able to provide a prediction since the characteristics of a house is inherently captured in the purchase price. This means that it is

not necessary to have the characteristics of the house, because a bigger house will transact for more than a smaller house in the same area, if the transactions are arms-length deals and the property sold for a market related price.

- The approach models the real inflation based on properties that have sold more than once during the modelling period. Other approaches (e.g. comparing the average price in an area from one time period to the next to calculate the inflation) are based on the assumption that the properties transacting in each period are representative of the properties in the areas. This assumption is often incorrect. Consider the example where an established area holds 50 properties that churn at 10% a year (i.e. five transactions per annum). Assume that the properties in year 1 had an average price of R500 000 and that the properties in the area were appreciating at 10% per annum. Then, in year 2 a new high-end development is completed with ten properties selling for R1million per unit each. This would obviously skew the inflation calculation substantially.

The advantages of the comparable sales model is:

- When we know which area a property falls into, we can obtain a prediction from the comparable sales model, even though there may be no previous purchase price for that property.

Some of the disadvantages to using these two models separately are

- Repeat sales model
  - A prediction can only be generated if there was a previous sale on the property since we need a price of the property to inflate using the estimated coefficients.

  - If the price that we inflate using the estimated coefficients from the model is wrong, the prediction from the repeated sales model will be inaccurate. This can easily happen when there are capture errors in the data or if the property has sold previously for a price which was not market related.

- Comparable sales model

  - The prices of properties in some areas may have a bimodal distribution and care should be taken when applying a comparable sales approach to properties falling in such areas as the comparable sales from the more expensive properties in the area may average out those prices from the less expensive homes in the area providing an unreliable prediction from the comparable sales model.
  - The comparable sales approach alone does not allow for sufficient differentiation of prices within a geographical area.

As we can see, if we can combine these two estimates in an effective way, we can try to negate the disadvantages from using each of these models separately and more importantly, provide the most accurate possible estimate of the current value for a property. Before we can show how Lightstone combines these estimates we need to describe what the safety and accuracy score of each prediction is.

Safety and accuracy scores

Every prediction (from both the repeat sales model and the comparable sales model) also has two diagnostic scores attached to it:

- The accuracy score, which is the probability of the prediction being within 20% of the actual value of the property.

- The safety score, which is the probability that the prediction does not exceed the actual value of the property by more than 10%.

These accuracy and safety scores are built using logistic regression models using mortage application data from the various banks. Six logistic models are used to derive the following scores:

- Repeat sales accuracy score
- Repeat sales safety score
- Comparable sales accuracy score
- Comparable sales safety score
- Combined prediction accuracy score.
- Combined prediction safety score.

Combining Predictions from the two Models

Suppose one has both repeated sale and comparable sales predictions for the price of a house, each one with its own accuracy score, and you want to combine them in an optimal way, in order to maximise the accuracy and safety score for the combined prediction (COMB).

I will not present the proof here, but if we assume that we have two predictions, in this case one from the repeated sales model, say X_1 with standard deviation σ_1, and one from the comparable sales model, say X_2 with standard deviation σ_2, it can be shown that the optimal combination of the two predictions is:

$$X_{comb} = \left(\frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2}\right) X_1 + \left(\frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}\right) X_2$$

with standard deviation

$$\sigma_{comb} = \left(\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}\right)^{-\frac{1}{2}}$$

The accuracy and safety score for the combined prediction can also be obtained by using logistic models as was discussed earlier.

3.4.4 Rules For Determining The Choice Of The Final Prediction

Lightstone most often uses the combined prediction as an estimate of the value of a property since the combined prediction tends to give a more accurate prediction because it is a weighted combination of two predictions rather than a single prediction. There are however times, when this prediction can be deemed unreliable and the repeated sales prediction or the comparable sales prediction is closer to the actual value of the property. Another problem can be that one of the two predictions from which the final prediction is derived, can be unreliable which means that the combined prediction may also not be as accurate. For example, we may have that the repeated sales prediction may be unreliable but the prediction from the comparable sales model is reliable. In this instance, we may not want to use the combined prediction because of the influence from the unreliable repeated sales prediction and would rather like to use the comparable sales prediction as the estimate for the value of a property. Various rules are also used to determine if a particular transaction was a arms-length transaction. This will give an indication whether the price, that we use to inflate off using the repeat sales coefficients, is a reliable price and hence if the repeat sales prediction is an accurate estimate.

Lightstone has developed a set of rules which aim to pick the correct prediction from the three possible predictions at hand (the repeated sales model prediction, comparable sales model prediction and also the combined prediction). These rules will not be given here but the basic idea of these rules is to determine whether a specific prediction is reliable by looking at the standard error of the prediction, the accuracy score and the safety score of the prediction and also the relative sizes of these predictions with regards to each other in order to pick the best possible prediction.

3.4.5 Aim

To summarise, the current approach that Lightstone employs is that we obtain a repeated sales prediction from the current data and together with this prediction we also obtain a standard deviation, and a safety and accuracy score. For the comparable sales model, we also have a prediction, standard deviation for this prediction, and then a safety and accuracy score attached to the prediction. These two predictions are then combined to form a combined prediction, together with a standard deviation, safety score and accuracy score. This means that there are basically three estimates for the value of the property which can be used. Lightstone then uses rules to determine which of these three predictions is the best prediction for the value of the property.

There are however two problems with this approach. The first is in the way the predictions from the repeated sales model and the comparable sales model are combined. It is basically assumed that both these predictions follow normal distributions and the final prediction is derived as a weighted average of the two predictions where the estimated standard deviations of the predictions determine the weights which are used. It may be that the distributions for the individual predictions may not satisfy the assumption of normality and hence the two predictions may not be combined in the most optimal way.

The second problem has to do with the rules that are used to determine the best prediction. These rules were derived by using trial and error to see what works and what does not but it is very difficult to keep track of what each rule does. It is also difficult to add new rules because any new rule may override some of the previous rules that are in use and then actually end up performing worse than before. As an example of this, assume that the true value of a property is R7 500 000. The property is much larger than the properties in the immediate surrounding area and therefore the estimate of the property from the comparable sales model is R2 000 000. The repeated sales model prediction is R8 000 000 and the combined prediction is R3 000 000. Now under the current rules in use, the prediction for this property is R3 000 000. This is of course not correct and a better prediction would have been the repeated sales model prediction of R8 000 000. If you go and add a rule such that for this particular case the prediction reverts to the repeated sales prediction of R8 000 000, it will more often than not cause lots of other predictions to also revert to the repeated sales model prediction, where in fact the combined prediction would have been a much better prediction and hence one ends up overall actually performing worse than before. Another problem is that the rules needs to be updated quite often since the property market changes and rules that have worked a few months ago may not be as effective when used now.

The first aim of this research would be to investigate if a neural network will not perhaps provide a better way to combine the estimates from the repeated sales model and the comparable sales model. The inputs to the neural network can be the various predictions together with their standard deviations, safety scores and confidence scores. These need not be the only inputs into the neural network and I will also experiment with other inputs to try and improve the combining of the predictions. Two that come to mind at this stage are the churn rate of the properties in a specific EA, and also the modelling segment into which the property falls. This will of course need to be experimented with and tested. The aim of this neural network would be to combine the predictions from

the two available models, by using the safety and accuracy scores and also various other inputs, to arrive at the best possible final prediction. This means that this model should in some way distinguish between when it is appropriate to weigh one prediction more and the other prediction less and vice versa. This is not impossible since we also have standard errors and indicators of the confidence we have in each of the estimates. This will be the first aim, to see if a neural network would not be able to do this combining in a more efficient way, such that the combined estimate from the neural network can always be used as the final prediction.

The second method that will also be experimented with is to use the current way of combining the estimates and then train a neural network that will be able to pick the best of the three available predictions, i.e. the repeated sales model prediction, the comparable sales model prediction or the combined prediction. The inputs to the neural network can be the different predictions, together with their safety and accuracy scores and various other inputs which we will look at in section 3.5. For this problem, we would then like the neural network to pick up the trends in the data and distinguish when a specific observation is considered a reliable observation. Some of the additional inputs to this network can be for example churn rate of properties in an area, since this may mean that the comparable sales model may provide a reliable estimate.

The reason why we are choosing to use these two methods is because the first method that we are going to try is an example of using a neural network in a regression context while the second method is for using a neural network in a classification context. These two problems will enable us to learn how to use a neural network for statistical problems and will therefore be very useful if we wish to use a neural network for a statistical problem in the future. This problem at hand is also quite complex in the sense that there might be multicollinearity present between the independent variables. It will also be seen from the data that there are quite a lot of inputs which can be used by the neural network but not

all of these inputs will provide meaningful information to the neural network. Therefore, there is quite a large exploratory data analysis phase to this study in order to determine the most appropriate inputs for the neural network.

3.5 Description of the data

I will now proceed to give a description of the data that will be used for building the neural network. This data set is obtained from mortgage application data that Lightstone collects from the banks which use their system. When Lightstone's AVM is used to do a valuation on a property, the user is required to enter a purchase price for the property. All this information that the desktop valuer at the bank enters is captured and a process is then used to see if this purchase price can be regarded as appropriate and therefore be used as the price at which the property is going to transact. These records are the ones which are going to be used for building the neural network.

Records from January 2009 to May 2009 were combined into one data set. From this data set, various checks and techniques were used to clean the data. This includes checking to make sure both the repeated sales and comparable sales are usable predictions, ensure that the purchase price which was entered by the valuer can be deemed as a valid purchase price at which the property will transact for and also removing data that can be considered as outliers. For the excluding of outliers, we looked at records which had extreme values (compared to the rest of the data) on one or more of the variables that are going to be considered for inputs to the neural network, and also records which had extreme values on the purchase price. These data cleaning techniques are used to make sure that the data set that is used for building the neural network contains only dependable records.

We will now continue and give a broad overview of the variables that will be considered as

inputs to the neural networks. It should be noted that this data set contains a very large number of variables, but most of the variables will not be useful in this specific problem. Therefore, I am going to give a description of the variables which are considered to be relevant in helping to solve this specific problem at hand. It should be noted, that this is done purely in a logical way, and not all the variables that are given here will be included into the final neural network. The variables that are finally included in the neural network, will be decided upon using exploratory data analysis and also by experimenting with a few neural networks. This involves including certain variables into a neural network and then excluding the variables to see if the variables really do contribute to increasing the performance of the neural network.

We will start by giving an explanation of the variables that are deemed to have an influence in the picking or combining of the optimal prediction. It is stressed again that these variables are only potential candidates for inputs to the neural network and will not necessarily all be included in the neural network that is going to be used. Each variable name (as it occurs in the data set) together with a description of the variable are going to be given here. We will start with available predictions together with their accuracy scores, safety scores and standard errors as these variables are deemed to be the most important inputs. Then we will name a few other variables which may also have an influence and add extra information to the model. The variables which will be considered as inputs to the neural network are:

Variable name	Description
predval_rs	Prediction from the repeated sales model.
p_ab_rs	Accuracy score for the repeated sales prediction.
p_90_rs	Safety score for the repeated sales prediction.
sigma_rs	Standard error of the repeated sales prediction.
predval_cs	Prediction from the comparable sales model.
p_ab_cs	Accuracy score for the comparable sales prediction.
p_90_cs	Safety score for the comparable sales prediction.
sigma_cs	Standard error of the comparable sales prediction.
predval_comb	Combined prediction.
p_ab_comb	Accuracy score for the combined prediction.
p_90_comb	Safety score for the combined prediction.
sigma_comb	Standard error of the combined prediction.
mod_seg	16 Modelling segments into which properties in South Afica are classified into by Lightstone. These modelling segments also accounts for whether a property is freehold or sectional title.
ssflag	Denotes whether it is a freehold property or a sectional title property. Should not be used together with *mod_seg* as this variable is contained in *mod_seg*.

flag_used Gives an indication of whether the last three years' or the last five years' comparable sales are used for the comparable sales model. The reason for the inclusion of this variable is that if there were many recent comparable sales, it might give an indication that the prediction from the comparable sales model can be deemed reliable.

comp_num_used The number of comparable sales used in the comparable sales model prediction. Again, if there are many recent comparable sales used for the calculation of the comparable sales prediction, this prediction may be reliable.

csflag Gives an indication of the which comparable sales are used, i.e. if the sales occured in the same EA or sectional scheme, or if the sales of this EA/sectional scheme was pooled with the sales of the adjacent EA/sectional scheme because there was not enough data in the EA/sectional scheme alone. As soos as the comparable sales are pooled with the data from adjacent area, the comparable sales prediction might not be as reliable. This is because all the properties within a certain EA/sectional scheme may be very homogenous, but may also differ a huge amount from the properties in the adjacent EA/sectional scheme.

churn Variable to indicate the number of properties sold in an area over a period of a year. This gives an indication of whether there is many properties that are transacting in an area. This might help with the comparable sales prediction of an area: high number of transactions in an area, more data for comparable sales and hence comparable sales more reliable.

newmonthdiff Variables that shows the number of months between the date the property valuation is done and the date at which the property last transacted on. Recall that for the repeated sales model, the coefficients from the model is used to inflate a previous transaction price of the property up to a value at a specified date. The reason behind this variable maybe being a potential candidate is that if the time elapsed between the date from which we inflate from to the date up to which we inflate to is a long period, the prediction from the repeated sales model is subject to more variability. This variable may help provide information on whether the particular repeated sales model prediction is a reliable prediction.

Other variables which are also used are:

Purcase_Price This is the price at which the property transacted for. The variable will be used as the dependent variable in the neural network for regression.

predval_final The final prediction which Lightstone currently uses to provide a value for a property.

pred_method Indicates which of the predictions were used for the current Lightstone final prediction (predval_final).

The data set that will be used for this analysis contains 44 498 observations.

3.6 Analysis

In this section we will start by conducting an exploratory data analysis phase. This is to enable us to determine which of the variables that were specified in the previous section (cf. section 3.5) are the most relevant to use as input into the neural network that will be used for this problem. It is not always easy to know which of the variables should be included and which should be excluded by just looking at descriptive statistics of the variables and therefore we will also train a few neural networks, including and excluding certain variables to see if they do increase the performance of the neural network.

3.6.1 Exploratory analysis

We will give a short discussion about each variable that are considered as input to the neural network and then also do an exploratory analysis to see if the variable should be included into the neural network. Before we continue with the analysis we first need to define the following three variables:

$$error\_percentage\_rs = 100\left(\frac{predval\_rs}{purchase\_price} - 1\right)$$

$$error\_percentage\_cs = 100\left(\frac{predval\_cs}{purchase\_price} - 1\right)$$

$$error\_percentage\_comb = 100\left(\frac{predval\_comb}{purchase\_price} - 1\right)$$

These variables shows the percentage error made of a prediction in relation to the actual purchase price. We will make extensive use of these variables in the exploratory analysis as it is easy to see if certain variables have an influence by lowering the error percentage of a prediction.

Accuracy and safety scores

We will start with some analysis of the variables that are thought to be most relevant to this analysis, for both the combining of the predictions and also the picking of the correct predictions. The variables that are thought to be the most relevant inputs in this analysis are the accuracy and safety scores of each of the predictions. Recall the accuracy score is calculated by using a logistic model and the value signifies the probability of the prediction being within 20% of the actual purchase price. To show the importance of the accuracy score as an input we will look at the percentage of predictions that lie within in 10% of the actual purchase price, for various bands of the accuracy score. This will be done for the repeated sales (RS), comparable sales (CS) and the combined prediction (COMB). The results are displayed in table 3.1. From this table we see that generally, a high accuracy score indicates that the prediction tends to be more accurate. Therefore, the accuracy score should be included as inputs to the neural network as these variables give information about the accuracy of the predictions. The same calculation have been done to have a look at the percentage of predictions that lie within 20% of the purchase price and this result can be found in table 3.2. This results again confirms that accuracy score should definitely be included as an input to the neural network.

		% RS predictions within 10% of purchase price	% CS predictions within 10% of purchase price	% COMB predictions within 10% of purchase price
Accuracy score band	[0;50)	17,31	21,58	22,51
	[50;60)	30,07	24,82	30,30
	[60;70)	36,25	36,28	35,62
	[70;80)	44,25	45,13	44,83
	[80;90)	55,75	55,96	58,04
	[90;100]	65,81	67,01	73,83

Table 3.1: Table displaying the percentage of predictions that are within 10% of the purchase price for various categories of accuracy scores. This is done for each of the available predictions.

The next input which is also considered to be important is the safety scores of each of

		% RS predictions within 20% of purchase price	% CS predictions within 20% of purchase price	% COMB predictions within 20% of purchase price
Accuracy score band	$[0; 50)$	34,33	41,28	43,00
	$[50; 60)$	56,04	55,15	55,93
	$[60; 70)$	65,17	64,27	63,80
	$[70; 80)$	74,47	74,67	74,27
	$[80; 90)$	84,61	85,61	86,32
	$[90; 100]$	93,45	92,78	94,39

Table 3.2: Table displaying the percentage of predictions that are within 20% of the purchase price for various categories of accuracy scores. This is done for each of the available predictions.

the predictions. The safety score is also the output from a logistic regression model and signifies the probability that the prediction will not exceed the purchase price by more than 10%. This means that the safety score should give the neural network an indication of whether the prediction is considered to be excessively low or excessively high. This is also valuable information which can be used in the neural network. To show the effect of the safety score, we will also divide the safety scores into bands and categorise each of the predictions for the various safety score bands. The results are given in table 3.3 for the RS predictions, in table 3.4 for the CS predictions and in table 3.5 for the COMB predictions.

		> 30% under	20-30% under	10-20% under	0-10% under
Safety score band	[0; 50)	3,27	3,47	6,77	9,66
	[50; 60)	5,51	6,49	10,50	14,25
	[60; 70)	8,10	7,71	12,12	17,21
	[70; 80)	10,99	10,29	15,75	19,62
	[80; 90)	16,87	12,15	18,23	20,71
	[90; 100)	32,37	13,83	17,83	18,55

		0-10% over	10-20% over	20-30% over	> 30% over
Safety score band	[0; 50)	13,54	16,92	14,61	31,75
	[50; 60)	18,62	17,28	11,31	16,04
	[60; 70)	19,10	16,11	9,27	10,39
	[70; 80)	18,47	12,53	6,25	6,08
	[80; 90)	16,78	8,58	3,64	3,05
	[90; 100)	12,04	3,72	0,98	0,67

Table 3.3: Percentage of repeated sales model predictions categorised into different classifications according to various safety score bands (these percentages are interpreted as row percentages)

		> 30% under	20-30% under	10-20% under	0-10% under
Safety score band	[0; 50)	5,94	4,38	7,07	10,94
	[50; 60)	7,94	6,23	9,28	13,95
	[60; 70)	9,02	8,49	12,02	17,56
	[70; 80)	10,60	10,09	14,79	20,27
	[80; 90)	14,63	11,66	18,10	22,58
	[90; 100)	22,36	13,84	22,51	23,40

		0-10% over	10-20% over	20-30% over	> 30% over
Safety score band	[0; 50)	12,85	15,00	12,64	31,19
	[50; 60)	16,38	15,61	10,72	19,91
	[60; 70)	18,25	13,30	8,99	12,37
	[70; 80)	18,10	11,93	6,92	7,30
	[80; 90)	17,63	8,82	3,52	3,06
	[90; 100)	13,07	3,78	0,66	0,39

Table 3.4: Percentage of comparable sales model predictions categorised into different classifications according to various safety score bands (these percentages are interpreted as row percentages)

		> 30% under	20-30% under	10-20% under	0-10% under
Safety score band	[0; 50)	5,56	4,84	9,47	11,43
	[50; 60)	8,82	6,46	11,85	15,71
	[60; 70)	9,80	9,06	12,68	17,38
	[70; 80)	11,55	9,77	15,03	19,88
	[80; 90)	14,78	11,58	17,75	21,98
	[90; 100)	14,88	13,61	21,42	27,11

		0-10% over	10-20% over	20-30% over	> 30% over
Safety score band	[0; 50)	15,45	17,10	12,67	23,48
	[50; 60)	17,68	15,20	9,13	15,16
	[60; 70)	17,32	13,29	8,31	12,16
	[70; 80)	17,65	12,18	6,59	7,34
	[80; 90)	17,48	9,18	3,56	3,68
	[90; 100)	18,48	5,32	0,89	0,28

Table 3.5: Percentage of combined model predictions categorised into different classifications according to various safety score bands (these percentages are interpreted as row percentages)

Standard errors of the predictions

The next variables to consider are the standard errors of the predictions. Intuitively one would feel that the standard error of the predictions will give a good indication of the variability in the estimate and hence will be a measure of how reliable the estimate is, with a smaller standard error indicating a more stable and hence better estimate. The histograms of the standard errors of the three predictions can be found in figure 3.21. We see that the distribution of these variables are very skew and therefore we would need to transform these variables in order to use them in the neural network. A logarithmic transformation will be done on the variables and the result can be found in figure 3.22. We see from the histograms that the transformed variables are more symmetrical and will be better suited for use in the neural network.

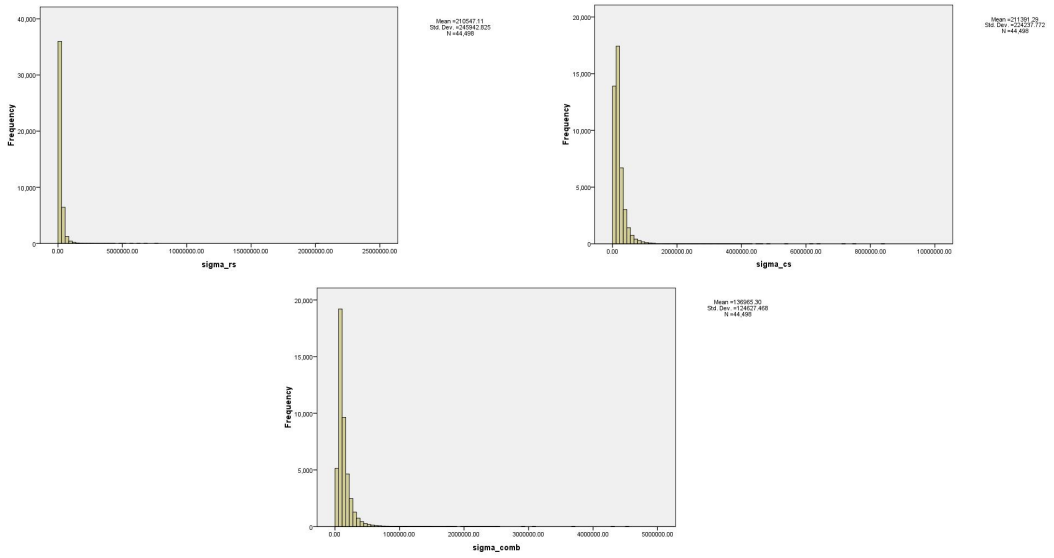


Figure 3.21: Histogram of *sigma_rs* (top left), variable *sigma_cs* (top right) and *sigma_comb* (bottom).

The problem with the standard errors are that they don't seem to add any useable information. If we do a plot of the transformed standard error against the percentage error made for each of the predictions we get the plots in figure 3.23. We see from this figure, that there does not seem to be any pattern which will add useable information to

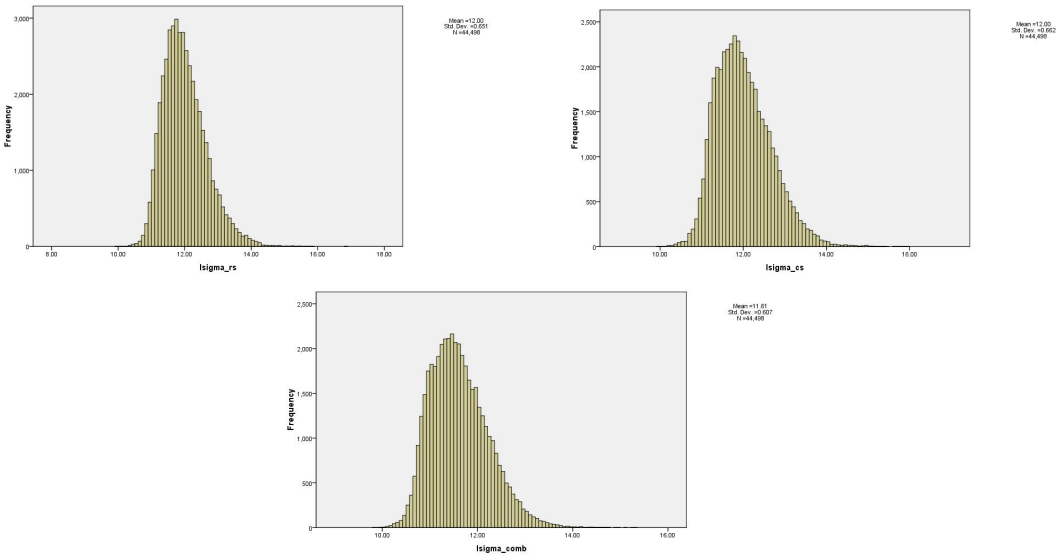


Figure 3.22: Histogram of *lsigma_rs* (top left), *lsigma_cs* (top right) and *lsigma_comb* (bottom).

the model for example a lower standard error which leads to a better estimate. From the scatterplots we see many cases where a prediction with a low standard error has a large error, both positive and negative, and vice versa. This variable will however not be just discarded and we will test this variable in the neural network to see if it improves performance or not.

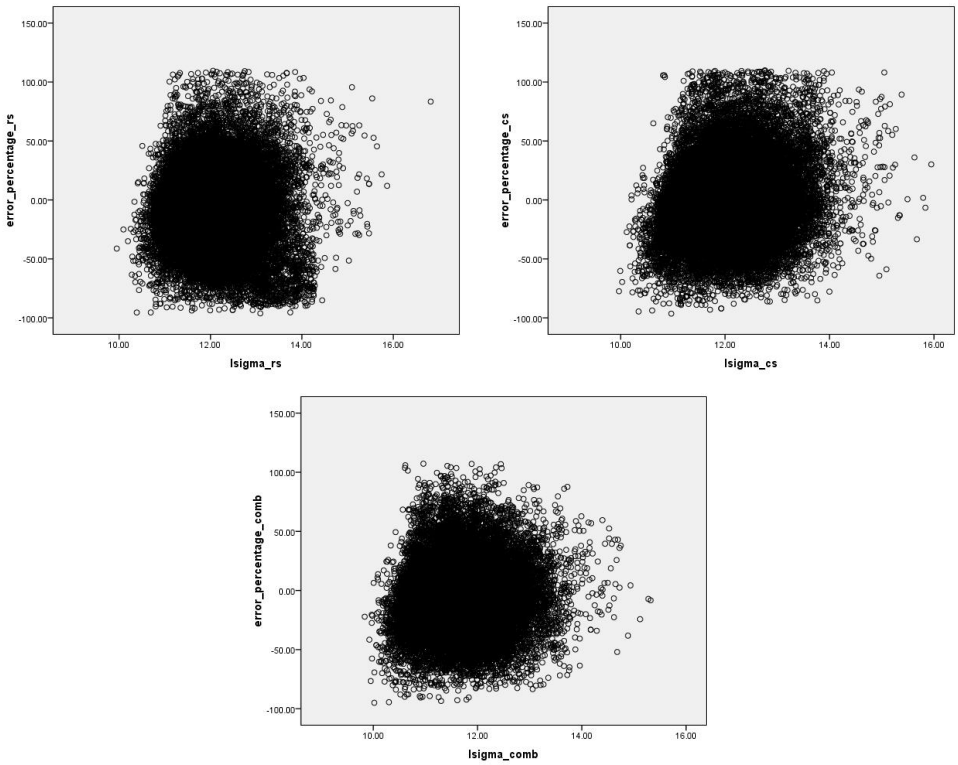


Figure 3.23: Scatterplot of the transformed standard error of each prediction against the percentage error made for that prediction

Modelling segments

Lightstone categorises properties into 16 different segments. These segments differ with respect to the value of the property and then also the location of the property and whether the property is freehold or sectional title. For the repeated sales model, a model is built for each of these segments. This means that the repeated sales model already account for the differences in segments. This is also true for the comparable sales model, since a property will only be a candidate for a comparable sale if the properties fall into the same segment. Therefore, we may not get a significant improvement by including the variable *mod_seg* into our neural network, but it may still be worthwhile to include this as an input to the neural network and see if we get a better result.

If we construct boxplots of the error percentages of the three predictions according to the various modelling segments, we obtain the results presented in figure 3.24. The + sign in each of the boxes indicate the mean, and the width of the boxes indicate the number of observations within that modelling segments relative to the other modelling segments. We see that there is variability of the errors for each prediction between the modelling segments. For example, for segment A, which corresponds to townships, we see that median error made by the repeated sales model is smaller than zero while the median error made by the comparable sales model is very close to zero. This may give extra information to the neural network and may signify that for a large number of cases in segment A, the comparable sales model performs better than the repeated sales model. Therefore, *mod_seg* should be tested as an input to the neural network.

The variable *ssflag* gives an indication whether a property is a freehold or a sectional title property and although this variable is contained within the *mod_seg* variable it might still be worthwhile to test on its own as well.

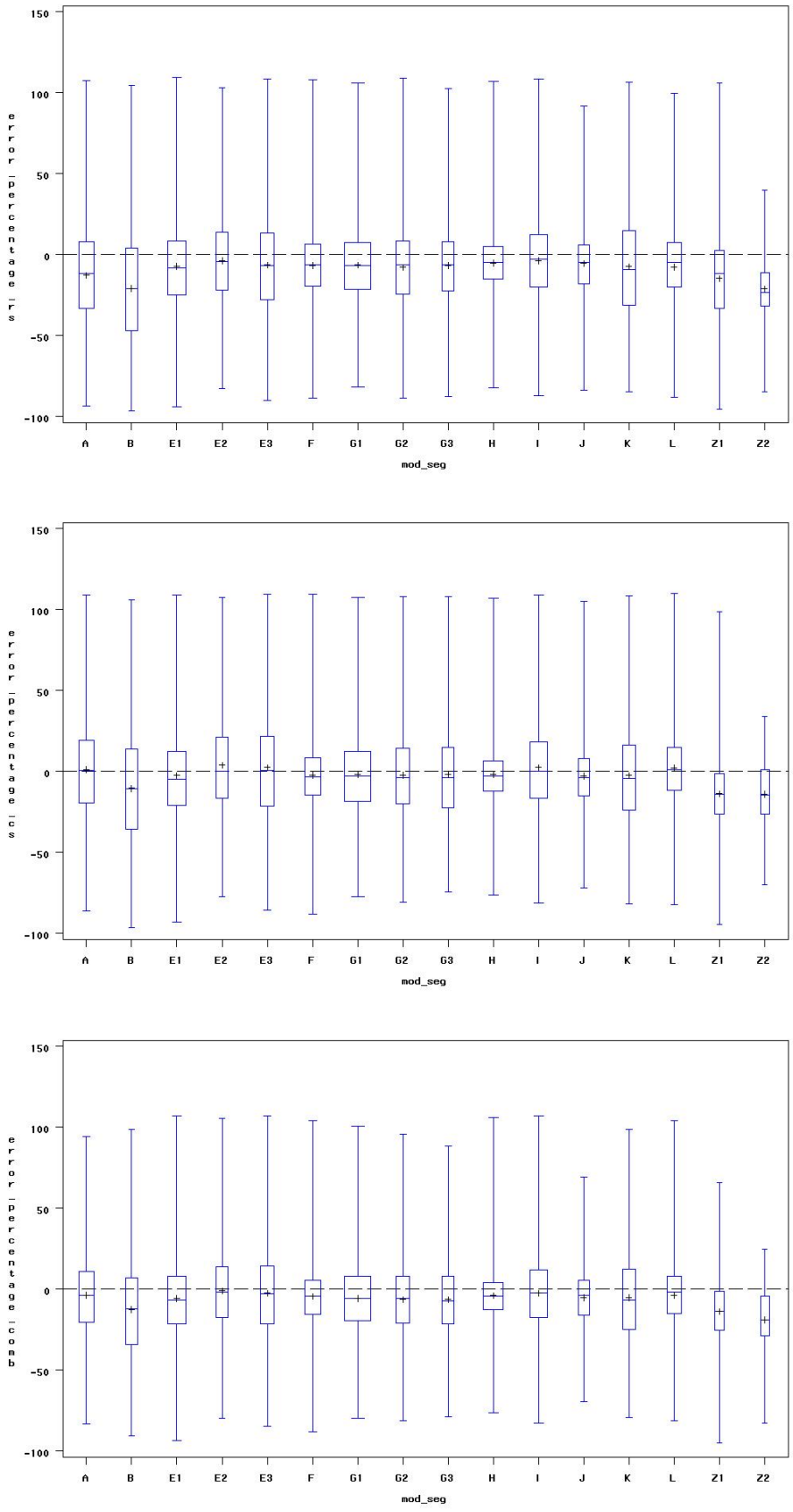


Figure 3.24: Boxplots of the error percentages of each prediction, according to *mod_seg*.

The variables *csflag*, *flag_used*, *comp_num_used* and *churn*

The two variables, *csflag* and *flag_used*, give information on how the properties were chosen to arrive at a comparable sales model prediction. In a nutshell, *csflag* gives info on whether we used this own EA's/Sectional Scheme's sales or whether we had to combine the sales with an adjacent EA or Sectional Scheme (remember the comparable sales are done differently for freehold properties, sectional schemes and estates). The variable *flag_used* gives information on whether the last three years' comparable sales were used or whether we had to resort to using the last five years' sales because there was not sufficient data within the last three years. The variable *comp_num_used* is the number of properties that was used to arrive at a comparable sales prediction. *Churn* is also closely related to these variables because if an area has a higher churn rate, it means that the houses are bought and sold relatively quickly. This implies that there should be many recent comparable sales in the area and this may help the comparable sales model.

The logic behind why we must look at including these variables is that if we have an area where all the properties are relatively homogenous and there were many recent sales of these properties, the comparable sales model should give a very good prediction. To give an example of this, if we have a sectional scheme where all the units within this scheme are relatively similar, and we have a large number of sales which occurred recently (within the last year or two), then we should get a very good idea of the market value of out subject property by looking at these comparable sales. On the other hand, if there is not enough comparable sales within this sectional scheme and we have to resort to the adjacent scheme, but the adjacent scheme are all duplexes where the scheme under consideration are simplexes, we would intuitively feel that the comparable sales prediction might not be as accurate in this case.

Therefore these variables will also be tested as inputs to the neural network to see if this

gives any improvement in the performance of the neural network. It should be noted that *csflag* and *flag_used* are categorical variables with 19 and 3 possible categories respectively and that *comp_num_used* and *churn* are discrete and continuous variables respectively.

The variable *newmonthdiff*

Recall that the repeated sales model works by inflating a previous sales price of the property from sales data, up to any date in the future using the coefficients from the repeated sales modelling. Now the way in which the repeated sales prediction is calculated for a specific property, is by using the previous sale price of the property and inflating the price from that previous sales date, up to a current estimate of the value of the property. The variable *newmonthdiff* denotes the timespan, measured in months, between the previous sale date and the date up to which we inflate to. This current date will correspond to the date at which the query for the value of the property is received, since we want an estimate of the value of the property at that specific time.

A reason why we might look to include this variable into the neural network is because, the longer the time span between the date from which we inflate from, to the date which we inflate to, the more variability there usually is in the repeated sales prediction. Of course, we should remember that this is not always the case but it is still a worthwhile variable to test in the neural network.

3.6.2 Method followed for building the neural network model

The challenge that we are facing with this problem is that we are not certain which inputs should be used to the neural network. It is also difficult to do an in-depth exploratory

study to determine the variables because we are working with a very large data set and this makes it difficult to look at basic descriptive statistics like correlations and basic graphs like scatterplots. If a sample size is sufficiently large like the one considered here, extremely small differences and correlations can be found to be statistically significant but these variables may not be practically significant to include in the neural network. For example, if we test whether the mean error percentage for the repeated sales model differs from segment to segment (*mod_seg*), the result would most likely be highly statistically significant, even though the mean percentages between the segments differ very little. This is directly related to the large sample size. Therefore, we can see that it is not sufficient to just look at the exploratory statistics when deciding which variables to include in the neural network's inputs and therefore we would need an approach to get around this problem.

The approach that we are going to follow to try and get around this problem is as follows: We will start by including the basic inputs which should logically be included in each of the two neural networks. This will basically consist of only the accuracy and safety scores of the predictions since they are directly related to the confidence we have in the accuracy of each of the predictions. Then we will start adding some variables to see if the variables significantly increase performance or not. In the end, we will have trained a number of neural networks for both the combining and the picking of the prediction and therefore we can then make comparisons between the networks by looking at various benchmarks that will be defined in section 3.7. This should give us a clear view of the variables that are of practical significance in the neural network and which are to be used in the final neural network.

In section 2.3.5 it was mentioned that the initial values influences to which minimum the neural network will converge to if backpropagation is used. Therefore, we will train five neural networks for each set of inputs, and only report the performance of the best neural

network obtained from these five independent runs. One would benefit from doing more than five runs, but because we are working with such a large data set and training is quite slow on such large data sets, we will only use five runs.

The neural networks to be trained will have a relatively large number of hidden neurons in the single hidden layer. The approach that will be followed will be similar to the approach described in section 3.3 where we divide the neural network into a modelling and testing set and then doing the modelling and testing concurrently for each epoch to arrive at the best neural network as tested on the testing set. The weights from this neural network will then be used for modelling on the whole data set, and the performance measures that are obtained from this neural network will be stated in the results.

The stopping criterion that is to be used for these neural networks differ from the normal neural network methods that are usually used in neural network applications. In our simulations (cf. section 3.3) we used the neural network which gave the lowest sum of squares error as measured on the testing data set for the regression case. For the classification neural network, we used the neural network that gave the lowest missclassification rate as measured on the testing data set. These are the most common measures which are used. For the neural networks that we are using, our main aim is to optimise the percentage of predictions that lie within 20% of the actual purchase price of the property. Therefore, this will be used as our stopping criterion. We will select the weights from the neural network, both in the classification and regression case, that give the highest percentage of predictions that are within 20% of the purchase price as measured on the independent testing set. For the classification case it is still possible to use this stopping criteria since the classification from the neural network signifies which of the RS, CS or COMB prediction should be used, and we can then still determine if this chosen prediction lies within 20% of the purchase price. This will not be possible when we are using a neural network in a typical statistical classification application.

All the continuous inputs to the neural network will be standardised and the categorical inputs will be coded using the 1-of-K coding. A small learning rate of $\alpha = 0,005$ will be used for training and we will also include a momentum term of $\eta = 0,4$ (cf. section 2.3.5). We will use a neural network with 40 hidden nodes in one single hidden layer and the network will be trained for 1 000 epochs. In the next two sections (cf. sections 3.6.3 & 3.6.4), the neural networks that are to be trained specifically for the classification and regression case will be discussed in more detail.

All the neural networks that are to be trained will be coded using SAS IML. The code is similar to the code that was used to train the backpropagation neural neural networks for the simulations in section 3.3 but SAS IML was chosen in this instance for its speed and ability to handle large amounts of data. Lightstone also uses SAS for all their analyses and therefore the neural network will have to be trained using SAS if the network is to be implemented in Lightstone's environment.

3.6.3 Neural network for picking correct prediction

The aim of the neural network in this instance is to classify each observation into a class, where this class will correspond to the best prediction for that observation. This means that the dependent variable of the neural network in this case will be categorical and will be able to take on the values *RS*, *CS* or *COMB* where each of these values signify which of the predictions for that observation is considered to be optimal. Hence we basically want the neural network to choose the best one of the possible predictions.

The first thing we need to do is to create a new variable which will contain the optimal prediction for each observation. The way in which this is going to do be done is rather simple but does seem to be effective. We will take the prediction that is the closest to

the purchase price in absolute value and that prediction will be considered the optimal prediction. To illustrate this, we will look at an example.

Purchase_Price	$\hat{y}_{RS}$	AS_{RS}	SS_{RS}	$\hat{y}_{CS}$	AS_{CS}	SS_{CS}
250 000	200 000	0,64	0,89	290 000	0,62	0,40
$\hat{y}_{COMB}$	AS_{COMB}	SS_{COMB}				
220 000	0,71	0,87				

$\mathbf{\hat{y}_{RS}}$, $\mathbf{\hat{y}_{CS}}$ and $\mathbf{\hat{y}_{COMB}}$ denotes each of the three predictions. *AS* refers to the accuracy score, while *SS* refers to safety score with the subscripts in each case stating to which prediction the scores relates to. From this observation, we see that RS model under predicts by 50 000, the CS model over predicts by 40 000 and the COMB prediction is under by 30 000. We will make no distinction between underpredicting and overpredicting and just try to get a prediction that is as close to the 250 000 as possible and in this case it will be the COMB prediction.

A new variable called *optimal* will be created using this approach. Although we use a fairly simple way to define the dependent variable, the expectation is that the neural network will fit the underlying structure in the data and get to see the hidden patterns between the accuracy scores, safety scores and all the other inputs which will be considered. This should then lead to a model, which can be used to classify new observations in the future. If however this neural network does not perform as well as we anticipate, the definition of the dependent variable will be a good place to start to search for improvement. The reason for this, will also be illustrated with an example. Suppose we have the following observation:

Purchase_Price	$\hat{y}_{RS}$	AS_{RS}	SS_{RS}	$\hat{y}_{CS}$	AS_{CS}	SS_{CS}
270 000	254 153	0,64	0,81	245 211	0,62	0,85
$\hat{y}_{COMB}$	AS_{COMB}	SS_{COMB}				
249 778	0,71	0,87				

We see from this record, that the RS prediction is the closest to the purchase price of 270 000, but the CS & COMB predictions are not that far off. If we consider that we are working with values of a property, it would seem that there is actually not a significant difference between these predictions and we can consider the predictions as equal. Here, one can use many creative ways to define the dependent variable and also try to influence the neural network model. For example we can divide the predictions into different error bands and say that if two predictions fall into the same error band, the optimal prediction should be the one with the highest accuracy score. This is perhaps something to pursue in the future to improve the performance of the neural network. More topics to be researched in the future will be discussed in section 3.12.

If we do a frequency count on the newly created variable *optimal* we find:

Value of *optimal*	Frequency	Percentage
RS	17 756	39,90
CS	19 430	43,66
COMB	7 312	16,43

Table 3.6: Frequency table on dependent variable used in neural network

For the purpose of picking the correct classification I will train a few neural networks, each with a different number of inputs and then compare the performance. The neural networks that will be trained are given in table 3.6.

As was stated in the previous section, we will train five neural networks for each set of inputs, each time choosing different initial values. The results from the best one of these five neural networks will be quoted.

Neural network	Inputs chosen
1	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}
2	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$
3	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, mod_seg
4	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, mod_seg, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, $\log(\sigma^2_{COMB})$
5	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, mod_seg, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, $\log(\sigma^2_{COMB})$, churn, comp_num_used, csflag, flag_used, newmonthdiff_query
6	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, mod_seg, churn, comp_num_used, csflag, flag_used, newmonthdiff_query

Table 3.7: Table showing the different neural networks that will be trained and the inputs to each of the different neural networks for the selection of the optimal prediction.

3.6.4 Neural network for combining predictions

In the previous section we discussed how a neural network will be used to select one of the existing predictions that are already in the data set, namely the RS prediction, the CS prediction and the COMB prediction. In this part we will discuss a neural network with the goal to combine the repeated sales prediction and the comparable sales prediction to arrive at a new combined prediction, where this combined prediction should be an optimal prediction. In its basic form, our aim is to give the neural network the RS and CS predictions and their respective accuracy and safety scores, and the neural network should assign weights to each of these inputs to arrive at a combined prediction which will always be optimal.

This neural network is an example of a neural network which can be used for regression purposes. The dependent variable for this neural network is the variable *Purchase_Price* which is a continuous variable. As with the previous neural network, we will again choose a few sets of inputs, train five different neural networks for each set of inputs, and quote the performance of the best run to see which of the neural networks perform the best.

Table 3.6.3 gives an indication of the neural networks that will be trained for combining the RS and CS predictions. From this table, we see that we will again start with a basic set of inputs, and then add some extra inputs to see if it makes any difference to the performance of the neural network.

Neural network	**Inputs chosen**
1	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$
2	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$
3	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, mod_seg
4	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, mod_seg, churn, comp_num_used, csflag, flag_used, new-monthdiff_query

Table 3.8: Table showing the different neural networks that will be trained and the inputs to each of the different neural networks for the combining the RS and CS predictions.

3.7 Results

In this section we will have a look at the results that were obtained from the neural networks that were specified in sections 3.6.3 & 3.6.4. The best result from both the neural network for picking the best observation, and the neural network that is used for combining will be compared with the results from the method that Lightstone currently

uses.

The performance of the current method and the neural networks will be measured by looking at the percentage of predictions that are within certain ranges from the actual purchase price. Specifically we will look at the following categories:

- Percentage of predictions within 5% of purchase price
- Percentage of predictions within 10% of purchase price
- Percentage of predictions within 20% of purchase price
- Percentage of predictions within 30% of purchase price

One of the most important performance measures for Lightstone is the percentage of predictions that are within 20% of the purchase price. This is the measure that is reported when giving an indication of the performance of the automated valuation model. Therefore, this will also be the single most important performance measure for us, and this will primarily be used to decide which neural network gives the best performance.

We will also analyse the results in more detail by looking at the nature of the predictions, i.e. by how much the predictions are over or below the actual sales price. These measures will be used to determine whether the neural network is suitable for implementation into the Lightstone environment. The reason why this is important is that the banks would generally prefer to undervalue a property than overvalue it. If a property is severely overvalued and a bank grants a loan based on this prediction it poses a very high risk to the bank. Therefore we should also investigate the chosen neural networks to assess the nature of the predictions arising from these neural networks with respect to whether the models tend to overpredict or underpredict too severely.

3.7.1 Benchmark results

We will start by giving the results of the method that Lightstone currently uses. This will give us a benchmark against which we can compare the neural networks. The performance of the method that is currently in use is given in table 3.9. From this table, we see that our ultimate goal would be to train a neural network that will be able to perform substantially better than the 66,04% that is given in the table for the percentage of observations that are within 20% of the purchase price.

	Within 5%	**Within 10%**	**Within 20%**	**Within 30%**
Percentage of observations	20,26	39,03	66,04	81,34

Table 3.9: Performance of current Lightstone method

If we look at the percentage of predictions that are below or above the given purchase price we obtain:

	Frequency	**Percent**
OVER	18 488	41,55
UNDER	26 010	58,45

Table 3.10: Table displaying the number of predictions that are above and below the purchase price

From table 3.10 we can clearly see the tendency of Lightstone to rather make sure that the prediction given to the banks is never too high. This is also where the safety score is used extensively, since it will attach a confidence level to a prediction not being too high. If we categorise the predictions according to the percentage overpredicting or underpredicting we obtain the results in table 3.11.

If we look at the percentages obtained in table 3.11 we see that more than 50% of the predictions are within 0-20% under the purchase price or 0-10% over the purchase price. This is also what we would want from the neural networks. Also note the relatively low

	Frequency	Percent
Over by > 30%	3 115	7,00
Over by 21-30%	2 446	5,50
Over by 11-20%	4 886	10,98
Over by 0-10%	8 041	18,07
Under by 1-10%	9 328	20,96
Under by 11-20%	7 128	16,02
Under by 21-30%	4 364	9,81
Under by > 30%	5 190	11,66

Table 3.11: Prediction categories for the current Lightstone predictions

number of predictions that are over by more than 20%. This is also a characteristic which the neural network should exhibit. If we do a frequency check on which of the available predictions Lightstone uses we obtain the following:

Prediction used	Frequency	Percentage
RS	2 881	6,47
CS	3 559	8,00
COMB	38 058	85,53

From this frequency table we see that Lightstone uses the COMB prediction for the majority of the cases as it is their policy to use the COMB prediction unless there is good evidence that this prediction is not reliable and one of the other two predictions should be used. This evidence can be in the form of a very low safety or accuracy score, which will indicate that the confidence in the prediction is low. The classification rule used in the neural network for picking the optimal prediction will probably also be modified to mimic this pattern in that the COMB prediction should be used unless there is enough reason to use one of the other predictions.

Now that we have a good idea of how the current Lightstone prediction method performs we can analyse the data from the two best neural networks in more detail to see if the neural network approach can match or outperform the Lightstone method.

3.7.2 Results and discussion for neural network used for picking best observation

We will start by giving the results for the six neural networks that were specified in section 3.6.3 which were used to select the best prediction out of the three available prdictions in the data. For each of the five runs that were done, for each set of inputs, we will report the results of the run that had the highest percentage of predictions within 20% of the actual purchase price, when compared against each other. These results for each of the five different neural networks are given in table 3.12. The complete detailed results for each of the runs that were conducted can be found in appendix A.5. In section 3.8.1 we will analyse the predictions from the best model as chosen here.

An interesting observation is that the neural networks did not seem to be influenced by the effect of multicollinearity in the data. In section 3.4, we showed that the COMB prediction is a combination of the RS prediction and the CS prediction, where the weights attached to each of the predictions are determined by the standard errors of each of the predictions respectively. It was shown that the standard error of the COMB predicion is also derived from the standard errors of the RS and CS predictions. In neural network number four and five, we have included the RS, CS and COMB predictions and the standard errors of each of the predictions. Therefore there is definitely multicollinearity present in the data, and the performance from those neural networks does not seem to be negatively influenced. This finding seems to be in-line with findings in the literature which also state that neural network does not seem to be influenced by multicollinearity (Carpio and Hermosilla (2001); De Veaux and Ungar (1994b)).

NN #	Input variables	Within 5%	Within 10%	Within 20%	Within 30%
1	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}	20,38	38,70	65,06	80,95
2	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$	20,88	39,57	66,07	81,50
3	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, mod_seg	21,32	40,14	66,65	81,71
4	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, mod_seg, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, $\log(\sigma^2_{COMB})$	21,30	40,06	66,31	81,40
5	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, $\log(\sigma^2_{COMB})$, mod_seg, churn, comp_num_used, csflag, flag_used, newmonthdiff_query	22,41	41,22	67,47	82,27
6	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS} AS_{COMB}, SS_{COMB}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\hat{y}_{COMB}$, mod_seg, churn, comp_num_used, csflag, flag_used, newmonthdiff_query	22,05	40,97	67,21	82,02

Table 3.12: Performance comparison of neural network used for picking correct prediction (classification)

3.7.3 Results from neural network used for combining

Now we will present the results for the neural network that was used in a regression context, i.e. to use each of the RS & CS predictions and other associated characteristics and combine these predictions to form a new optimal combined prediction. The results for the best run out of the five runs conducted, for each of the four different neural networks, can be found in table 3.13. The complete detailed results for each of the runs that were conducted can be found in appendix A.5.

From table 3.13, we see that the simplest of the neural networks that were considered, namely neural network number one, seem to perform the best. The inputs to this neural network is the RS & CS prediction and each of the associated accuracy and safety scores. It is quite interesting to note that the more inputs we included, the more the performance of this neural network degraded. The predictions from neural network one will be saved and these predictions will be further analysed in section 3.8.1.

NN #	Input variables	Within 5%	Within 10%	Within 20%	Within 30%
1	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$	20,77	39,63	67,08	82,36
2	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$	20,97	39,78	66,96	82,34
3	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, mod_seg	20,77	39,89	66,84	82,39
4	AS_{RS}, SS_{RS}, AS_{CS}, SS_{CS}, $\hat{y}_{RS}$, $\hat{y}_{CS}$, $\log(\sigma^2_{RS})$, $\log(\sigma^2_{CS})$, mod_seg, churn, comp_num_used, csflag, flag_used, newmonthdiff_query	20,62	39,53	66,75	81,94

Table 3.13: Performance comparison of neural network used for combining (regression)

3.8 Performance of fitted neural networks

In this section we will have a look at each of the best neural networks from the two different approaches. We will start by doing a post hoc analysis for each of the chosen neural networks for the selection of the best prediction and also the combining of the predictions. We will also discuss some of the problems that were found while we were building these two different neural networks and then we will discuss how neural networks like these can be implemented in Lightstone's environment for use.

For convenience, we will display the performance characteristics of the current Lightstone approach, together with performance measures from the best neural network for combining and the neural network for selection the best prediction in table 3.14.

Approach	**Within 5%**	**Within 10%**	**Within 20%**	**Within 30%**
Current Lightstone	20,26	39,03	66,04	81,34
Combining NN (Nr.1)	20,77	39,63	67,08	82,36
Selection NN (Nr.5)	22,41	41,22	67,47	82,27

Table 3.14: Performance comparison neural network approaches against the current Lightstone approach.

Our next task would be to give a more in-depth analysis of the results from the two different chosen neural networks respectively. We will also look at whether the neural networks are more prone to picking predictions which are too high or too low.

3.8.1 Post-hoc analysis

We will continue to do a post hoc analysis of the neural networks under consideration. The reason why this is important is that before we can implement a black box prediction model into a business environment like Lightstone, we should first try to see exactly what

the different consequences from the neural network will be when presented with different scenarios. This is a little easier to do for the classification neural network in this case than it is for the regression neural network. For the classification neural network, we can have a look at a few of the observations that were misclassified and try to get an understanding why these observations are not correctly classified, but for the regression neural network this is not possible since the output is a continuous value which is a highly non-linear combination of all the inputs. For this reason, we will start with a small post hoc analysis of the regression neural network before we continue with a more in-depth study for the classification neural network.

Post hoc analysis of neural network used for combining the predictions

From the results that were presented in table 3.13, we see that neural network number one seems to perform the best. This neural network takes the RS & CS prediction together with each of their associated accuracy and safety scores respectively, and combines all these inputs to form a continuous output which can be used as the predicted value of the property. We see that the 67,08 percent of neural network predictions that are within 20% of the purchase price compares quite favourably to the 66,04% that Lightstone currently achieves (table 3.9). We see that the neural network outperforms the current approach based on the other performance measures as well.

If we categorise the predictions according to whether we are underpredicting or overpredicting we obtain the results in table 3.15 and the more detailed categories can be found in table 3.16. We see from table 3.15 that the predictions from the neural network are generally higher compared to the Lightstone predictions in table 3.10. In table 3.16 we can clearly see how the predictions shifted upwards with quite an increase in overpredictions compared against the Lightstone method for which the results are given in table

3.11. This is a disadvantage of this method, but we will need to investigate further before any conclusions can be made with regards to this neural network.

	Frequency	**Percent**
OVER	21 796	48,98
UNDER	22 702	51,02

Table 3.15: Table displaying the number of predictions from the neural network that are above and below the purchase price

If we plot the predicted values from this neural network against the purchase price of the property we obtain the plot in figure 3.25. The 45 degree line is also displayed in this plot to make comparisons easier. The first problem we encounter is that some of the predicted values seem to be negative which is impossible for a value of a property. This is not a major concern as there can be many ways to work around this problem. One of these is to revert back to the old combined prediction when a prediction from the neural network is negative. Generally we see that the neural network seems to underpredict as the bulk of the predictions lie below the 45 degree line. This is not a bad thing since the banks would generally prefer to underpredict on a property than overpredict to avoid granting too high loan to a client. If we focus on the area for predictions with a purchase price of between 0 and 5 000 000 we obtain the plot presented in figure 3.26.

In figure 3.26, we can more clearly see the underpredicting nature of the neural network.

	Frequency	**Percent**
Over by $> 30\%$	4 253	9,56
Over by 21-30%	3 170	7,12
Over by 11-20%	5 747	12,92
Over by 0-10%	8 262	19,39
Under by 1-10%	8 977	20,17
Under by 11-20%	6 379	14,34
Under by 21-30%	3 674	8,26
Under by $> 30\%$	3 672	8,25

Table 3.16: Prediction categories for predictions obtained from neural network used for combining the RS and CS predictions

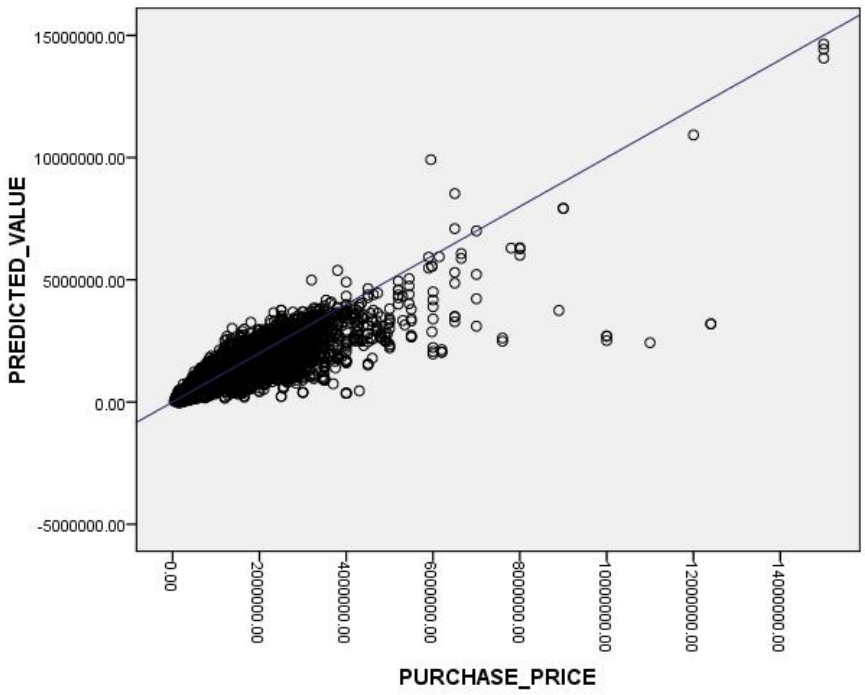


Figure 3.25: Scatterplot of *Predicted_value* against *Purchase_Price*. The 45 degree line is also displayed.

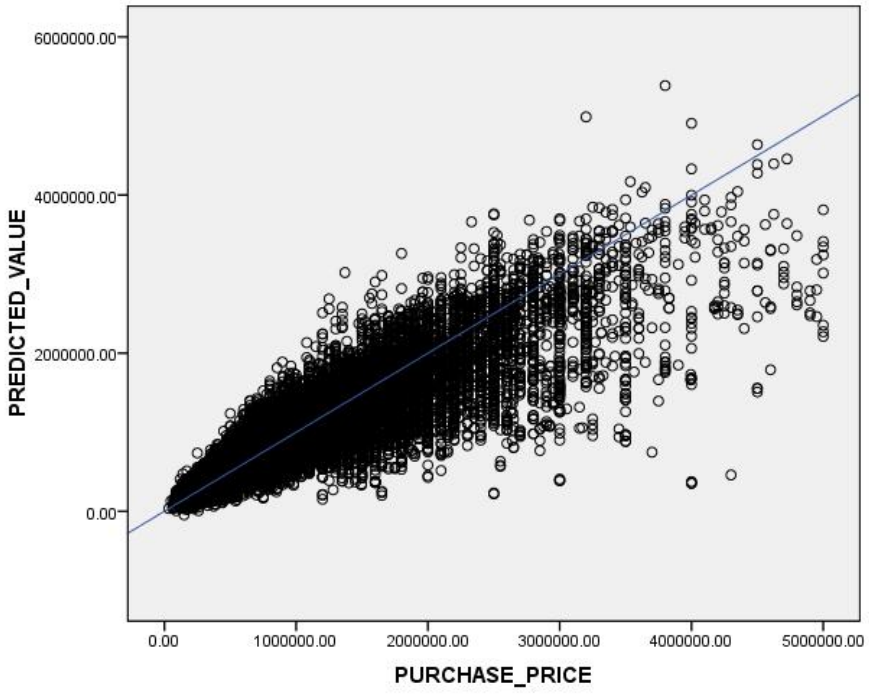


Figure 3.26: Scatterplot of *Predicted_value* against *Purchase_Price* where Purchase_Price $\in (0, 5000000)$. The 45 degree line is also displayed.

There are also quite a few observations, where the predicted value from the neural network is very low when compared to the purchase price. It is well known that a neural network does not extrapolate well to observations which are not in the range of the inputs that were used to train the neural network (Haley and Soloway, 1992). These observations may well be examples of this.

Before we are able to use this neural network in practice (cf. section 3.9), we will also need to build scoring models which will enable us to attach a confidence and safety score to each of these prediction from the neural network. This will enable us to revert back to one of the other available predictions when the predicion from the neural network does not seem to be reliable. This should increase the accuracy of this combined approach even more.

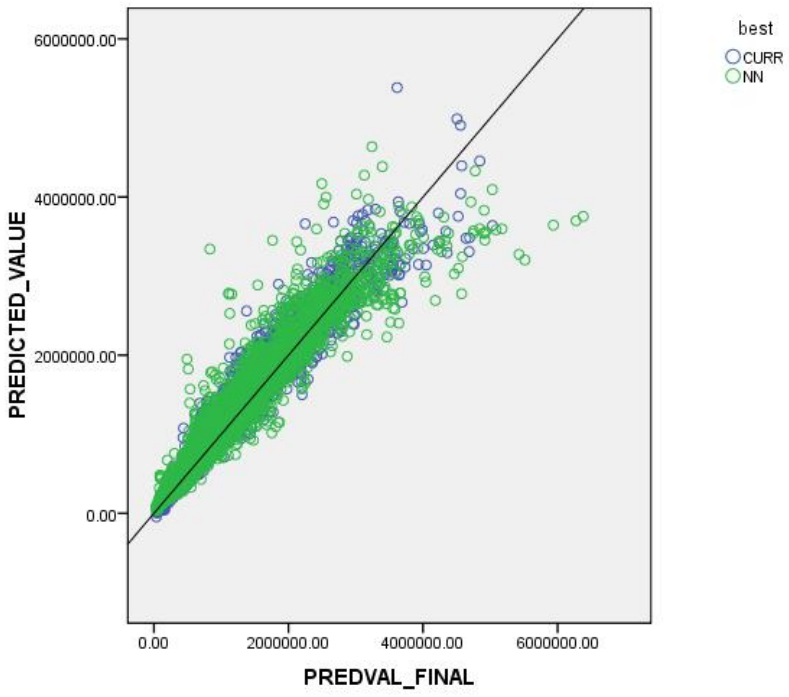


Figure 3.27: Scatterplot of *Predicted_value* against *predval_final* where Purchase_Price $\in (0, 5000000)$. The 45 degree line is also displayed. Green dots indicate observations where the *Predicted_Value* is closer to the purchase price than what *predval_final* is and blue dots indicate where *predval_final* is closer.

A plot of the predicted value from this neural network against the final prediction from

the current Lightstone approach is presented in figure 3.27. We will again only look at a contained range of values as this makes it easier to see what is going on in the plot. The green dots indicate the observations where the predicted value from the neural network is closer to the actual purchase price than the final prediction from the Lightstone approach and vice versa for the blue dots. We see that generally the prediction from the neural network and the prediction from Lightstone's approach do not differ that much. Keep in mind that the Lightstone final prediction (*predval_final*), already contains rules which will pick the best of the three available observations, i.e. will already pick the best of RS, CS or COMB predictions and therefore the prediction from the neural network does compare quite well.

It will be too risky to employ a neural network like this one on its own in a business environment. The best approach will probably be to use it as an alternative to the current combined value from Lightstone. This will involve building scoring models for the predictions and also using rules to revert back to one of the other predictions if the prediction from the neural network is deemed to be unreliable. The issue of implementation will be further discussed in section 3.9. However, if we construct a plot to compare the predictions against the current combined value from Lightstone (*predval_comb*) as in figure 3.28, we see that the predicted value from the neural network can be a good substitute in the place of the traditional combined value, although generally the predicted value from the neural network seem to be higher than the COMB prediction.

Post hoc analysis of neural network used for picking optimal prediction

We will now discuss the results from the neural network that was trained to select the best prediction from the available RS, CS & COMB predictions. From table 3.14 we see that the neural network that performed the best for this specific task was the neural network

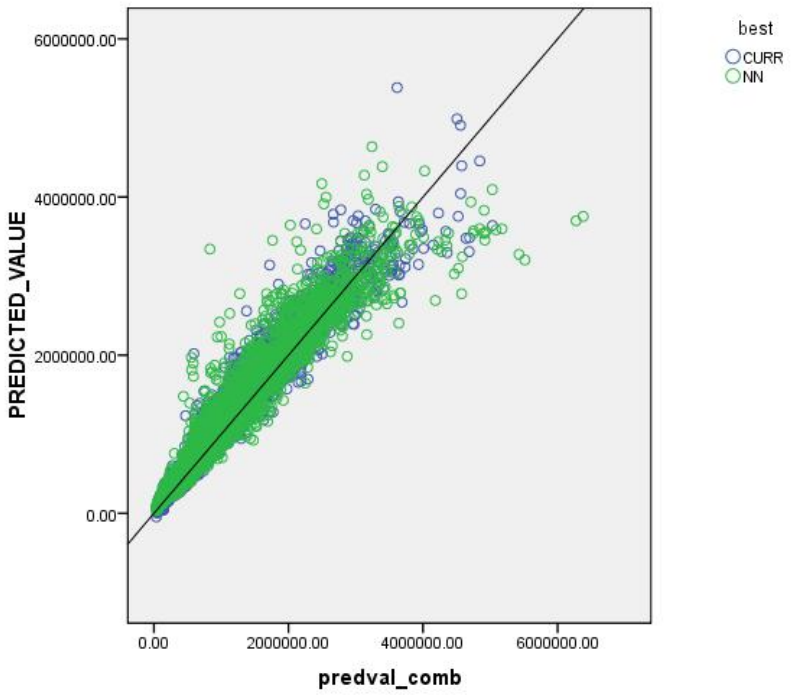


Figure 3.28: Scatterplot of *Predicted_value* against *predval_comb* where Purchase_Price $\in (0, 5000000)$. The 45 degree line is also displayed. Green dots indicate observations where the *Predicted_Value* is closer to the purchase price than what *predval_comb* is and blue dots indicate where *predval_comb* is closer.

which made use of all the available inputs (neural network number five in table 3.12). The percentage of observations which are within 20% of the actual purchase price for this neural network is 67.47 and this is a notable increase from the 66.04 which Lightstone currently achieves. Before we can continue to think about how to implement this model, we should first conduct an analysis of some of the results from this neural network. Because we are aiming to implement the model in a very sensitive environment, we should try to see why certain predictions are classified wrongly and what the impact of the missclassification is.

We will start by giving a frequency table on the outputs from this network which can be found in 3.26.

Lightstone's current approach is to choose the combined prediction except if there is enough evidence that the combined prediction is not reliable. This evidence can be based

Value of *predicted_class*	Frequency	Percentage
RS	15 977	35,90
CS	22 348	50,22
COMB	6 173	13,87

Table 3.17: Frequency table for predicted classes from neural network

on the accuracy and safety score of the COMB prediction being too low. We see from the frequency table that very few of the observations are classified to the COMB prediction, which is contrary to what Lightstone is currently doing. Therefore we will propose the following modification to the neural network and in particular the classification rule used. Recall from section 2.4.3 that a neural network which is trained using the softmax output activation function, like this neural network under consideration, approximates the posterior probabilities of an observation belonging to a specific class. Currently an observation is classified to the class which has the largest posterior probability. To be more in agreement with what Lightstone is currently doing, this classification rule will be changed to use the COMB prediction except if there is enough evidence that the RS or CS predictions should be used.

Since we are working with posterior probabilities, we will only use the RS or CS predictions if either of their posterior probabilities are relatively high, i.e. we can say with a high confidence that the RS or CS prediction is better than the COMB prediction. The new classification rule that is going to be used is as follows:

$$predicted\_class = \begin{cases} \text{RS} & \text{if } P(RS|\mathbf{x}) > 0.5 \\ \text{CS} & \text{if } P(CS|\mathbf{x}) > 0.5 \\ \text{COMB} & \text{otherwise} \end{cases} \tag{3.19}$$

$P(RS|\mathbf{x})$ and $P(CS|\mathbf{x})$ denotes the estimated posterior probability from the neural network for RS and CS respectively. The performance results from this neural network with

the modified decision rule can be found in table 3.18. We see that this approach gives a quite favorable increase in the percentage of predictions that are within 20% of the purchase price.

	Within 5%	**Within 10%**	**Within 20%**	**Within 30%**
Percentage of observations	22,25	41,38	67,75	82,45

Table 3.18: Performance of neural network using for classification using modified decision rule

This neural network with the modified classification rule gives the best performance of all the neural networks considered so far. If we do a frequency check on the classifications from this new classification rule we find:

From this frequency table we see that we are using the COMB prediction for the majority of observations which is closer to what we want. The results to show the number of over and underpredictions and also how much the predictions are over or under can be found in tables 3.20 and 3.21:.

Comparing the results from tables 3.20 & 3.21 to the Lightstone results found in tables 3.10 & 3.11 we see that this neural network does compare quite well to the current Lightstone method with regards to the number of overpredictions, underpredictions and also the severity of the over and underpredictions. We see that the percentage of predictions that are over by more than 20% is lower than that of the Lightstone method while we have a good increase in the number of predictions that are within 10% of the purchase price. This means that this neural network can at least match what the current Lightstone method

Value of *predicted_class_modified*	Frequency	Percentage
COMB	20 068	45,10
CS	14 464	32,50
RS	9 966	22,40

Table 3.19: Table displaying the frequencies of the categories for predictions obtained from neural network used for combining the RS and CS.

	Frequency	Percent
OVER	18 856	42,37
UNDER	25 642	57,63

Table 3.20: Table displaying the number of predictions from the neural network that are above and below the purchase price

	Frequency	Percent
Over by > 30%	3 054	6,86
Over by 21-30%	2 438	5,48
Over by 11-20%	4 907	11,03
Over by 0-10%	8 459	19,01
Under by 1-10%	9 687	21,77
Under by 11-20%	6 960	15,64
Under by 21-30%	4 200	9,44
Under by > 30%	4 793	10,77

Table 3.21: Prediction categories for predictions obtained from neural network used for combining the RS and CS predictions

achieves in terms of the number of underpredictions and overpredictions and also increase the accuracy of the predictions.

If we plot the values of the chosen prediction from this neural network with the modified classification rule against the actual purchase price we obtain the plot in figure 3.29.

From the plot in figure 3.29 we see that the predictions that this neural network have chosen are very good. The overpredictions are clearly in the minority. Also note that when an accurate prediction is not chosen, the chosen prediction would more likely to be below the purchase price than above. Therefore we see that this neural network has a tendency to choose predictions which are lower than the purchase price. This is a good property if we are looking to implement this neural network in the Lightstone environment.

In the plot in figure 3.30 we zoomed in on the plot in figure 3.29 to make it clearer. Here we can see that the neural network with the modified classification rule does a great job of picking the best prediction. We see that the neural network is in most cases relatively

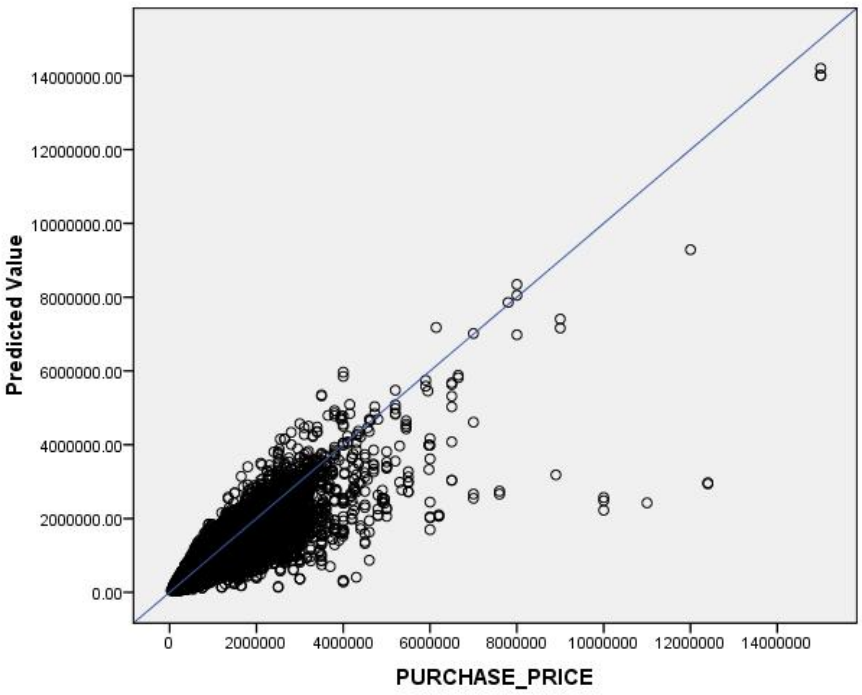


Figure 3.29: Scatterplot of *Predicted_value* against *Purchase_Price*. This predicted value is from the neural network with the modified classification rule found in 3.19. The 45 degree line is also displayed.

accurate and that the predictions are never too high and more prone to be smaller than the purchase price which is a favourable property of this neural network.

We would however still need to investigate some of the cases that are not correctly classified. What will be of particular interest is the observations for which the neural network predicts that either the RS or CS prediction should be used when the COMB is in fact the best prediction. The missclassifications that happen as a consequence of choosing the COMB prediction rather than the RS or CS predictions are not as severe since the COMB does overall give the best available prediction with the highest stability.

We find that 2 466 out of the 44 498 predictions, or 5,54% of the predictions are wrongly classified as either the CS or RS prediction when the COMB prediction is in fact the best prediction. If we classify these wrongly classified predictions into categories to see the extent to which the neural network overpredicts and underpredicts on these records we

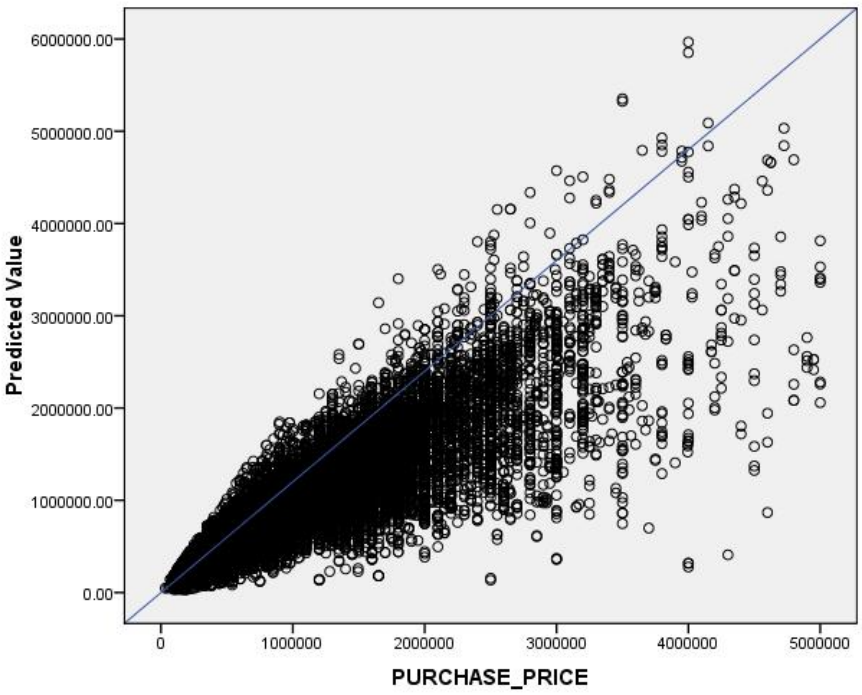


Figure 3.30: Scatterplot of *Predicted_value* against *Purchase_Price* where Purchase_Price $\in (0, 5000000)$. The 45 degree line is also displayed.

obtain the results in table 3.22.

	Frequency	**Percent**
OVER	1 921	77,90
UNDER	545	22,10

Table 3.22: Table displaying the nature of the predictions from the neural network which are misclassified as either a RS or CS prediction when the COMB seemed to be the optimal prediction

Table 3.22 shows that for the majority of these observations, the prediction that was chosen was higher than the actual purchase price. If we have a look at the extent to which these predictions overpredict or underpredict we find the results in table 3.23.

Table 3.23 shows that a large majority of these misclassified predictions fall within 10% of the purchase price and although most of these predictions are higher than the purchase price, they still lie within an acceptable range and therefore this will still be an acceptable risk for the banks. We see that more than 60% of these observations that are misclassified

	Frequency	Percent
Over by > 30%	151	6,12
Over by 21-30%	162	6,57
Over by 11-20%	521	21,13
Over by 0-10%	1 087	44,08
Under by 1-10%	442	17,92
Under by 11-20%	77	3,12
Under by 21-30%	17	0,69
Under by > 30%	9	0,36

Table 3.23: Table displaying the extent to which the predictions are under or above the purchase price for the predictions from the neural network that was wrongly classified as either CS or RS when COMB seemed to be the optimal prediction

as either RS or CS are still within 10% of the purchase price and more than 80% of these observations are within 20% of the purchase price. This means that although these observations are misclassified they are still accurate and can be used.

If we plot the percentage of observations that are within 20% of the actual purchase price across the different modelling segments for the neural network approach and also the current Lightstone approach we obtain the plot in figure 3.31. From this plot we see that the accuracy of the neural network is better than that of the current Lightstone approach across all the modelling segments except for segment G2, for which it is only a little lower but still very close to the Lightstone method.

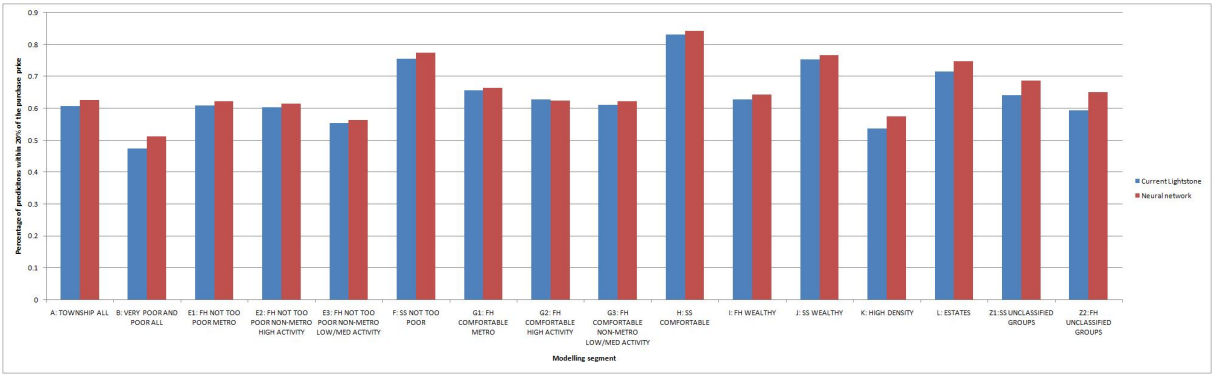


Figure 3.31: Comparison of accuracy of current Lightstone approach against the neural network approach across the modelling segments.

For the next plot, we will divide the accuracy score of the chosen Lightstone prediction

and also the accuracy score of the chosen neural network prediction into categories. This is done to construct a graph which can be used for comparison purposes. We will label these categories as *High* for an accuracy score above 0,75, *Medium* for an accuracy score between 0,5 and 0,75, and *Low* for an accuracy score of below 0,5. If we plot the percentage of predictions that are within 20% or more of the purchase price for both the current Lightstone approach and the neural network approach, according to accuracy score categories for the predictions arising from each of the two approaches, we obtain the plot in figure 3.32. From this plot, we see that the neural network method outperforms the current Lightstone method when the confidence in the chosen Lightstone prediction is *Medium* or *Low*. This can help when implementing the neural network in the Lightstone environment, since we can still use the current Lightstone method when the confidence in the prediction is high, otherwise we can use the prediction from the neural network method and thus minimising the risk from using the neural network alone, but still getting the benefits that the neural network has to offer.

The results achieved from this neural network are very good. Overall it seems that this neural network does perform better than what Lightstone currently achieves. The encouraging result is that it does not seem to use predictions which are too high and in fact decreases the number of predictions which are over the purchase price by 20% of more. This approach of using a neural network to choose the best of the available predictions definitely works better than the regression approach that was also discussed. Therefore we will use this approach and discuss how this approach can be implemented into the Lightstone environment in the next section.

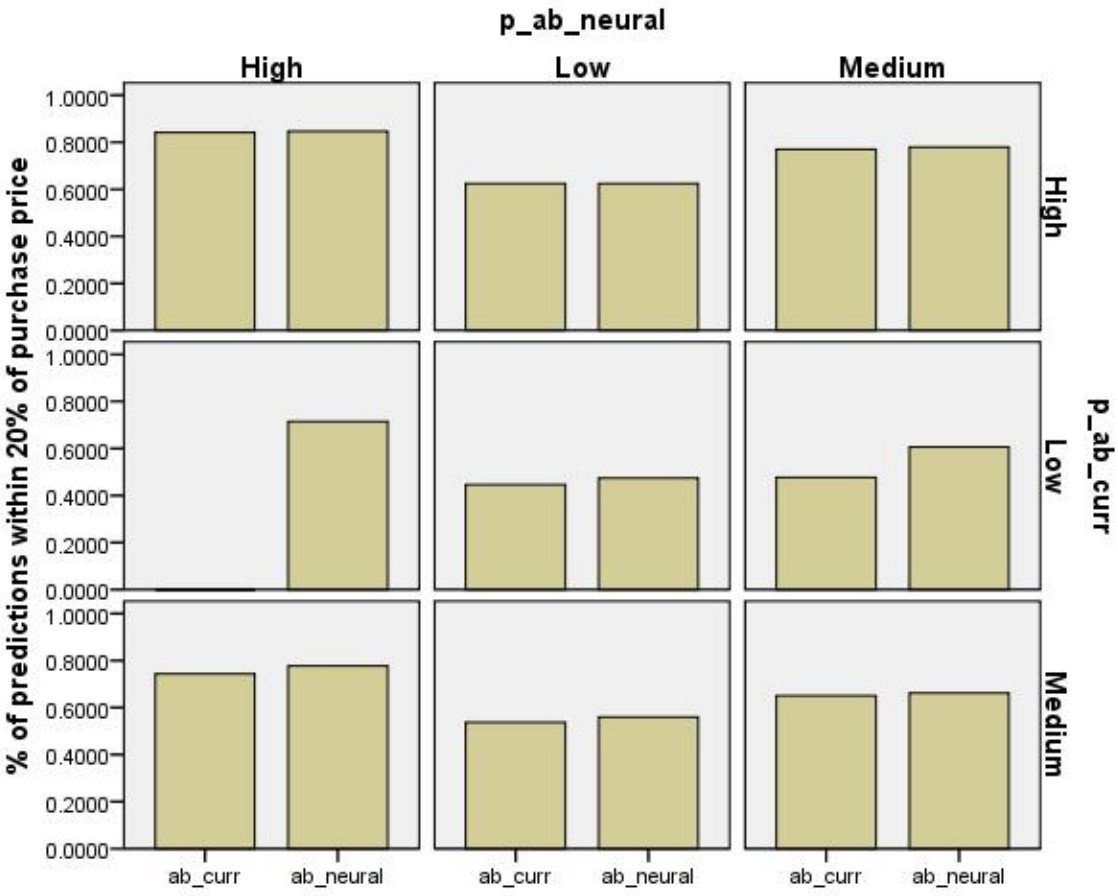


Figure 3.32: Comparison of accuracy of current Lightstone approach against the neural network approach split according to the accuracy score of the Lightstone prediction and the accuracy score of the neural network chosen prediction. For example, the plot in the top left hand corner of the figure, indicates the percentage of predictions that are within 20% of the purcase price for the Lightstone approach on the left and the neural network approach on the right, where the accuracy scores of the predictions from the Lightstone approach and also the chosen neural network predictions are high.

3.9 Implementation

The Lightstone environment is very sensitive in the sense that the accuracy of the predictions from Lightstone directly influences the risk of the banks that use the Lightstone AVM system. Therefore any new methodology should be tested very thoroughly before it can be implemented into the system. The specific area of usage for this neural network means that it can have a major impact on the quality of the predictions that Lightstone

gives and therefore this method should be tested for a few months to assess the effect. In this section, we will discuss a few of the issues which should be concentrated on when implementing this neural network and also some of the testing that should be done. The results that were given in the previous section (cf. section 3.8.1) were encouraging and suggested that the neural network which was used for picking the best prediction could outperform the current approach that Lightstone employs. In this part we will also look at some results on a new independent data set using the previously trained neural network which will already give us an idea of how the neural network will perform in practice.

Implementing the neural network into AVM system

The best neural network which was trained to pick the optimal prediction is the neural network which will be used to test for implementation into the Lightstone AVM system. This neural network which was used to pick the best prediction performed better than the neural network which was used for combining of predictions. One of the important reasons why we would rather want to use the neural network for classification than the regression case is because of the extrapolating features of a neural network. It is well known that a neural network does not extrapolate well and therefore the regression neural network is more risky to use, since it can arrive at totally incorrect values when extrapolating (Haley and Soloway, 1992). The neural network which is used for picking the optimal prediction will almost certainly do better in this case, especially with the modified classification rule given in (3.19) since it is more restricted and cannot as easily give a way-out value as in the case of the regression neural network.

The way in which this neural network is going to be used in the AVM system is to pool a couple of months' data and then train the neural network on that data, much like it was done throughout this section when the neural network was trained on mortgage

application data for January 2009 to May 2009. The neural network that was trained on this data will then be used on the next month's data to select the best prediction. At this stage we used five month's data for the training of the neural network but this may also be experimented with by adding more data or using fewer data. At this stage we will stay with using five months' data. In practice this neural network will also need to be retrained every month on the last five months' data, for it to be used for the next month.

Fortunately we have access to the mortgage application data for June 2009 and therefore the available trained neural network can be tested on totally independent data. This will allow us to see the performance of the neural network if it were to be implemented in the Lightstone system. This data set for June 2009 which will be used for testing contains 1 863 observations. This data set is not cleaned in any way since we are not modelling on this data and would like to get an accurate as possible measure of performance when the neural network is used in the AVM system. The current performance measures on this data set using the Lightstone approach is as follows:

	Within 5%	**Within 10%**	**Within 20%**	**Within 30%**
Percentage of observations	20,02	36,50	62,43	77,78

Table 3.24: Performance of current Lightstone method on mortage application data for June 2009

The percentage of predictions that are below or above the given purchase price and also the categories are given in tables 3.25 and 3.26 respectively.

	Frequency	**Percent**
OVER	754	40,47
UNDER	1 109	59,53

Table 3.25: Table displaying the number of predictions that are above and below the purchase price for June 2009 data for the current Lightstone method

If we train a neural network on the January 2009 - May 2009 data with inputs corresponding to neural network number 5 in table 3.12, and use this neural network to pick

	Frequency	Percent
Over by > 30%	163	8,75
Over by 21-30%	87	4,67
Over by 11-20%	197	10,57
Over by 0-10%	307	16,48
Under by 1-10%	373	20,02
Under by 11-20%	286	15,35
Under by 21-30%	199	10,68
Under by > 30%	251	13,47

Table 3.26: Prediction categories for the current Lightstone predictions on June 2009 data

the best available prediction by using the modified classification rule

$$predicted\_class = \begin{cases} \text{RS} & \text{if } P(RS|\mathbf{x}) > 0,5 \\ \text{CS} & \text{if } P(CS|\mathbf{x}) > 0,5 \\ \text{COMB} & \text{otherwise} \end{cases} \tag{3.20}$$

which was first presented in (3.19), we obtain the following results on the June 2009 data:

	Within 5%	Within 10%	Within 20%	Within 30%
Percentage of observations	19,97	36,50	63,39	78,69

Table 3.27: Performance of optimal neural network used to choose the best prediction on mortage application data for June 2009

If we look at the percentage of predictions that are below or above the given purchase price we obtain:

	Frequency	Percent
OVER	817	43,85
UNDER	1 046	56,15

Table 3.28: Table displaying the number of predictions that are above and below the purchase price for June 2009 data

Comparison of the results from the neural network to the current Lightstone method is very interesting. We see that percentage of predictions which are within 20% of the

	Frequency	Percent
Over by > 30%	166	8,91
Over by 21-30%	100	5,37
Over by 11-20%	234	12,56
Over by 0-10%	317	17,02
Under by 1-10%	362	19,43
Under by 11-20%	268	14,39
Under by 21-30%	185	9,93
Under by > 30%	231	12,40

Table 3.29: Prediction categories for optimal neural network on June 2009 data

purchase price is higher for the neural network than for the current Lightstone method. The increase is from 62,43% for the Lightstone method up 0,96% to 63,39% for the neural network. This is a sizeable increase in the accuracy. If we have a look at the percentage of overpredictions we see that this number is up from 40,47% for the Lightstone method to 43,85% for the neural network. If we look at the distribution of these predictions for the neural network into various error bands we see that the big increase was in the percentage of predictions that lie within 0-20% above the purchase price. This is acceptable as these final predictions are quite accurate albeit above the purchase price.

Next we will modify the classification rule of the neural network. The new classification rule is as follows:

$$predicted\_class = \begin{cases} \text{RS} & \text{if } P(RS|\mathbf{x}) > 0,6 \\ \text{CS} & \text{if } P(CS|\mathbf{x}) > 0,6 \\ \text{COMB} & \text{otherwise} \end{cases} \tag{3.21}$$

The classification rule in (3.21) implies that the large majority of observations will use the COMB prediction unless the posterior probability is relatively high indicating that one of the other two predictions should be used. A cutoff value of greater 0,6 was also experimented with but did not yield any major improvement over the current chosen value

of 0,6. The results on the June 2009 data when using this classification rule are displayed in table 3.30.

	Within 5%	**Within 10%**	**Within 20%**	**Within 30%**
Percentage of observations	20,34	37,20	63,88	79,12

Table 3.30: Performance of neural network, with classification rule give in (3.21), used to choose the best prediction on mortgage application data for June 2009.

If we look at to percentage of predictions that are below or above the given purchase price we obtain:

	Frequency	**Percent**
OVER	788	42,30
UNDER	1 075	57,70

Table 3.31: Table displaying the number of predictions that are above and below the purchase price for June 2009 data.

	Frequency	**Percent**
Over by $>$ 30%	153	8,21
Over by 21-30%	95	5,10
Over by 11-20%	223	11,97
Over by 0-10%	317	17,02
Under by 1-10%	376	20,18
Under by 11-20%	274	14,71
Under by 21-30%	189	10,14
Under by $>$ 30%	236	12,67

Table 3.32: Prediction categories for neural network, with classification rule give in (3.21), on June 2009 data

We see from the results just presented that this neural network compares even better to the current Lightstone method with a 1,45% percent increase in the percentage of predictions that are within 20% of the purchase price over the current Lightstone method. The overpredictions are also just slightly up from the Lightstone method and overall the accuracy is better than that of the Lightstone method. This can clearly be seen by comparing table 3.24 to the results in table 3.30.

If we plot the percentage of predictions that are within 20% of the purchase price across

the modelling segments for the current approach as well as the neural network (figure 3.33) we see that for the Township (A) and Poor (B) modelling segments, the accuracy of the neural network is much higher than the current approach. This is a very encouraging sign as those two segments are currently the segments in which Lightstone performs the worst as they are very difficult markets to predict in. We see that for the Comfortable segments, the neural network also gives an increase in accuracy. For the other segments the accuracy of the current and Lightstone approaches are very similar. Segments Z1 and Z2 are not a big worry as these segments usually contain very few observations (Z1 contains 13 observations in this example) and therefore accurate comparisons cannot be made.

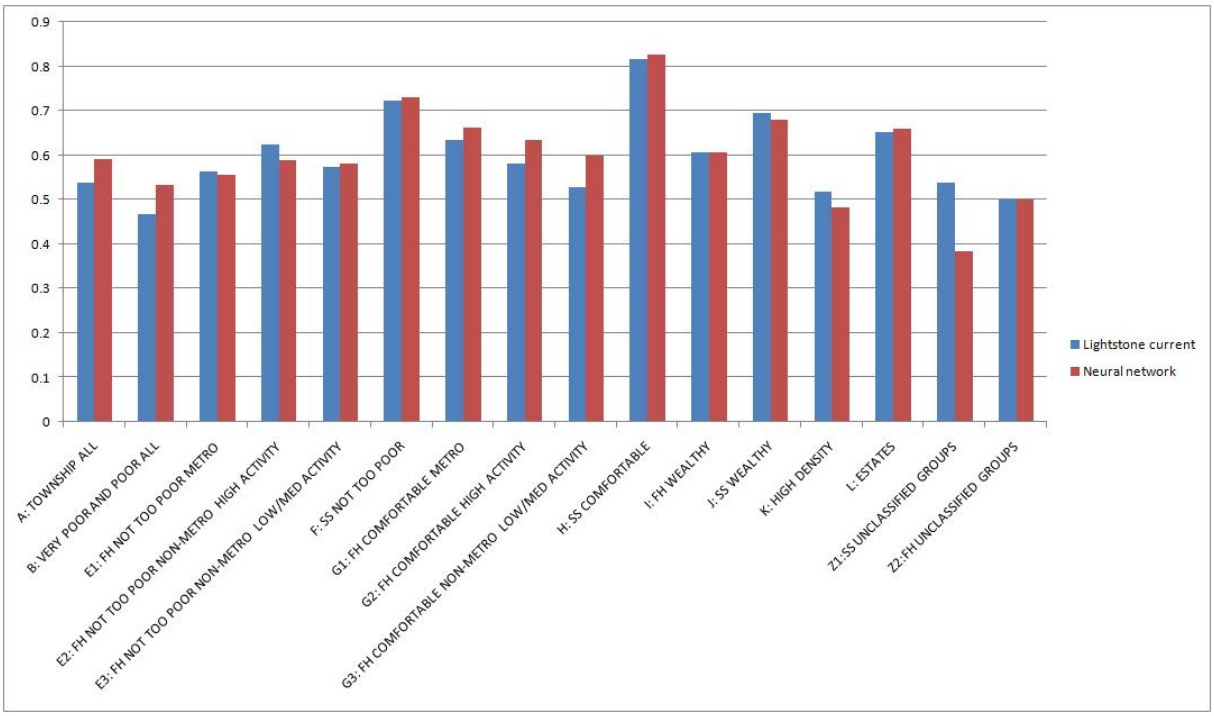


Figure 3.33: Comparison of accuracy of current Lightstone approach against the neural network approach across the modelling segments.

If we divide the accuracy score of the combined prediction into different categories and then analyse the accuracy of the two methods within each of these bands we find the plot that is represented in figure 3.35. The reason why we look at this is because the current Lightstone approach chooses the combined prediction unless there is evidence

that this prediction should not be used, then it reverts to one of the other predictions. This graph can give us insight to see if the neural network perhaps performs better when we have a low confidence in the combined prediction. This is then another way in which the neural network can be implemented, by always choosing the combined prediction, unless the accuracy of the combined prediction is below a certain level in which case we will then revert to the neural network prediction. From the graph in figure 3.34 we see that the neural network does perform better than the current Lightstone approach when the accuracy score of the combined prediction is relatively low (below 60%). This may very well be a way in which the neural network can be implemented in the Lightstone environment.

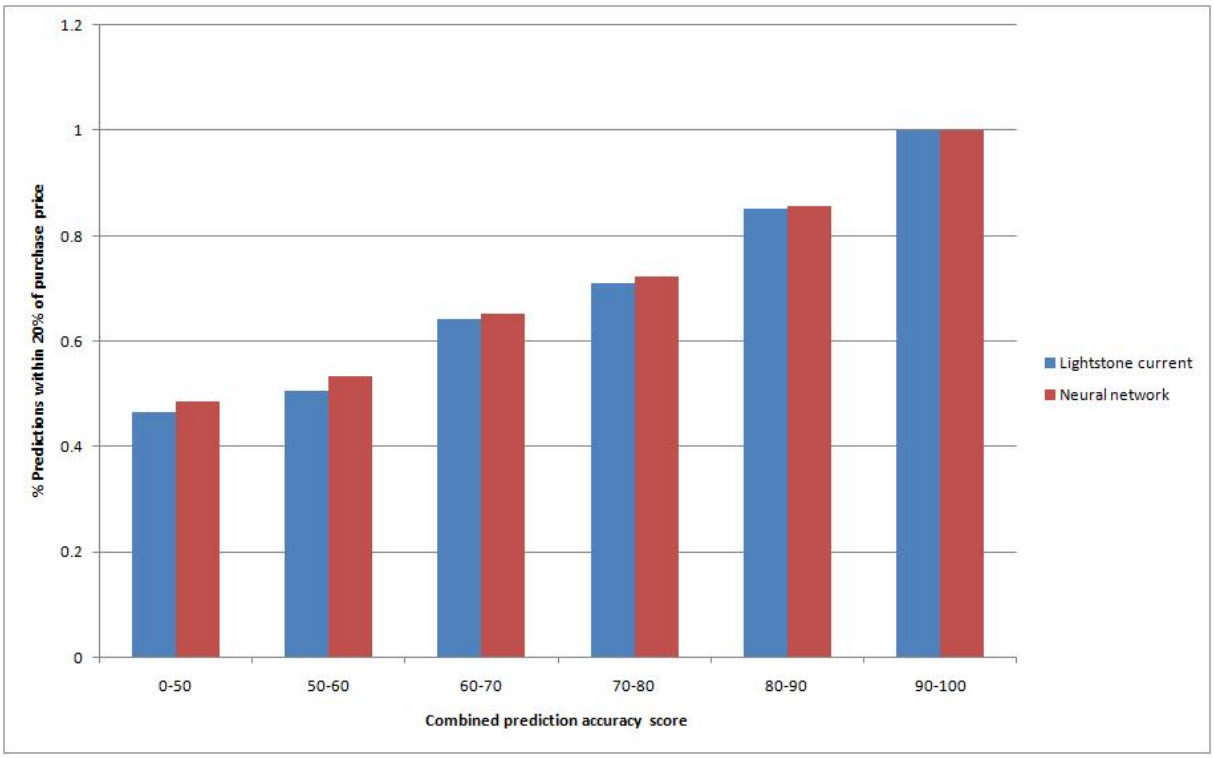


Figure 3.34: Comparison of accuracy of current Lightstone approach against the neural network approach for different values of the accuracy score of the combined prediction.

An interesting observation arises when we plot the accuracy score from the Lightstone prediction against the accuracy score of the neural network prediction and then have a look at those records for which one of the predictions are within 20% of the actual purchase price but the other prediction is not. The plot can be found in figure 3.31. From

this graph we see that there are a relatively high number of observations for which the accuracy score from the Lightstone prediction is higher than that of the neural network, but the prediction from the neural network is within 20% and the Lightstone prediction is not. This indicates to us that accuracy score alone cannot be used for choosing between predictions and this is where the neural network can improve the current approach.

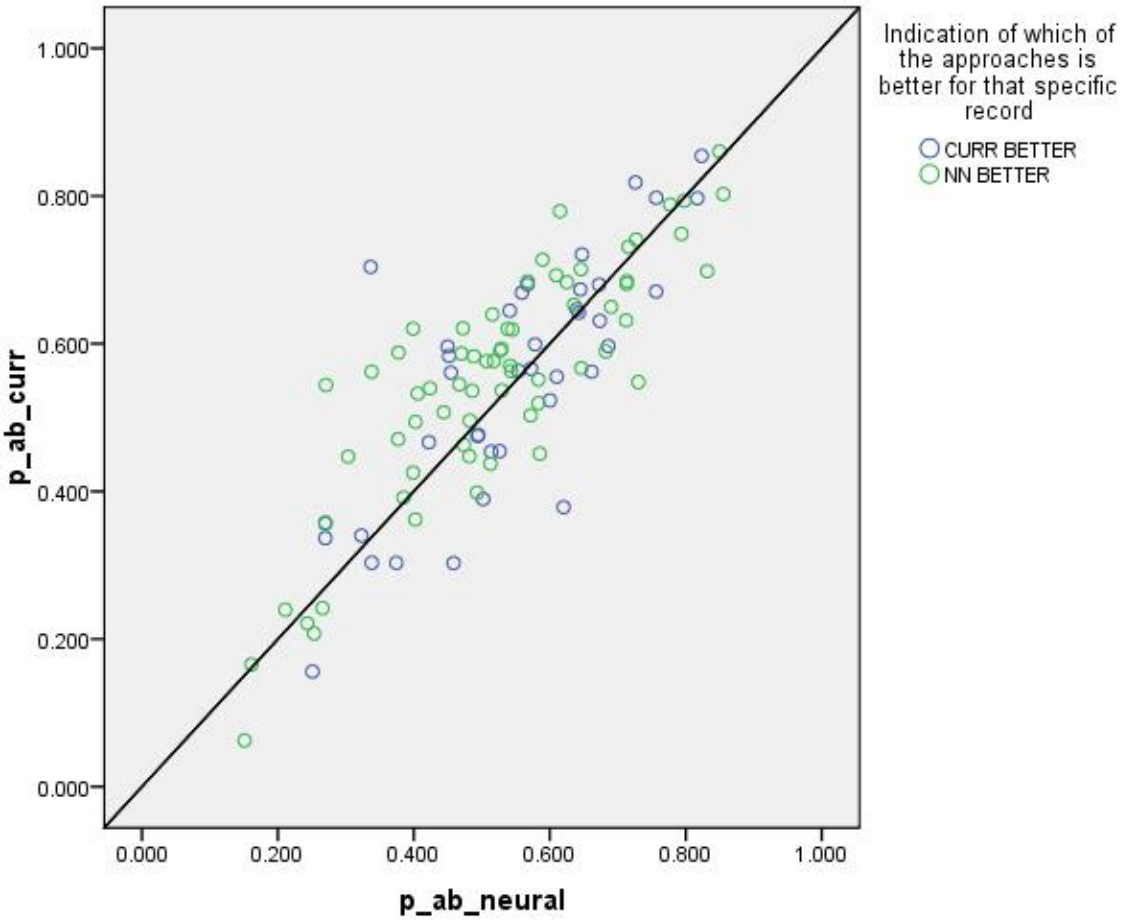


Figure 3.35: Plot of accuracy score of current Lightstone approach against the accuracy score from the neural network prediction for records where the one prediction is within 20% of the purchase price but the other prediction is not. The green circles indicate the records for which the prediction from the neural network is within 20% of the purchase price but the prediction chosen by the current Lightstone approach is not and the blue circles signifies the opposite. The 45 degree line is also indicated to make comparisons.

The neural network with the modified classification rule as given in (3.21) will still need to be tested for another couple of months before a final conclusion can be made. Testing should be done similarly to what was done in this section, i.e. by pooling a couple of

months' data into one data set, training and arriving at the optimal neural network on this data and then doing independent testing on a new month's data which was not part of the training of the neural network. From the testing that was done in this section, it seems that the neural network can outperform the current Lightstone method, especially with the classification rule as given in (3.21). It is definitely worthwhile testing this neural network for a few more months as this neural network can replace the current Lightstone method and can be used as a standalone method without the need for any additional rules.

3.10 Difficulties experienced

In this section we will focus on some problems that were encountered while building the neural network that was used for the Lightstone application. We will discuss some of the problems relating to the neural network and neural network methodology and also some of the issues encountered on the problem and the data for which the neural network was used. These issues will be presented in list form and are in no particular order:

- The commercial packages which include neural networks often have various shortcomings. One of the shortcomings which were experienced was the lack of a sufficient method to stop training the neural network. Most of the packages stop training when the weights converged, i.e. the weights do not differ a lot from epoch to epoch. This usually results in a suboptimal neural network model. The error from these packages are usually reported on an independent testing set but the testing set is not used in the training itself. This is where the neural network that was used and coded in this dissertation provides a clear benefit in that the training and testing of the neural network occurs concurrently and that optimal neural network as calculated

on the testing set is used to arrive at the final neural network. On average the neural network which was used for picking the optimal prediction on the Lightstone data improved with 3% on the percentage of predictions within 20% of the purchase price when compared to the neural networks which was trained using some of the commercial software packages.

- Backpropagation training is very slow. SAS IML was used as the software of choice because it is relatively fast with matrix and algebra computations and even with this software running on a very powerful server the neural network still took a long time to train. For the neural networks which were specified in section 3.6.3 we found that the first neural network with the smallest number of inputs took on average around 1 hour and 15 minutes to complete 1 000 epochs while neural network number five, which has the most inputs, took on average around 6 hours to complete 1 000 epochs. This shows that a neural network is very computer intensive especially when the training data set is fairly large.

- The Lightstone problem that was used in this dissertation for application of neural networks is quite a unique problem and is not a typical statistical problem which one often finds. Some of the variables in this problem like the RS and CS predictions and also the accuracy and safety scores are predictions from underlying models themselves and are not measured values like in most common statistical applications. These predictions are already subject to error and then by using these variables as independent variables in a statistical model means that the error gets compounded. The problem with this is that the neural network can only be as good as the data allows it to be and therefore the underlying models which provide the inputs to the neural network also need to be as accurate as possible for the best possible results from the neural network.

- If we look at the percentage of observations where we have a prediction that is within 20% of the purchase price on the data set that was used to train the neural network,

we find that the theoretical maximum is 78,09%. This means that potentially we could get almost 80% of the observations from Lightstone within 20% of the purchase price. However, this is a very misleading and over optimistic number. This point will be further illustrated with an example that is taken from the actual data. For this example, we will keep things as simple as possible and only work with the relevant predictions and their accuracy and safety scores. Consider the following record:

Purchase_Price	$\hat{y}_{RS}$	AS_{RS}	SS_{RS}	$\hat{y}_{CS}$	AS_{CS}	SS_{CS}
600 000	601 007	0,64	0,55	501 660	0,80	0,98
$\hat{y}_{COMB}$	AS_{COMB}	SS_{COMB}				
528 293	0,80	0,97				

From this record, we see that the RS prediction is almost spot on with the purchase price but if we look at the accuracy score of the RS prediction we might conclude that it is safer to go with the COMB prediction which has a relatively high accuracy score. The neural network that was trained also predicted that for this record the COMB prediction is the best prediction. Even though the COMB prediction is still within the reasonable 20% range of the purchase price, we could have been a lot more accurate had we chosen the RS prediction. There are two main reasons which can explain records like this one. The first was already touched on in the previous point in that the accuracy and safety score are also predictions from models and are already subject to error. The second reason is that a property value is a very subjective thing and also extremely difficult to determine exactly. There are various factors which influence the price for which a property sells for. If a buyer is in a bad financial position and quickly needs to sell his property, he is more likely to accept a lower price for the property than someone who is not a desperate seller. A willing buyer may also be prepared to buy a property for more than it is worth. Therefore each property has its own price distribution and a property can sell anywhere on that distribution depending on the circumstances of both the buyer and the seller.

For the record that is presented above, the 528 293 may be a very accurate value for that property, but the seller has put it in the market for a higher price and was in the position to wait until he got the price that he asked, which reflects in the higher purchase price. Therefore although we had a prediction for this property which was almost spot on with the purchase price, there was no information to tell the neural network that this was indeed the better prediction.

This leads us to the point that we have to make a distinction between the theoretical maximum accuracy on this data set, and what is possible in practice. The record in the example is not the most extreme example of this as all the predictions still fall within the 20% of the purchase price and there are other cases where only one of the predictions fall within 20% of the purchase price but there is no information to tell the model that this is indeed the best prediction. For these records the neural network will often choose the best prediction given the input data, although this prediction will often not be the closest to the purchase price. Unfortunately it is not a simple task to calculate the maximum accuracy that can be attained in practice on this data set.

- For the neural network that was used to pick the correct prediction we had to calculate the optimal prediction by using the data we had at hand. This posed a problem because the dependent variable influences the fitted neural network and there were quite a few approaches which could be followed to calculate the optimal dependent variable. We opted for the easiest method in that we used the prediction that was closest to the purchase price which seemed to work quite well. However, if we have a look at the frequency table of the dependent variable in the neural network (cf. figure 3.6) and compare this against the frequency table for the predicted outputs from the neural network (cf. table 3.17) we see that the priors probabilities and the output class probabilities differ. This may give an indication that the dependent variable is not entirely correctly specified and this is something which can be looked into when trying to increase the performance of the neural network (cf. section

3.12). This issue is closely related to the previous point in that the actual value of the property and the purchase price of the property can differ quite significantly because of various factors, which in turn can make it difficult to assign the correct prediction to each observation.

3.11 Final remarks

In this section we will share thoughts and experiences while using neural networks. Most of these remarks will come from a statistical point of view and this section will closely tie in with the next section where I discuss some future research opportunities for neural networks, also from a statistical point of view. Some of my thoughts regarding neural networks from a theoretical and practical point of view are:

- Coming from a statistical background and trying to use a neural network in the same manner as a regression analysis is quite a challenge, but can be very rewarding in terms of the performance that a well trained and carefully planned neural network can produce.
- Building an efficient neural network which can be used for predictive inference is not an easy task. There is always an element of uncertainty when using neural networks since it is a black-box model and it is difficult to explain why certain predictions occur the way they do.
- Deciding on the relevant inputs to include in a neural network is not easy. The problem is that one cannot entirely proceed to do an exploratory analysis phase like in a statistical regression, since the sample sizes for neural network data sets are often very large with lots of variables. Therefore it is difficult to distinguish between statistical significance and practically significance. The reason for this is

that if one only looks at correlations between variables in these large data sets, because of their large size, very small correlations between the variables will turn out to be statistically significant although they may not be of practical importance or may not add anything useful to a neural network model.

- Neural networks are very sensitive to the input data. The old saying of garbage in, garbage out has never been so true as it is with neural networks. Therefore, one should put considerable time and effort into choosing the relevant input variables and also cleaning data by removing outliers. This is where statistical knowledge can be very advantageous.

- Neural networks' predictive ability does not seem to be influenced by multicollinearity. This seem to be in line with what Carpio and Hermosilla (2001); De Veaux and Ungar (1994b) found, although more work is necessary on this topic before any conclusions can be made.

- Neural networks need tweaking to get the best predictive ability from them. For this dissertation, we started by using a few of the commercial software packages that include neural networks and none of them could even get close to the performance that was obtained from the one coded in IML. Although the coded one was a lot simpler than the neural networks in the software packages, it could also be customised more. The one customisation that improved performance was the change of the stopping criterion which we made specific to the problem at hand. This can of course not be done with a commercial software package.

- When neural networks first arrived on the scene, lots of exaggerated claims where made that they can be used to model any process without the need of any statistical knowledge. From the experience with building the neural networks in this dissertation, I believe that this is very far from the truth and that a neural network should in no case be used as a blackbox model without at least analysing some of the results and trying to understand the underlying process. Statistical knowledge

is still very much needed with the use of neural networks and may very well prove to be the difference between a neural network that performs poorly and is unreliable and a neural network that performs very well.

- Statisticians should get involved in neural networks. Many of the issues in neural networks are closely related to statistics and many of the shortcomings of neural networks at this stage will need to be researched by a person with a strong knowledge of statistical theory. If statisticians do get involved in research in neural networks, neural networks can potentially become one of the most powerful predictive modelling tools that a statistician can have available, especially in this era where statisticians are often confronted by very large data sets.

3.12 Future work

This section will be used to give ideas of what can be done in the future to improve the performance of this particular neural network and also future research ideas for neural networks from a statistical point of view. These future research ideas will be based on my own experience that I had while building the neural network that is presented in this dissertation. I will give the difficulties that I encountered while using neural networks. For some of these difficulties I could find no satisfactory answer in the literature and it could be worthwhile to research some of these topics more. A list of topics for this is:

- One of the first things I noticed when starting with this problem was the importance of standardising the inputs when using backpropagation. For the first of the simulation programs that were written, observations from two perfectly linearly separable groups were simulated and the aim was to train a neural network to classify each of the observations. Now with the groups being perfectly separable, one expects that

a classification method should obtain a zero percent misclassification rate. None of the neural networks that were trained could classify every single observation correctly. The inputs were then standardised and the result was that the neural network classified all the observations correctly. The same problem was then taken to a commercial software package and the same problem occurred. No answer to this could be found in the literature and most of the literature just state that it is a good idea to standardise the inputs without specifying why. It seemed that the neural network was more prone to converge to a suboptimal local minimum when the inputs were not standardised. It might be worthwhile to research this and see exactly why this is.

- The Lightstone problem of picking the best possible prediction of those given is quite an unique problem and not typically a common statistical problem. Research should be done on other methods, both statistical and nonstatistical, which can be used to solve a problem like this. A decision tree may also be considered as an alternative method to this problem that might work well.

- In section 3.10 it was mentioned that the Lightstone problem does not actually fit the typical statistical application in the sense that the independent variables are predicted values themselves and also subject to prediction error. It might be worthwhile to research methods that are more suited to this type of problem or otherwise research how statistical methods can be used to handle problems like this with a special emphasis on statistical inference on data of this nature.

- Another statistical technique which might be worthwhile to try on this specific problem is the nonparametric technique called multivariate adaptive regression splines (MARS). This technique is also nonparametric but the coefficients obtained from this technique are interpretable since the basis function used to fit the splines have a specific form. This might help give a better understanding of the process.

- We could consider increasing the number of hidden layers for the neural network

that was used to pick the best prediction to see if there is any improvement.

- We could also use other types of learning rules for this problem to see if there is any performance gain, both in speed and in predictive ability of the neural network.

- If one looks at the tables in appendix A.5 which contain the performance of each of the five runs conducted we see that the different starting points can have a huge impact on the performance of the neural network. There ought to be a better way to choose these starting points than just randomisation to small numbers. This can be included in future research.

- We could also experiment with different ways of calculating the dependent variable to be used in the neural network for picking the best prediction. The issues around this were discussed in section 3.10.

- A modification which may increase the performance on this specific problem is to use a varying learning rate instead of a fixed learning rate. This is something which will need to be included in the SAS IML program that was used for training the neural network and although it is conceptually an easy modification, it is not so easy to bring it into the program and this will also need to be experimented with to see what works the best.

Chapter 4

Conclusion

In this dissertation is was shown how a neural network can be used as a tool for statistical regression and classification purposes. The methodology was explained in a manner which is familiar to persons with a statistical background. The necessary background knowledge of nonlinear optimisation in a nonlinear regression context and also nonparametric regression was given to make it easier to see how a neural network will fit into a regression framework.

A neural network can be described as a flexible nonparametric regression method which can be used for both regression and classification applications. The major difference between conventional statistical regression techniques and a neural network model is in the way the weights are estimated. An extensive literature study was conducted into how a neural network can be expressed as a regression model and how it can be used for modelling purposes. A two-layer neural network can be explained as a two step regression, where the first regression takes the original independent variables and combines them in a nonlinear way to arrive at derived features. The second regression takes these derived features and then performs another nonlinear regression on these variables to arrive at

the output. This whole process functions as a single unit and backpropagation is an easy way to calculate the weights of this regression.

After the literature study we used a neural network in an application. We started with simulations to show what the function of the hidden nodes is. Simulations were also used to show the most effective way to train a neural network. The approach that was followed was to split the training data into a modelling and testing data set and do the training and testing concurrently while training a neural network with a relatively large number of hidden nodes. This ensures that the neural network is complex enough to fit all the nonlinearities in the data while limiting the effect of overfitting.

A neural network was also used for a very interesting practical application. The problem for which the neural network was used is not a typical statistical application. One of the major differences of this problem to a normal statistical is that some of the independent variables that are considered for this problem are predictions from models and not measured values. These predictions are also subject to prediction error and this compounds the error in the model. This made me reflect on how one should actually approach problems like these and whether normal statistical methods can be used in such applications. This is still an unanswered question and opens up a wide field of research.

The practical application that was considered here is currently handled with a rules-based approach. We used a neural network approach to this problem and two main types of neural networks, each with a different goal in mind, were used. The first is an example of a neural network with a continuous output. This type of neural network will typically be used for regression in statistical applications. The second kind of neural network was used to calculate posterior probabilities of classification. This type of neural network will typically be used for discriminant analysis applications in statistics. Both of these approaches were considered on the Lightstone data. The challenge was to determine the

optimal variables to use as inputs to the neural networks. We considered different subsets of inputs variables to determine what worked the best.

The final regression neural network that was used did perform better than the current Lightstone approach with regards to the percentage of predictions that are within 20% of the purchase price. The problems that were encountered with this approach is that the neural network had the tendency to overpredict much more than the current approach of Lightstone, which is not ideal. The other problem with this neural network is that a neural network does not extrapolate well. This means that some of the predictions from the regression neural network can be far off, especially if the input observation looks different to the training data. Therefore it is recommended that this approach should be used to augment the current Lightstone approach in some way, rather than substituting the current approach.

The neural network which was used to pick the best prediction performed better than the current Lightstone approach and although the number of overpredictions increased slightly, the neural network is still accurate enough to be considered as an alternative to the current Lightstone approach. This neural network was tweaked to fit the specific application for which it will be used. The two tweaks that brought about the greatest improvement in accuracy was modification of the stopping criterion used in training so as not to give the lowest misclassification rate, but rather the highest accuracy in terms of the number of chosen predictions that are within 20% of the actual value. This application is a rather unique case and this change in the stopping criterion increased the accuracy of the predictions from this neural network quite significantly. The second change we made to the neural network was that we used posterior probabilities that were output from the neural network and changed the classification rule according to what will work the best in the Lightstone environment.

This neural network was also tested on a new month's data which was not part of the training process. On this new independent data set we also had an increase in performance over the current Lightstone approach. This test gives a good indication if the neural network will work in practise in Lightstone and the results that were obtained were encouraging and indicated that the neural network is indeed viable for implementation in the Lightstone AVM.

In this dissertation it was shown how a neural network can be used for statistical regression or statistical classification tasks. It was also shown that a lot of effort goes into building a neural network that will perform optimally. Statistical knowledge is very advantageous when building a neural network as one can approach the neural network in a similar fashion as when building a regression model. We have also discussed the problems that big data sets create on normal statistical tests and therefore one should also test various neural networks to decide which works better.

Overall we conclude that a neural network can improve the performance of the Lightstone AVM. We also made a few recommendations with regard to topics that can be looked at in future research. These topics include research in neural networks from a statistical point of view and also specific aspects of the network that can be experimented with to increase the performance of the neural network to be used in the Lightstone AVM.

Bibliography

Anderson, J., Rosenfeld, E., 1988. Neurocomputing: Foundations of Research. MIT Press.

Bailey, M., Muth, R., Nourse, H., 1963. A regresion method for real estate price index construction. Journal of the American Statistical Association. 58 (304), pp.933–942.

Basheer, I., Hajmeer, M., 2000. Artificial neural networks: fundamentals, computing, design and application. Journal of Microbiological Methods 43, pp3–31.

Battiti, R., 1992. First- and second-order methods for learning: Between steepest descent and newton's method. Neural Computation 4, pp.141–166.

Beale, R., Jackson, T., 1990. Neural Computing: An Introduction. Institute of Physics Publishing.

Berka, P., Rauch, J., Zighed, D., 2009. Data Mining and Medical Knowledge Management. Medical Information Science Reference.

Bishop, C. M., 1995. Neural networks for pattern recognition. Oxford.

Boden, M., 2001. A guide to recurrent neural networks and backpropagation. [online] <URL: http://www.ctan.org/tex-archive/macros/latex/contrib/custom-bib/merlin.pdf> [Accessed on 2009.07.12].

Bowman, A., Azzalini, A., 1997. Applied Smoothing Techniques for Data Analysis: The Kernel Approach with S-Plus Illustrations. Oxford: Oxford University Press.

Bridle, J., 1989. Probalistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In: Fogleman-Soulie, F., Herault, J. (Eds.), Neuro-computing: Algorithms, Architectures and Applications. New York: Springer.

Bryson, A., Ho, Y. C., 1969. Applied Optimal Control: Optimization, Estimation, and Control. Blaisdell, New York.

Carpio, K., Hermosilla, A., 2001. On multicollinearity and artificial neural networks.

Cherkassky, V., Friedman, J., Wechsler, H., 1994. From Statistics to Neural Networks. Springer-Verlag.

College, W., Veaux, R. D. D., Veaux, R. D. D., 1995. A guided tour of modern regression methods.

Cristianini, N., Shawe-Taylor, D., 2000. An introduction to Support Vector Machines and other kernel-based learning methods. Cambridge University Press.

De Veaux, R., Ungar, L., 1994a. A brief introduction to neural networks.

De Veaux, R., Ungar, L., 1994b. Multicollinearity: A tale of two nonparametric regressions.

DiNardo, J., Tobias, J., 2001. Nonparametric density and regression estimation. The Journal of Economic Perspectives 15(4), pp.11–28.

Draper, N. R., Smith, H., 1998. Applied Regression Analysis. Wiley.

Dreiseitl, S., Ohno-Machado, L., 2002. Logistic regression and artificial neural network classification models: a methodology review. Journal of Biomedical Informatics 35, pp.352–359.

Duda, R., Hart, P., 1973. Pattern classification and scene analysis. New York : Wiley.

Eubank, R., 1988. Spline smoothing and nonparamteric regresion. New York and Basel.

Fahlman, S., 1988. An empirical study of learning speed in back-propagation networks. Tech. Rep. CMU-CS-88-162, Carnegie Mellon University.

Fausett, L., 1994. Fundamentals of Neural Networks. Prentice Hall.

Fiesler, E., Beale, R., 1997. Handbook of Neural Computation. IOP Publishing and Oxford University Press.

Fleming, W., 1965. Functions of Several Variables. Addison-Wesley.

Fletcher, L., 2002. Statistical modelling by neural networks. Ph.D. thesis, University of South Africa.

Fox, J., 2000. Nonparametric Simple Regression: Smoothing Scatterplots. Thousand Oaks CA:Sage.

Fox, J., January 2002. Nonparametric regression. Appendix to An R and S-Plus Companion to Applied Regression.

Frank, I., 1995. Modern nonlinear regression methods. Chemometrics and Intelligent Laboratory Systems 27, pp.1–9.

Friedman, J., Stuetzle, W., 1981. Projection pursuit regression. Journal of American Statistical Association 76, pp 817–823.

Gallant, A. R., 1975a. Nonlinear regression. The American Statistician 29, pp.73–81.

Gallant, A. R., 1975b. Seemingly unrelated non-linear regressions. Journal of Econometrics 3, pp.35–50.

Gallant, A. R., 1987. Nonlinear statistical models. Wiley.

Green, P., Silverman, B., 1994. Nonparametric regression and generalized linear models. Chapman & Hall New York.

Haley, P., Soloway, D., 1992. Extrapolation limitations of multilayer feedforward neural networks. International Joint Conference on Neural Networks 4, pp.25–30.

Hand, D., 1981. Discrimination and Classification. Wiley.

Hartley, H., 1961. The modified gauss-newton method for fitting of non-linear regression functions by least squares. Technometrics 3 (2), pp.269–280.

Hassoun, M., 1995. Fundamentals of Artificial Neural Networks. MIT Press.

Hastie, T., Tibshirani, R., Friedman, J., 2001. The elements of statistical learning. Springer.

Hebb, D., 1949. The Organization of Behaviour. New York: John Wiley & Sons.

Hertz, J., Krogh, A., Palmer, R., 1991. Introduction to the theory of neural computation. Addison-Wesley.

Hill, T., Lewicki, P., 2006. Statistics: methods and applications: a comprehensive reference for science, industry, and data mining. Statsoft Ltd.

Hopfield, J., 1982. Neural networks and physical systems with emergent collective computational abilities 79.

Hornik, K., 1991. Approximation capabilities of multi-layer neural networks. Neural Networks 4 (2), pp.251–257.

Hornik, K., Stinchcombe, M., White, H., 1989. Multilayer feedforward networks are universal approximators. Neural Networks 2, pp.359–366.

Hosmer, D., Lemeshow, S., 1989. Applied Logistic Regression. John Wiley and Sons.

Hwang, J.-N., Lay, S.-R., Maechler, M., Martin, D., Schimert, J., 1994. Regression modeling in back-propagation and projection pursuit learning. IEEE Transactions on neural networks 5, pp.342–353.

Ildiko, F., 1995. Modern nonlinear regression methods. Chemometrics and Intelligent Laboratory Systems 27, pp.1–9.

Jennrich, R., 1969. Asymptotic properties of non-linear least squares estimators. The Annals of Mathematical Statistics 40, pp.633–643.

Johnson, R., Wichern, D., 2002. Applied Multivariate Statistical Analysis. Prentice Hall.

Kennedy, W. J., Gentle, J. E., 1980. Statistical Computing. New York and Basel.

Kolen, J., Pollack, J., 1991. Backpropagation is sensitive to initial conditions. Advances in Neural Information Processing Systems 3, pp.860–867.

Levenberg, K., 1944. A method for the solution of non-linear problems in least squares. Quarterly journal of Apllied Mathematics 2 (2), pp.164–168.

Malinvaud, E., 1966. Statistical Methods of Econometrics. Rand McNally and Company.

Malinvaud, E., 1970. The consistency of nonlinear regressions. The Annals of Mathematical Statistics 41, pp956–969.

Marquardt, D., 1963. An algorithm for least-squares estimation of non-linear parameters. Journal of the Society of Industrial and Applied Mathematics 11 (2), pp.431–441.

McClelland, J., Rumelhart, D., 1986. Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations. Cambridge: MIT Press.

McClelland, J., Rumelhart, D., 1988. Explorations in the Parallel Distributed Processing. Cambrige: MIT Press.

McCullagh, P., Nelder, J., 1983. Generalized Linear Models. Chapman & Hall.

McCulloch, W., Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 5, pp.115–133.

Meese, R., Wallace, N., 1997. The construction of residential house price indices: A comparison of repeat-sales, hedonic-regression and hybrid approaches. Journal of Real Estate Finance and Economics 14:1-2, pp.51–74.

Michie, D., Spiegelhalter, D., Taylor, C., 1994. Machine Learning, Neural and Statistical Classification. NY: Ellis Horwood.

Minsky, M., Papert, S., 1969. Perceptrons: An Introduction to Computational Geometry. MIT Press.

Parker, D., 1985. Learning logic. Tech. Rep. TR–47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA.

Ratkowsky, D. A., 1983. Nonlinear Regression Modeling. New York and Basel.

Richard, M., Lippmann, R., 1991. Neural network classifiers estimate a-posteriori probabilities. Neural computation 3 (4), pp.461–483.

Ripley, B., 1994. Neural networks and related methods for classification. Journal of Royal Statistical Society 56 (3), pp.409–456.

Ripley, B., 1996. Pattern Recognition and Neural Networks. Cambridge University Press.

Rosenblatt, F., 1958. The perceptron: A probalistic model for information storage and organization in the brain. Psychological Review 65(6), pp.386–408.

Rosenblatt, F., 1962. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Spartan Books.

Rumelhart, D., Hinton, G., Williams, R., 1986. Learning representations by back-propagating errors 323, 533–536.

Sarle, W., 1994. Neural networks and statistical models. In: Proceedings of the Nineteenth Annual SAS Users Group International Conference.

Sarle, W., 1997. Neural network FAQ: Periodic posting to the usenet newsgroup comp.ai.neural-nets. [online] <URL: ftp://ftp.sas.com/pub/neural/FAQ.html> [Accessed on 2008.07.03].

Schalkhoff, R., 1992. Pattern Recognition: Statistical, Structural and Neural approaches. John Wiley and Sons Inc.

Schumacher, M., Reinhard Rossner, R., Vach, W., 1996. Neural networks and logistic regression: Part I. Computational Statistics and Data Analysis 21(6), pp.661–682.

Seber, G. A. F., Wild, C. J., 1989. Nonlinear regression. Wiley.

Shewchuk, J. R., 1994. An introduction to the conjugate gradient method without the agonizing pain. [online] <URL: http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf> [Accessed on 2008.06.18].

Stern, H., 1996. Neural networks in applied statistics. Technometrics 38 (3), pp.205–214.

Svozil, D., Kwasnivcka, V., Posp'ichal, 1997. Introduction to multi-layer feed-forward neural networks. Chemometrics and Intelligent Laboratory Systems 39, pp.43–62.

Thisted, R. A., 1988. Elements of statistical computing. Chapman and Hall.

Vapnik, V., 1996. The Nature of Statistical Learning. Springer-Verlag.

Wang, F., Zorn, P., 1997. Estimating house price growth with repeat sales data: What's the aim of the game? Journal of Housing Economics 6, pp.93–118.

Warner, B., Misra, M., 1996. Understanding neural networks as statistical tools. The American Statistician. 50(4), pp.284–293.

Waugh, S., Adams, A., 1997. A practical comparison between quickprop and backpropagation. Tech. rep., Artificial Neural Network Research Group Department of Computer Science University of Tasmania.

Werbos, P., 1974. Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. thesis, Harvard University.

White, H., 1989. Learning in artificial neural networks: a statistical perspective. Neural Computation 1 (4), pp.425–464.

White, H., 1992. Artificial Neural Networks: Approximation and Learning Theory. Oxford, UK: Blackwell.

Widrow, B., Hoff, M., 1960. Adaptive switching circuits. Pages 96-104 of: 1960 IRE Western Electric Show and Convention (WESCON) Record.

Zhang, G., 2000. Neural networks for classification: A survey. IEEE Transactions on systems, man, and cybernetics. 30 (4), pp.451–462.

Appendix A

Appendix

A.1 Comparison of neural network literature terminology and statistical literature terminology

Neural network literature term	**Statistical term**
Training, learning, adaption	Estimation or optimisation of parameters in model
Supervised learning	Both independent variables and response is present like in a regression
Unsupervised learning, Self-organisation, Competitive learning	Only independent variables are present like in cluster analysis or principal component analysis
Mapping, function approximation	Regression
Classification	Discriminant analysis
Training set	Training sample
Test set	Hold-out sample
Features	Variables
Inputs	Independent variables, predictors, regressors, explanatory variables
Outputs	Response, predicted values or fitted values
Target values	Response variables, dependent variables or observed values
Weights	Coefficients (Regression coefficients) or parameter estimates
Bias term	Intercept or constant term
Binary (0/1), Bipolar($-1/1$)	Binary, dichotomous
Training pair, exemplar or pattern	Observation containing dependent and independent variables
Errors	Residuals
Noise	Error term
Cost function or error function	Estimation criterion
Generalising from noisy data	Statistical inference
Backpropagation	An optimisation method similar to the method of steepest descent
Activations	Value of a particular variable
Epoch	Iteration, Cycle through all the training data
Weight decay	Shrinkage method, Ridge regression

Table A.1: Comparison of terminology in the neural network literature and statistical literature (Sarle, 1997)

A.2 Notation used to describe multilayer neural networks

A description of the notation that is going to be used in this dissertation will be given here. This notation is the same as was used in the literature review but a complete summary of the notation will be given to make things easier. We will start with the weights: The weights that connect the input units to the hidden units will be denoted by w_{1jm} where $j = (1, \ldots, p)$ and $m = (1, \ldots, M)$. The 1 specifies that the weight occurs in the first layer of adaptive weights, the j subscript specifies the neuron from which the weight connect and the subscript m denotes the unit to which the weight connects. The weights in the second layer are denoted by w_{2mk} where $m = (1, \ldots, M)$ and $k = (1, \ldots, K)$, and this follows the same methodology as those in the first layer. The i-th observation is denoted by $(\underline{x}_i, \underline{y}_i)$, where $\underline{x}_i = (x_{i1}, \ldots, x_{ip})$ are the independent variables and $\underline{y}_i = (y_{i1}, \ldots, y_{iK})$ denote the dependent variables for this particular observation. When working with a two-layer network, we have that the weighted sum of the inputs form the net input into the hidden unit which are denoted by z_{in-m}. For observation i, this is written as

$$z_{in-im} = \sum_{j=0}^{M} w_{1jm} x_{ij}$$

Remember that $x_0 = 1$ denotes the bias term. The net-input is passed through the activation function, denoted by $f_m(\cdot)$, for this specific node and produces output

$$z_{im} = f_m \left(\sum_{j=0}^{M} w_{1jm} x_{ij} \right)$$

Following the same principle as above we have that the output $\hat{y}_i k$, when the neural network is presented with input $\underline{x}_i$ is given by

$$\hat{y}_{ik} = g\left(\sum_{m=0}^{M} w_{2mk} f\left(\sum_{j=1}^{p} w_{1jm} x_{ij}\right)\right)$$

To keep the notation simple, I will generally not specify the number of the observation that is presented to the neural network, i.e. subscript i will be dropped from the notation. This means that the output $\hat{y}_k$, when the neural network is presented with an input $\underline{x}$ from the training sample is given by

$$\hat{y}_k = g_k\left(\sum_{m=0}^{M} w_{2mk} f\left(\sum_{j=0}^{p} w_{1jm} x_j\right)\right)$$

The aim is to keep the notation as simple as possible. If we have a look at the notation, we will see that the subscript i is always used to refer to the observation number, subscript j is always used to refer to an input unit or variable, subscript m is used to denote a hidden unit and subscript k is used to refer to an output unit. The weights always follow the convention in which the first subscript denotes the number of the layer of adaptive weights in which the weight occurs, the second subscript denotes the input unit from which the weight connects and the last subscript denotes the output unit to which the weight connects. When there is only one output unit, which is often the case when the neural network is to be used for regression, then the subscript k is dropped from all the notation in the network. This means that the network output is then denoted by $\hat{y}_1 = \hat{y}$ and the weights which connect the hidden-layer to the output-layer are denoted by w_{2m}. Similarly if we have a neural network with only one layer, we can drop the appropriate subscripts to simplify the notation. For example, we can drop the number which indicates the number of the layer in which the weights occur. This means that the weight, which connects input node j with output node k will be denoted by w_{jk}. It can be seen that the notation is chosen in such a way that each subscript has a specific meaning and hence

can easily be dropped when it is not needed without any confusion to what the remaining subscripts mean.

A.3 R programs used for simulations

A.3.1 R program to illustrate overfitting in regression case

```
### Neural network for regression simulations - Used to illustrate ###
### underfitting and overfitting                                    ###

# Simulate the x and y values
x <- seq(0,15,by=1)
y <- 1 + +sin(0.5*x) + rnorm(length(x),mean=0,sd=0.2)

train.epochs <- 1000000 # Maximum number of epochs to be used for training

# Plot the true underlying distribution, together with the simulated data
x.underlying <- seq(0,15,by=0.1)
y.underlying <- 1 + sin(0.5*x.underlying)

plot.new()
title(main="Simulated data",col.main="black",font.main =4)
axis(1,labels=F,tick=F)
axis(2,tick=F,labels=F)
par(new=T)
plot(x,y)
lines(x.underlying,y.underlying,col='blue')

# Put the x and y values together in a matrix with each row representing
# one observation
```

```
xdat <- cbind(x,y)
colnames(xdat) <- c('x','y')

# Calculate the mean and standard deviation of the x-values
mean.x <- mean(xdat[,1])
sd.x <- sd(xdat[,1])

# Standardize the x-values
prepared.data <- cbind((xdat[,1] - mean.x)/sd.x,y)

# Obtain the x and y values that will be used for training
modelling.x <- prepared.data[,1]
modelling.y <- prepared.data[,2]

n.epoch = 0 # Denotes the number of the current epoch
n.obs = length(modelling.x) # Number of training observations
n.input = ncol(modelling.x) # Number of neural network inputs
n.output = ncol(modelling.y) # Number of outputs from neural network
n.hidden = 3 # Number of hidden nodes
alpha = 0.25 # Learning rate parameter
wtrange = 0.1 # The range around zero to which the initial weights are
              # randomized to
momentum = 0.3 # Momentum parameter

cat("Starting training of neural network with the following parameters:","\n")
cat("Number of training observations:",n.obs,"\n")
cat("Number of inputs:",n.input, "\n")
cat("Number of hidden nodes:", n.hidden, "\n")
```

```
cat("Number of outputs:", n.output, "\n")
cat("Learning rate:", alpha, "\n")
cat("Momentum term:", momentum, "\n")

# Define matrix in which the result of each epoch will be stored
result.mat <- NULL
# Matrix that will be used to give indication of how long training
# will take
time.mat <- NULL

# Initialize the weight matrices in the neural network
bias.ih <- matrix(runif(1*n.hidden,min=-wtrange,max=wtrange),
                  nrow=1,ncol=n.hidden)
wt.ih   <- matrix(runif(n.input*n.hidden,min=-wtrange,max=wtrange),
                  nrow=n.input,ncol=n.hidden)
bias.ho <- matrix(runif(1*n.output,min=-wtrange,max=wtrange),
                  nrow=1,ncol=n.output)
wt.ho   <- matrix(runif(n.hidden*n.output,min=-wtrange,max=wtrange),
                  nrow=n.hidden,ncol=n.output)

# Initialize the delta weights to be used for momentum in the NN training
delta.wt.ih.prev    <- matrix(0,nrow=n.input,ncol=n.hidden)
delta.bias.ih.prev  <- matrix(0,nrow=1,ncol=n.hidden);
delta.wt.ho.prev    <- matrix(0,nrow=n.hidden,ncol=n.output)
delta.bias.ho.prev  <- matrix(0,nrow=1,ncol=n.output)

# This matrix will be used in training. This will be used such that
# the training data is presented in a random order at each epoch
```

```
sequence  <- matrix(0,nrow=n.obs,ncol=1)
for (i in 1:n.obs){
  sequence[i] <- i
}

cat("Initial weights and biases","\n")
cat("Bias Input-->Hidden:",bias.ih,"\n")
cat("Weights Input-->Hidden:",wt.ih,"\n")
cat("Bias Hidden-->Output:",bias.ho,"\n")
cat("Weights Hidden-->Output:",wt.ho,"\n")

# Start the training process
for (n.epoch in 1:train.epochs){

# This function is to give an indication of the time remaining for
# training the neural network
A <- Sys.time()
temp <- n.epoch/1000
if ((temp - floor(temp)) == 0 ){
  cat("Performing epoch number",n.epoch,"out of",train.epochs,"\n")
  rem.time <- (train.epochs-n.epoch)*mean(time.mat)
  cat(floor(rem.time/60),"minutes",
      round(rem.time-floor(rem.time/60)*60),"seconds remaining","\n")
}

# Little function that will ensure the observations are presented
# to the neural network in a random order at each epoch
for (i in n.obs:2){
```

```
  a <- floor(runif(1)*i + 1)
  b <- sequence[i]
  sequence[i] <- sequence[a]
  sequence[a] <- b
}

# Cycle through the whole training set
for (i in 1:n.obs){
  obs.nr <- sequence[i] # Get observation number to present to NN
  input <- modelling.x[obs.nr,] # Get current input
  output <- modelling.y[obs.nr,] # Get current output

  # Calculate the net input to the hidden nodes
  z.in  <- bias.ih + input%*%wt.ih

  # Pass this net input through the hidden node activation function
  # to produce the outputs from the hidden nodes.
  # The logistic function is used as activation function here.
  z     <- (1+exp(-z.in))**(-1)

  # Calculate the net input to the output node
  y.in  <- bias.ho + z%*%wt.ho

  # Pass this net input through the output activation function
  # to obtain a predicted value from the neural network. The linear
  # activation function is used in this case.
  y.hat <- y.in
```

```
# Compare the predicted value from the neural network to the
# output value and calculate the error made
error <- output - y.hat

# Calcuate the deltas for the output nodes
delta.2 <- error

# Use these deltas to calculate the weight adjustments for the
# weights connecting the hidden and the output units
delta.wt.ho   <- alpha*t(z)%*%delta.2 + momentum*delta.wt.ho.prev
delta.bias.ho <- alpha*delta.2 + momentum*delta.bias.ho.prev

# Calculate the deltas for the hidden nodes
delta.1 <- (z*(1-z))*t(wt.ho%*%t(delta.2))

# Use these deltas to calculate the weight adjustments for the
# weights connecting the input and the hidden units
delta.wt.ih <- alpha*t(input)%*%delta.1 + momentum*delta.wt.ih.prev
delta.bias.ih <- alpha*delta.1 + momentum*delta.bias.ih.prev

# Use the weight adjustments to update the weights
bias.ih <- bias.ih + delta.bias.ih
wt.ih   <- wt.ih + delta.wt.ih
bias.ho <- bias.ho + delta.bias.ho
wt.ho   <- wt.ho + delta.wt.ho

# Store the current weight updates for use in the momentum term
# of the next epoch
```

```
  delta.bias.ih.prev  <- delta.bias.ih
  delta.wt.ih.prev    <- delta.wt.ih
  delta.bias.ho.prev  <- delta.bias.ho
  delta.wt.ho.prev    <- delta.wt.ho
}



testing.results <- NULL
testing.results <- matrix(0,nrow=n.obs,ncol=3)

# Test the model on all the observations in the training set
for (i in 1:n.obs){
  # Present each observation to the neural network
  input <- modelling.x[i,]
  output <- modelling.y[i,]

  # Calculate the predicted value from the neural network for each
  # observation
  z.in  <- bias.ih + input%*%wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- bias.ho + z%*%wt.ho
  y.hat <- y.in
  error <- output - y.hat

  # Save each observation and the error made on that observation
  testing.results[i,] = c(input,output,error)
}
```

```
# Calculate the sum of squared errors
total.sse <- sum(testing.results[,3]^2)

# Save the epoch number and the SSE
result.mat <- rbind(result.mat,c(n.epoch,total.sse))

# This code will save the network obtained from the epoch with
# the lowest SSE.
if (nrow(result.mat) > 1){
  if (result.mat[n.epoch,2] < min(result.mat[-n.epoch,2])){
    best.iteration <- n.epoch
    best.error <- total.sse
    best.bias.ih <- bias.ih
    best.wt.ih <- wt.ih
    best.bias.ho <- bias.ho
    best.wt.ho <- wt.ho
    }
}

B <- Sys.time()
time.iteration <- B-A
time.mat <- c(time.mat,time.iteration)
}

# Print a summary of the training results and the best neural network
cat("Total number of epochs:",n.epoch,"\n")
cat("Epoch with lowest error:", best.iteration,"\n")
cat("Error for best epoch:",best.error,"\n")
```

```
cat("Input-->Hidden Bias",best.bias.ih,"\n")
cat("Input-->Hidden Weight Matrix",best.wt.ih,"\n")
cat("Hidden-->Output Bias",best.bias.ho,"\n")
cat("Hidden-->Output Weight Matrix",best.wt.ho,"\n")

name1 = c("Epoch","SSE")
colnames(result.mat) <- name1

# Make a grid of values to display the fitted neural network

# Get a range of x-values, that can be used to predict a y-value
# from the trained neural network
min.x <- min(modelling.x)-0.1
max.x <- max(modelling.x)+0.1
grid.x <- NULL
grid.x <- seq(min.x,max.x,by=0.01)
grid.data <- matrix(0,nrow=length(grid.x),2)

# Present each of these x-values to the neural network and obtain
# a predicted y-value
for (i in 1:length(grid.x)){
  input <- grid.x[i]

  z.in  <- best.bias.ih + input%*%best.wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- best.bias.ho + z%*%best.wt.ho
  y.hat <- y.in
```

```
  grid.data[i,] <- cbind(input,y.hat)
}

# Unstandardize the inputs to display on a graph
grid.data[,1] <- grid.data[,1]*sd.x + mean.x

# Plot the simulated data and the fitted neural network on one graph
plot.new()
axis(1,labels=F,tick=F)
axis(2,labels=F,tick=F)
par(new=T)
plot(xdat[,1],xdat[,2],xlim=c(0,15),ylim=c(-0.25,2.25),xlab="x",ylab="y")
par(new=T)
plot(grid.data[,1],grid.data[,2],type='l',
      xlim=c(0,15),ylim=c(-0.25,2.25),xlab="x",ylab="y",lwd=2)
lines(x.underlying,y.underlying,col='blue',lty=3,lwd=0.8)
```

R program to illustrate use of modelling and testing set in regression neural network

```
### Neural network for regression simulations - Used to illustrate ###
### the effect of splitting the data into a training and testing   ###
### set and doing the training and testing concurrently            ###

# Simulate the x and y values
x <- runif(150,0,15)
y <- 1 + +sin(0.5*x) + rnorm(length(x),mean=0,sd=0.2)
```

```
train.epochs <- 1000000 # Maximum number of epochs to be used for training

# Plot the true underlying distribution together with the simulated data points
x.underlying <- seq(0,15,by=0.1)
y.underlying <- 1 + sin(0.5*x.underlying)
plot.new()
title(main="Simulated data",col.main="black",font.main =4)
axis(1,labels=F,tick=F)
axis(2,tick=F,labels=F)
par(new=T)
plot(x,y)
lines(x.underlying,y.underlying,col='blue')

# Put the x and y values together in a matrix with each row representing one
# observation
xdat <- cbind(x,y)
colnames(xdat) <- c('x','y')

# Calculate the mean and the standard deviation of the inputs (x)
mean.x <- mean(xdat[,1])
sd.x <- sd(xdat[,1])

# Standardize the x-values
prepared.data <- cbind((xdat[,1]-mean.x)/sd.x,y)

# Get a training and testing set
sample <- sample(1:nrow(prepared.data))
```

```
per.testing <- 0.2 # Specify the percentage of values in the testing set
modelling <- sample[1:((1-per.testing)*length(sample))]
testing <- sample[-(1:((1-per.testing)*length(sample)))]

# Get the independent and dependent variable for the training data
modelling.x <- matrix(prepared.data[modelling,1],ncol=1)
modelling.y <- matrix(prepared.data[modelling,2],ncol=1)

# Get the independent and dependent variable for the testing data
testing.x <- matrix(prepared.data[testing,1],ncol=1)
testing.y <- matrix(prepared.data[testing,2],ncol=1)

n.epoch = 0 # Denotes the number of the current epoch
n.obs = length(modelling.x) # Number of training observations
n.input = ncol(modelling.x) # Number of inputs into NN
n.output = ncol(modelling.y) # Number of outputs from NN
n.hidden = 20 # Number of hidden nodes
alpha = 0.2 # Learning rate parameter
wtrange = 0.1 # The range around zero to which the initial weights are
              # randomized to
momentum = 0.5 # Momentum rate parameter

# Print a few initial statistics about the neural network to be trained
cat("Starting training of neural network with the following parameters:","\n")
cat("Number of training observations:",n.obs,"\n")
cat("Number of inputs:",n.input, "\n")
cat("Number of hidden nodes:", n.hidden, "\n")
cat("Number of outputs:", n.output, "\n")
```

```
cat("Learning rate:", alpha, "\n")
cat("Momentum term:", momentum, "\n")

result.mat <- NULL
time.mat <- NULL

# Intialize the weight matrices to be used in the neural network
bias.ih   <- matrix(runif(1*n.hidden,min=-wtrange,max=wtrange),
                    nrow=1,ncol=n.hidden)
wt.ih     <- matrix(runif(n.input*n.hidden,min=-wtrange,max=wtrange),
                    nrow=n.input,ncol=n.hidden)
bias.ho   <- matrix(runif(1*n.output,min=-wtrange,max=wtrange),
                    nrow=1,ncol=n.output)
wt.ho     <- matrix(runif(n.hidden*n.output,min=-wtrange,max=wtrange),
                    nrow=n.hidden,ncol=n.output)

# Initialize the delta weights to be used for momentum in the NN training
delta.wt.ih.prev    <- matrix(0,nrow=n.input,ncol=n.hidden)
delta.bias.ih.prev  <- matrix(0,nrow=1,ncol=n.hidden);
delta.wt.ho.prev    <- matrix(0,nrow=n.hidden,ncol=n.output)
delta.bias.ho.prev  <- matrix(0,nrow=1,ncol=n.output)

# This matrix will be used in training. This will be used such that the training
# data is presented in a random order at each epoch
sequence  <- matrix(0,nrow=n.obs,ncol=1)
for (i in 1:n.obs){
  sequence[i] <- i
}
```

```
cat("Initial weights and biases","\n")
cat("Bias Input-->Hidden:",bias.ih,"\n")
cat("Weights Input-->Hidden:",wt.ih,"\n")
cat("Bias Hidden-->Output:",bias.ho,"\n")
cat("Weights Hidden-->Output:",wt.ho,"\n")

# Start training of the neural network
for (n.epoch in 1:train.epochs){

# This function is to give an indication of the time remaining for training
# the neural network
A <- Sys.time()
tss <- 0
temp <- n.epoch/1000
if ((temp - floor(temp)) == 0 ){
  cat("Performing epoch number",n.epoch,"out of",train.epochs,"\n")
  rem.time <- (train.epochs-n.epoch)*mean(time.mat)
  cat(floor(rem.time/60),"minutes",
      round(rem.time-floor(rem.time/60)*60),"seconds remaining","\n")
}

# Little function that will ensure that the observations are presented in a
# random order when training the neural network
for (i in n.obs:2){
  a <- floor(runif(1)*i + 1)
  b <- sequence[i]
  sequence[i] <- sequence[a]
```

```
  sequence[a] <- b
}

# Cycle through the whole training dataset
for (i in 1:n.obs){
  obs.nr <- sequence[i] # Get the number of the observation that will next
                        # be used for training
  # Get the input for the chosen observation/data point
  input <- modelling.x[obs.nr,]
  # Get the output for that observation
  output <- modelling.y[obs.nr,]

  # Calculate the net input into the hidden nodes from this input,incl bias
  z.in  <- bias.ih + input%*%wt.ih

  # Pass the net input through the activation function to get the output from
  # the hidden nodes. The logistic function is used as activation function
  z     <- (1+exp(-z.in))**(-1)

  # Use the outputs from the hidden nodes to calculate the net input into the
  # output nodes
  y.in  <- bias.ho + z%*%wt.ho

  # Calculate the outputs from the neural network by passing the net input to
  # the outputs through the output activation function. The output activation
  # function used here is a linear activation function.
  y.hat <- y.in
```

```
# Calculate the error made on each of the outputs by comparing the
# predicted value from the neural network to the desired/actual output
error <- output - y.hat

# Calculate the deltas for the output units
delta.2 <- error

# Use these deltas to calculate the weight adjustments for the weights
# which connect the hidden units and the output units
delta.wt.ho   <- alpha*t(z)%*%delta.2 + momentum*delta.wt.ho.prev
delta.bias.ho <- alpha*delta.2 + momentum*delta.bias.ho.prev

# Calculate the deltas for the hidden nodes
delta.1 <- (z*(1-z))*t(wt.ho%*%t(delta.2))

# Use these deltas to calculate the weight adjustments for the weights
# which connect the input units and the hidden units
delta.wt.ih <- alpha*t(input)%*%delta.1 + momentum*delta.wt.ih.prev
delta.bias.ih <- alpha*delta.1 + momentum*delta.bias.ih.prev

# Use the calculated weight adjustments to update the weights
bias.ih <- bias.ih  + delta.bias.ih
wt.ih   <- wt.ih    + delta.wt.ih
bias.ho <- bias.ho  + delta.bias.ho
wt.ho   <- wt.ho    + delta.wt.ho

# Store the current weight updates for use in the momentum term of the
# next iteration
```

```
  delta.bias.ih.prev  <- delta.bias.ih
  delta.wt.ih.prev    <- delta.wt.ih
  delta.bias.ho.prev  <- delta.bias.ho
  delta.wt.ho.prev    <- delta.wt.ho
}

testing.results <- NULL
testing.results <- matrix(0,nrow=nrow(testing.x),ncol=3)

# Will now use the testing set to test the prediction error of the current NN
# Cycle through all the testing observations
for (i in 1:(nrow(testing.x))){
  # Present each testing observation to the network one at a time
  input <- testing.x[i,]
  output <- testing.y[i,]

  # Calculate the predicted value from the neural network
  z.in  <- bias.ih + input%*%wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- bias.ho + z%*%wt.ho
  y.hat <- y.in
  error <- output - y.hat

  # Save each observation and the error made on that observation
  testing.results[i,] = c(input,output,error)
}

# Calculate the sum of squared errors for all observations in the testing set
```

```
testing.sse <- sum(testing.results[,3]^2)

# Save each epoch number and the SSE for that epoch
result.mat <- rbind(result.mat,c(n.epoch,testing.sse))

# Here we will compare the SSE calculated on the testing set for this epoch
# against all those obtained from the previous epochs. We will then keep the
# best network, where the best is measured on the testing set
if (nrow(result.mat) > 1){
  if (result.mat[n.epoch,2] < min(result.mat[-n.epoch,2])){
    best.iteration <- n.epoch
    best.error <- testing.sse
    best.bias.ih <- bias.ih
    best.wt.ih <- wt.ih
    best.bias.ho <- bias.ho
    best.wt.ho <- wt.ho
    }
}

B <- Sys.time()
time.iteration <- B-A
time.mat <- c(time.mat,time.iteration)
}

# Print a summary of the training results and the best neural network
cat("Total number of epochs:",n.epoch,"\n")
cat("Epoch with lowest error:", best.iteration,"\n")
cat("Error for best epoch:",best.error,"\n")
```

```
cat("Input-->Hidden Bias",best.bias.ih,"\n")
cat("Input-->Hidden Weight Matrix",best.wt.ih,"\n")
cat("Hidden-->Output Bias",best.bias.ho,"\n")
cat("Hidden-->Output Weight Matrix",best.wt.ho,"\n")

name1 = c("Epoch","SSE")
colnames(result.mat) <- name1

# Make a grid of values to display the fitted neural network

# Get a range of x-values, that can be used to predict a y-value
# from the trained neural network
min.x <- min(modelling.x)-0.1
max.x <- max(modelling.x)+0.1
grid.x <- NULL
grid.x <- seq(min.x,max.x,by=0.01)
grid.data <- matrix(0,nrow=length(grid.x),2)

for (i in 1:length(grid.x)){
  # Present each of these values from this grid to the neural network and
  # get a predicted value y_hat
  input <- grid.x[i]
  z.in  <- best.bias.ih + input%*%best.wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- best.bias.ho + z%*%best.wt.ho
  y.hat <- y.in
  # Save the input and the predicted value
  grid.data[i,] <- cbind(input,y.hat)
```

```
}

# Unstandardize the inputs
grid.data[,1] <- grid.data[,1]*sd.x + mean.x

# Plot the simulated data points and the fitted neural network on one graph
plot.new()
title(main='Neural network simulation indicating the effect of a testing set'
      , col = 'black', font.main = 4)
axis(1,labels=F,tick=F)
axis(2,labels=F,tick=F)
par(new=T)
plot(xdat[modelling,1],xdat[modelling,2],
      xlim=c(0,15),ylim=c(-0.25,2.25),xlab="x",ylab="y",pch=20)
par(new=T)
plot(xdat[testing,1],xdat[testing,2],
      xlim=c(0,15),ylim=c(-0.25,2.25),xlab="x",ylab="y",pch=4)
par(new=T)
plot(grid.data[,1],grid.data[,2],type='l',
      xlim=c(0,15),ylim=c(-0.25,2.25),xlab="x",ylab="y",lwd=2)
lines(x.underlying,y.underlying,col='blue',lty=3,lwd=0.8)
```

R program to illustrate overfitting in classification case

```
### Neural network for classification simulations - Used to illustrate ###
### underfitting and overfitting ###
```

```
library(MASS)
train.epochs <- 100000 # Maximum number of epochs used for training the network
ng <- 100 # Number of observations in each of the different groups

### Simulate the the data from the different groups
X.1 <- mvrnorm(n=ng,mu= c(40,50),Sigma= matrix(c(100,0,0,100),2,2))
X.2 <- mvrnorm(n=ng,mu= c(60,90),Sigma= matrix(c(100,0,0,100),2,2))
X.3 <- mvrnorm(n=ng,mu= c(100,80),Sigma= matrix(c(100,0,0,100),2,2))
X.4 <- mvrnorm(n=ng,mu= c(80,40),Sigma= matrix(c(100,0,0,100),2,2))
X.5 <- mvrnorm(n=ng,mu= c(70,65),Sigma= matrix(c(50,0,0,50),2,2))

# Put all the data from the different groups together in one matrix and give a
# y-value to each group (in this case I just numbered each group to indicate
# which group each observation belongs to)
G.1 <- cbind(X.1,1)
G.2 <- cbind(X.2,2)
G.3 <- cbind(X.3,3)
G.4 <- cbind(X.4,4)
G.5 <- cbind(X.5,5)
name1 <- c("x1","x2","y")
xdat <- rbind(G.1,G.2,G.3,G.4,G.5)
colnames(xdat) <- name1

# Calculate the mean and the standard deviation of the inputs (x1 and x2)
mean.x <- colMeans(xdat[,1:2])
sd.x <- c(sd(xdat[,1]),sd(xdat[,2]))

# Standardize the inputs and use 1-of-K coding scheme for the dependent variable
```

```
prepared.data<-cbind((xdat[,1]-mean.x[1])/sd.x[1],(xdat[,2]-mean.x[2])/sd.x[2],
1*(xdat[,3]==1),1*(xdat[,3]==2),1*(xdat[,3]==3),1*(xdat[,3]==4),1*(xdat[,3]==5))

# Obtain the independent variables (x1,x2) and the dependent variables (y1-y5)
# that will be used for training
modelling.x <- prepared.data[,1:2]
modelling.y <- prepared.data[,3:7]

n.epoch = 0 # Denotes the number of the current epoch
n.obs = nrow(modelling.x) # Number of training dataset observations
n.input = ncol(modelling.x) # Number of inputs (independent variables)
n.output = ncol(modelling.y) # Number of outputs (5 in this case)
n.hidden = 3 # Number of hidden nodes
alpha = 0.2 # Learning rate parameter
wtrange = 0.1 # The range around zero to which the initial weights are
              # randomized to
momentum = 0.4 # Momentum parameter

# Print some initial statistics on the neural network to be fitted
cat("Starting training of neural network with the following parameters:","\n")
cat("Number of training observations:",n.obs,"\n")
cat("Number of inputs:",n.input, "\n")
cat("Number of hidden nodes:", n.hidden, "\n")
cat("Number of outputs:", n.output, "\n")
cat("Learning rate:", alpha, "\n")
cat("Momentum term:", momentum, "\n")

result.mat <- NULL
```

```
time.mat <- NULL

# Initialize the weight matrices in the neural network
bias.ih   <- matrix(runif(1*n.hidden,min=-wtrange,max=wtrange),
                    nrow=1,ncol=n.hidden)
wt.ih     <- matrix(runif(n.input*n.hidden,min=-wtrange,max=wtrange),
                    nrow=n.input,ncol=n.hidden)
bias.ho   <- matrix(runif(1*n.output,min=-wtrange,max=wtrange),
                    nrow=1,ncol=n.output)
wt.ho     <- matrix(runif(n.hidden*n.output,min=-wtrange,max=wtrange),
                    nrow=n.hidden,ncol=n.output)

# Initialize the delta weights to be used for momentum in the NN training
delta.wt.ih.prev    <- matrix(0,nrow=n.input,ncol=n.hidden)
delta.bias.ih.prev  <- matrix(0,nrow=1,ncol=n.hidden);
delta.wt.ho.prev    <- matrix(0,nrow=n.hidden,ncol=n.output)
delta.bias.ho.prev  <- matrix(0,nrow=1,ncol=n.output)

# This matrix will be used in training. This will be used such that the training
# data is presented in a random order at each epoch
sequence  <- matrix(0,nrow=n.obs,ncol=1)
for (i in 1:n.obs){
  sequence[i] <- i
}

cat("Initial weights and biases","\n")
cat("Bias Input-->Hidden:",bias.ih,"\n")
cat("Weights Input-->Hidden:",wt.ih,"\n")
```

```
cat("Bias Hidden-->Output:",bias.ho,"\n")
cat("Weights Hidden-->Output:",wt.ho,"\n")

# Start the training process
for (n.epoch in 1:train.epochs){

# This function is to give an indication of the time remaining for training
# the neural network
A <- Sys.time()
temp <- n.epoch/1000
if ((temp - floor(temp)) == 0 ){
  cat("Performing epoch number",n.epoch,"out of",train.epochs,"\n")
  rem.time <- (train.epochs-n.epoch)*mean(time.mat)
  cat(floor(rem.time/60),"minutes",
      round(rem.time-floor(rem.time/60)*60),"seconds remaining","\n")
}

# Little function that will ensure that the observations are presented in a
# random order when training the neural network
for (i in n.obs:2){
  a <- floor(runif(1)*i + 1)
  b <- sequence[i]
  sequence[i] <- sequence[a]
  sequence[a] <- b
}

# Cycle through the whole training dataset
  for (i in 1:n.obs){
```

```
obs.nr <- sequence[i] # Get the number of the observation that will next
                      # be used for training

# Get the inputs of the chosen observation/datapoint
input <- matrix(modelling.x[obs.nr,],nrow=1,ncol=2)
# Get the outputs
output <- matrix(modelling.y[obs.nr,],nrow=1)

# Calculate the net input into the hidden nodes from this input,incl bias
z.in <- bias.ih + input%*%wt.ih

# Pass the net input through the activation function to get the output from
# the hidden nodes. The logistic function is used as activation function
z <- (1+exp(-z.in))**(-1)

# Use the outputs from the hidden nodes to calculate the net input into the
# output nodes
y.in  <- bias.ho + z%*%wt.ho

# Calculate the outputs from the neural network by passing the net input to
# the outputs through the output activation function. The output activation
# function used here is the softmax function.
y.hat <- exp(y.in)/sum(exp(y.in))

# Calculate the error made on each of the outputs by comparing the
# predicted value from the neural network to the desired/actual output
error <- output - y.hat
```

```
# Calculate the deltas for the output units
delta.2 <- error

# Use these deltas to calculate the weight adjustments for the weights
# which connect the hidden units and the output units
delta.wt.ho   <- alpha*t(z)%*%delta.2 + momentum*delta.wt.ho.prev
delta.bias.ho <- alpha*delta.2 + momentum*delta.bias.ho.prev

# Calculate the deltas for the hidden nodes
delta.1 <- (z*(1-z))*t(wt.ho%*%t(delta.2))

# Use these deltas to calculate the weight adjustments for the weights
# which connect the input units and the hidden units
delta.wt.ih <- alpha*t(input)%*%delta.1 + momentum*delta.wt.ih.prev
delta.bias.ih <- alpha*delta.1 + momentum*delta.bias.ih.prev

# Use the calculated weight adjustments to update the weights
bias.ih <- bias.ih + delta.bias.ih
wt.ih   <- wt.ih + delta.wt.ih
bias.ho <- bias.ho + delta.bias.ho
wt.ho   <- wt.ho + delta.wt.ho

# Store the current weight updates for use in the momentum term of the
# next iteration
delta.bias.ih.prev  <- delta.bias.ih
delta.wt.ih.prev    <- delta.wt.ih
delta.bias.ho.prev  <- delta.bias.ho
delta.wt.ho.prev    <- delta.wt.ho
```

```
  }

testing.results <- NULL
testing.results <- matrix(0,nrow=n.obs,ncol=4)

# Test the model on all the observations in the training set
for (i in 1:n.obs){
  # Present each observation to the neural network
  input <- matrix(modelling.x[i,],nrow=1)
  output <- matrix(modelling.y[i,],nrow=1)

  # Calculate the predicted value from the neural network for each observation
  z.in  <- bias.ih + input%*%wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- bias.ho + z%*%wt.ho
  y.hat <- exp(y.in)/sum(exp(y.in))

  # Classify the actual outputs to one of the five possible categories
  category.y <- which.max(output)
  # Classify the output from the neural network to one of the five possible
  # categories
  category.y.hat <- which.max(y.hat)
  # Save the input, the correct output category and also the predicted category
  testing.results[i,] = c(input,category.y,category.y.hat)
}
name2 <- c("x1","x2","y","y hat")
colnames(testing.results) <- name2
```

```
# Calculate the missclassification rate for this network
miss.class <- 1*(testing.results[,3]!=testing.results[,4])
mcr <- mean(miss.class)

# Save the epoch number and the missclassification rate
result.mat <- rbind(result.mat,c(n.epoch,mcr))

# Keep the best network i.e. the one with the lowest mcr as calculated on the
# training dataset
if (nrow(result.mat) > 1){
  if (result.mat[n.epoch,2] < min(result.mat[-n.epoch,2])){
    best.iteration <- n.epoch
    best.error <- mcr
    best.bias.ih <- bias.ih
    best.wt.ih <- wt.ih
    best.bias.ho <- bias.ho
    best.wt.ho <- wt.ho
    }
}

B <- Sys.time()
time.iteration <- B-A
time.mat <- c(time.mat,time.iteration)
}

# Print a summary of the training results and the best neural network obtained
cat("Total number of epochs:",n.epoch,"\n")
cat("Epoch with lowest missclassification rate:", best.iteration,"\n")
```

```
cat("Missclassification rate for best epoch:",best.error,"\n")
cat("Input-->Hidden Bias",best.bias.ih,"\n")
cat("Input-->Hidden Weight Matrix",best.wt.ih,"\n")
cat("Hidden-->Output Bias",best.bias.ho,"\n")
cat("Hidden-->Output Weight Matrix",best.wt.ho,"\n")

name1 = c("Epoch","MCR")
colnames(result.mat) <- name1

# Create a grid to display the classification areas fitted by the neural network

# Again, get a range of x-values, which can be presented to the neural network
# to obtain a predicted value, and hence classify that specific point to a group
min.x1 <- min(modelling.x[,1])-0.1
max.x1 <- max(modelling.x[,1])+0.1
min.x2 <- min(modelling.x[,2])-0.1
max.x2 <- max(modelling.x[,2])+0.1
step.x1 <- seq(min.x1,max.x1,by=0.01)
step.x2 <- seq(min.x2,max.x2,by=0.01)
grid.x <- NULL
for (a in step.x1){
  for (b in step.x2){
    grid.x <- rbind(grid.x,c(a,b))
  }
}
grid.data <- matrix(0,nrow=nrow(grid.x),3)

# Present each of the x-values to the neural network and classify the input
```

```
# to one of the five groups
for (i in 1:nrow(grid.x)){
  input <- grid.x[i,]
  z.in  <- best.bias.ih + input%*%best.wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- best.bias.ho + z%*%best.wt.ho
  y.hat <- exp(y.in)/sum(exp(y.in))
  y.hat.class <- which.max(y.hat)
  grid.data[i,] <- c(input,y.hat.class)
}

# Unstandardize the inputs to display it on a graph
grid.data[,1] <- grid.data[,1]*sd.x[1]+mean.x[1]
grid.data[,2] <- grid.data[,2]*sd.x[2]+mean.x[2]



# Plot the grid of values and the predicted values from the best neural network
# together with simulated data on one graph
plot.new()
title(main="Effect of overfitting",
      sub="Network with 3 hidden nodes trained for 100000 epochs"
      ,col.main="black",font.main =4)
axis(1,labels=F,tick=F)
axis(2,labels=F,tick=F)
par(new=T)
plot(grid.data[,1],grid.data[,2],col=grid.data[,3],
    pch=20,cex=2,xlim=c(20,110),ylim=c(20,110),xlab="x1",ylab="x2")
par(new=T)
```

```
plot(xdat[,1],xdat[,2],col=xdat[,3]+10,
    pch=16,cex=1,xlim=c(20,110),ylim=c(20,110),xlab="x1",ylab="x2")
```

R program to illustrate use of modelling and testing set in classification neural network

```
### Neural network for classification simulations - Used to illustrate    ###
### the effect of incorporating a testing set into the learning algorithm ###

library(MASS)

train.epochs <- 10000 # Maximum number of epochs used for training the network
ng <- 100  # Number of observations in each of the different groups

### Simulate the the data from the different groups using bivariate normal
X.1 <- mvrnorm(n=ng,mu= c(40,50),Sigma= matrix(c(100,0,0,100),2,2))
X.2 <- mvrnorm(n=ng,mu= c(60,90),Sigma= matrix(c(100,0,0,100),2,2))
X.3 <- mvrnorm(n=ng,mu= c(100,80),Sigma= matrix(c(100,0,0,100),2,2))
X.4 <- mvrnorm(n=ng,mu= c(80,40),Sigma= matrix(c(100,0,0,100),2,2))
X.5 <- mvrnorm(n=ng,mu= c(70,65),Sigma= matrix(c(50,0,0,50),2,2))

# Put all the data from the different groups together in one matrix and give a
# y-value to each group (in this case I just numbered each group to indicate
# which group each observation belongs to)
G.1 <- cbind(X.1,1)
G.2 <- cbind(X.2,2)
```

```
G.3 <- cbind(X.3,3)
G.4 <- cbind(X.4,4)
G.5 <- cbind(X.5,5)
name1 <- c("x1","x2","y")
xdat <- rbind(G.1,G.2,G.3,G.4,G.5)
colnames(xdat) <- name1

# Calculate the mean and the standard deviation of the inputs (x1 and x2)
mean.x <- colMeans(xdat[,1:2])
sd.x <- c(sd(xdat[,1]),sd(xdat[,2]))

# Standardize the inputs and use 1-of-K coding scheme for the dependent variable
prepared.data <-cbind((xdat[,1]-mean.x[1])/sd.x[1],(xdat[,2]-mean.x[2])/sd.x[2],
1*(xdat[,3]==1),1*(xdat[,3]==2),1*(xdat[,3]==3),1*(xdat[,3]==4),1*(xdat[,3]==5))

# Split the data into a modelling and testing dataset
sample <- sample(1:nrow(prepared.data))
per.testing <- 0.2 # Percentage of observations assigned to testing set
modelling <- sample[1:((1-per.testing)*length(sample))]
testing <- sample[-(1:((1-per.testing)*length(sample)))]

# Assign the modelling and testing inputs and outputs of the neural network
modelling.x <- prepared.data[modelling,1:2]
modelling.y <- prepared.data[modelling,3:7]
testing.x <- prepared.data[testing,1:2]
testing.y <- prepared.data[testing,3:7]

n.epoch = 0 # Denotes the number of the current epoch
```

```
n.obs = nrow(modelling.x) # Number of training dataset observations
n.input = ncol(modelling.x) # Number of inputs (independent variables)
n.output = ncol(modelling.y) # Number of outputs (5 in this case)
n.hidden = 1 # Number of hidden nodes
alpha = 0.2 # Learning rate parameter
wtrange = 0.1 # The range around zero to which the initial weights are
              # randomized to
momentum = 0.4 # Momentum parameter

# Print some initial statistics on the neural network to be fitted
cat("Starting training of neural network with the following parameters:","\n")
cat("Number of training observations:",n.obs,"\n")
cat("Number of inputs:",n.input, "\n")
cat("Number of hidden nodes:", n.hidden, "\n")
cat("Number of outputs:", n.output, "\n")
cat("Learning rate:", alpha, "\n")
cat("Momentum term:", momentum, "\n")

result.mat <- NULL
time.mat <- NULL

# Initialize the weight matrices in the neural network
sequence  <- matrix(0,nrow=n.obs,ncol=1)
bias.ih   <- matrix(runif(1*n.hidden,min=-wtrange,max=wtrange),
             nrow=1,ncol=n.hidden)
wt.ih     <- matrix(runif(n.input*n.hidden,min=-wtrange,max=wtrange),
             nrow=n.input,ncol=n.hidden)
bias.ho   <- matrix(runif(1*n.output,min=-wtrange,max=wtrange),
```

```
                nrow=1,ncol=n.output)
wt.ho      <- matrix(runif(n.hidden*n.output,min=-wtrange,max=wtrange),
                nrow=n.hidden,ncol=n.output)

# Initialize the delta weights to be used for momentum in the NN training
delta.wt.ih.prev    <- matrix(0,nrow=n.input,ncol=n.hidden)
delta.bias.ih.prev  <- matrix(0,nrow=1,ncol=n.hidden);
delta.wt.ho.prev    <- matrix(0,nrow=n.hidden,ncol=n.output)
delta.bias.ho.prev  <- matrix(0,nrow=1,ncol=n.output)

# This matrix will be used in training. This will be used such that the training
# data is presented in a random order at each epoch
for (i in 1:n.obs){
  sequence[i] <- i
}

cat("Initial weights and biases","\n")
cat("Bias Input-->Hidden:",bias.ih,"\n")
cat("Weights Input-->Hidden:",wt.ih,"\n")
cat("Bias Hidden-->Output:",bias.ho,"\n")
cat("Weights Hidden-->Output:",wt.ho,"\n")

# Start the training process
for (n.epoch in 1:train.epochs){

# This function is to give an indication of the time remaining for training
# the neural network
A <- Sys.time()
```

```
temp <- n.epoch/1000
if ((temp - floor(temp)) == 0 ){
  cat("Performing epoch number",n.epoch,"out of",train.epochs,"\n")
  rem.time <- (train.epochs-n.epoch)*mean(time.mat)
  cat(floor(rem.time/60),"minutes",round(rem.time-floor(rem.time/60)*60),
        "seconds remaining","\n")
}

# Little function that will ensure that the observations are presented in a
# random order when training the neural network
for (i in n.obs:2){
  a <- floor(runif(1)*i + 1)
  b <- sequence[i]
  sequence[i] <- sequence[a]
  sequence[a] <- b
}

  # Cycle through the whole training dataset
  for (i in 1:n.obs){
    #i <- 1
    obs.nr <- sequence[i]  # Get the number of the next observation that will be
                           # used for training

    # Get the inputs of the chosen observation/datapoint
    input <- matrix(modelling.x[obs.nr,],nrow=1)
    # Get the outputs
    output <- matrix(modelling.y[obs.nr,],nrow=1)
```

```
# Calculate the net input into the hidden nodes from this input,incl bias
z.in <- bias.ih + input%*%wt.ih

# Pass the net input through the activation function to get the output from
# the hidden nodes. The logistic function is used as activation function
z <- (1+exp(-z.in))**(-1)

# Use the outputs from the hidden nodes to calculate the net input into the
# output nodes
y.in <- bias.ho + z%*%wt.ho

# Calculate the outputs from the neural network by passing the net input to
# the outputs through the output activation function. The output activation
# function used here is the softmax function.
y.hat <- exp(y.in)/sum(exp(y.in))

# Calculate the error made on each of the outputs by comparing the
# predicted value from the neural network to the desired/actual output
error <- output - y.hat

# Calculate the deltas for the output units
delta.2 <- error

# Use these deltas to calculate the weight adjustments for the weights
# which connect the hidden units and the output units
delta.wt.ho   <- alpha*t(z)%*%delta.2 + momentum*delta.wt.ho.prev
delta.bias.ho <- alpha*delta.2 + momentum*delta.bias.ho.prev
```

```
    # Calculate the deltas for the hidden nodes
    delta.1 <- (z*(1-z))*t(wt.ho%*%t(delta.2))

    # Use these deltas to calculate the weight adjustments for the weights
    # which connect the input units and the hidden units
    delta.wt.ih <- alpha*t(input)%*%delta.1 + momentum*delta.wt.ih.prev
    delta.bias.ih <- alpha*delta.1 + momentum*delta.bias.ih.prev

    # Use the calculated weight adjustments to update the weights
    bias.ih <- bias.ih + delta.bias.ih
    wt.ih   <- wt.ih + delta.wt.ih
    bias.ho <- bias.ho + delta.bias.ho
    wt.ho   <- wt.ho + delta.wt.ho

    # Store the current weight updates for use in the momentum term of the
    # next iteration
    delta.bias.ih.prev  <- delta.bias.ih
    delta.wt.ih.prev    <- delta.wt.ih
    delta.bias.ho.prev  <- delta.bias.ho
    delta.wt.ho.prev    <- delta.wt.ho
  }

testing.results <- NULL
testing.results <- matrix(0,nrow=nrow(testing.x),ncol=4)

# Will now use the testing set to calculate the misclassification rate of
# the current NN
# Cycle through all the testing observations
```

```
for (i in 1:nrow(testing.x)){
  # Present each testing observation to the network one at a time
  input <- matrix(testing.x[i,],nrow=1)
  output <- matrix(testing.y[i,],nrow=1)

  # Calculate the predicted category from the neural network
  z.in  <- bias.ih + input%*%wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- bias.ho + z%*%wt.ho
  y.hat <- exp(y.in)/sum(exp(y.in))
  error <- output - y.hat
  category.y <- which.max(output)
  category.y.hat <- which.max(y.hat) # Classify to the category with the largest
                                     # posterior probability

  # Save each observation together with the predicted category from the NN
  testing.results[i,] = c(input,category.y,category.y.hat)
}

name2 <- c("x1","x2","y","y hat")
colnames(testing.results) <- name2

# Compare the predicted category against the true category to see which
# observations are missclassified
miss.class <- 1*(testing.results[,3]!=testing.results[,4])
#Calculate the missclassification rate
mcr <- mean(miss.class)
```

```
# Save the epoch number and the missclassification rate
result.mat <- rbind(result.mat,c(n.epoch,mcr))

# Keep the best network i.e. the one with the lowest mcr as calculated on the
# independent testing set
if (nrow(result.mat) > 1){
  if (result.mat[n.epoch,2] < min(result.mat[-n.epoch,2])){
    best.iteration <- n.epoch
    best.error <- mcr
    best.bias.ih <- bias.ih
    best.wt.ih <- wt.ih
    best.bias.ho <- bias.ho
    best.wt.ho <- wt.ho
    }
}
B <- Sys.time()
time.iteration <- B-A
time.mat <- c(time.mat,time.iteration)
}

# Print a summary of the training results and the best neural network obtained
cat("Total number of epochs:",n.epoch,"\n")
cat("Epoch with lowest missclassification rate:", best.iteration,"\n")
cat("Missclassification rate for best epoch:",best.error,"\n")
cat("Input-->Hidden Bias",best.bias.ih,"\n")
cat("Input-->Hidden Weight Matrix",best.wt.ih,"\n")
cat("Hidden-->Output Bias",best.bias.ho,"\n")
cat("Hidden-->Output Weight Matrix",best.wt.ho,"\n")
```

```
name1 = c("Epoch","MCR")
colnames(result.mat) <- name1



# Create a grid to display the classification areas fitted by the neural network

# Get a range of x-values, which can be presented to the neural network
# to obtain a predicted value, and hence classify that specific point to a group
min.x1 <- min(modelling.x[,1])-0.1
max.x1 <- max(modelling.x[,1])+0.1
min.x2 <- min(modelling.x[,2])-0.1
max.x2 <- max(modelling.x[,2])+0.1
step.x1 <- seq(min.x1,max.x1,by=0.01)
step.x2 <- seq(min.x2,max.x2,by=0.01)
grid.x <- NULL
for (a in step.x1){
  for (b in step.x2){
    grid.x <- rbind(grid.x,c(a,b))
  }
}
grid.data <- matrix(0,nrow=nrow(grid.x),3)

# Present each of the x-values in the grid to the neural network and classify
# the input to one of the five groups
for (i in 1:nrow(grid.x)){
  input <- grid.x[i,]
```

```
  z.in  <- best.bias.ih + input%*%best.wt.ih
  z     <- (1+exp(-z.in))**(-1)
  y.in  <- best.bias.ho + z%*%best.wt.ho
  y.hat <- exp(y.in)/sum(exp(y.in))
  y.hat.class <- which.max(y.hat)
  grid.data[i,] <- c(input,y.hat.class)
}

# Unstandardize the inputs to display it on a graph
grid.data[,1] <- grid.data[,1]*sd.x[1]+mean.x[1]
grid.data[,2] <- grid.data[,2]*sd.x[2]+mean.x[2]

# Plot the grid of values and the predicted values from the best neural network
# together with simulated data on one graph
plot.new()
title(main="Effect of underfitting",
        sub="Network with 1 hidden nodes trained for 100000 epochs",
        col.main="black",font.main =4)
axis(1,tick=F,labels=F)
axis(2,tick=F,labels=F)
par(new=T)
plot(grid.data[,1],grid.data[,2],col=grid.data[,3],
      pch=20,cex=2,xlim=c(20,110),ylim=c(20,110),xlab="x1",ylab="x2")
par(new=T)
plot(xdat[modelling,1],xdat[modelling,2],col=xdat[modelling,3]+10,
      pch=20,cex=1,xlim=c(20,110),ylim=c(20,110),xlab="x1",ylab="x2")
par(new=T)
plot(xdat[testing,1],xdat[testing,2],col=xdat[testing,3]+10,
```

```
pch='*',cex=2,xlim=c(20,110),ylim=c(20,110),xlab="x1",ylab="x2")
```

A.4 SAS programs used for Lightstone application

```
/* Neural network for picking the optimal Lightstone prediction available */

/* Assign the necessary libraries */
libname neural "C:\Personal\M_Verandeling\Programs";
libname data_jan "L:\Lightstone\System Testing\System Testing 17 Jan 2009";
libname data_feb "L:\Lightstone\System Testing\System Testing 18 Feb 2009";
libname data_mar "L:\Lightstone\System Testing\System Testing 19 Mar 2009";
libname data_apr "L:\Lightstone\System Testing\System Testing 20 Apr 2009";
libname data_may "L:\Lightstone\System Testing\System Testing 21 May 2009";
libname data_jun "L:\Lightstone\System Testing\System Testing 22 Jun 2009";

/* Use the logistic6 files from Jan-May 09 */
/* Flagcompare is used for data cleaning purposes, to ensure the purchase price entered from */
/* the system is reliable and can be used as the purchase price of a property    */
data neural.logistic6_jan;
set data_jan.logistic6;
where flagcompare = "COMPARE";
run;

data neural.logistic6_feb;
set data_feb.logistic6;
where flagcompare = "COMPARE";
run;

data neural.logistic6_mar;
set data_mar.logistic6;
where flagcompare = "COMPARE";
run;

data neural.logistic6_apr;
set data_apr.logistic6;
```

```
where flagcompare = "COMPARE";
run;

data neural.logistic6_may;
set data_may.logistic6_may;
where flagcompare = "COMPARE";
run;

data neural.logistic6_jun;
set data_jun.logistic6;
where flagcompare = "COMPARE";
run;

/* Make one big file from the past few month's logistic6 files */
data neural.logistic6_augmented (keep= property_id my ssflag mod_seg purchase_price
    predicted_value
    IPURCHDATE flag_used comp_num_used csflag
    churn newmonthdiff_query
    predval_RS p_ab_rs p_90_rs null_RS
    predval_CS p_ab_cs p_90_cs null_CS
    better
    sigma1_sq sigma2_sq
    predval_comb sigma_comb p_ab_comb p_90_comb
    pred_method
    predval_final p_ab_final p_90_final choice right
    best_pred
    );
set neural.logistic6_jan neural.logistic6_feb neural.logistic6_mar
    neural.logistic6_apr neural.logistic6_may;
if predval_cs ^= . & predval_rs ^= .;
*Neural network will only be trained on records for which we have a RS and CS pred;
run;

/* Prepare the file */
```

```
data neural.logistic6_keep;
set neural.logistic6_augmented;
length optimal optimal2 optimal3 optimal4 optimal5 $4;
length overunder $10;
length overun $40;

ipurchdate_date =mdy(substr(put(ipurchdate,8.),5,2),
                        substr(put(ipurchdate,8.),7,2),substr(put(ipurchdate,8.),1,4));
diff = intck('month',ipurchdate_date,my);
error_cs = predval_cs - purchase_price;
error_rs = predval_rs - purchase_price;
error_comb = predval_comb - purchase_price;
per_error_cs = (predval_cs/purchase_price)*100;
per_error_rs =  (predval_rs/purchase_price)*100;
per_error_comb = (predval_comb/purchase_price)*100;

/* Remove some of the unrelaible records */
if predicted_value = 0 then delete;
if null_RS = 'USE' and null_CS = 'USE';
if per_error_rs > 210 then delete;
if per_error_cs > 210 then delete;
if comp_num_used > 200 then delete;

sigma_rs = sqrt(sigma1_sq);
sigma_cs = sqrt(sigma2_sq);

lsigma_rs = log(sigma_rs);
lsigma_cs = log(sigma_cs);
lsigma_comb = log(sigma_comb);

/*Defines the prediction that is the closest to the entered purchase price*/
best_error = min(abs(error_cs),abs(error_rs),abs(error_comb));
if best_error = abs(error_comb) then absolute_best = 'COMB';
else if best_error = abs(error_cs) then absolute_best = 'CS';
```

```
else if best_error = abs(error_rs) then absolute_best = 'RS';

if absolute_best = 'COMB' then absolute_best_pred = predval_comb;
if absolute_best = 'RS' then absolute_best_pred = predval_rs;
if absolute_best = 'CS' then absolute_best_pred = predval_cs;

if  0.8*purchase_price < absolute_best_pred < 1.2*purchase_price then ab_abs = 1;
else ab_abs = 0;
optimal = absolute_best; *Define dependent variable that will be used;

error_percentage_rs = per_error_rs - 100;
error_percentage_cs = per_error_cs - 100;
error_percentage_comb = per_error_comb - 100;

if 0.7*purchase_price <= predval_final <= 1.3*purchase_price then per_curr30 = 1;
else per_curr30 = 0;

if 0.8*purchase_price <= predval_final <= 1.2*purchase_price then per_curr20 = 1;
else per_curr20 = 0;

if 0.9*purchase_price <= predval_final <= 1.1*purchase_price then per_curr10 = 1;
else per_curr10 = 0;

if 0.95*purchase_price <= predval_final <= 1.05*purchase_price then per_curr05 = 1;
else per_curr05 = 0;

right_curr = 0;
if pred_method = 'A: USED COMBINED PREDICTION' and optimal = 'COMB' then right_curr = 1;
else if pred_method = 'B: USED RS PREDICTION' and optimal = 'RS' then right_curr = 1;
else if pred_method = 'C: USED COMP SALES PREDICTION' and optimal = 'CS' then right_curr = 1;

obs_nr = _n_;

if predval_final > purchase_price then overunder = 'OVER'; else overunder = 'UNDER';
```

```
if 1*purchase_price <= predval_final < 1.1*purchase_price then overun = 'OVER 0-10%';
else if 1.1*purchase_price <= predval_final < 1.2*purchase_price then overun = 'OVER 10-20%';
else if 1.2*purchase_price <= predval_final < 1.3*purchase_price then overun = 'OVER 20-30%';
else if predval_final > 1.3*purchase_price then overun = 'OVER >30%';
else if 0.9*purchase_price <= predval_final < 1*purchase_price then overun = 'UNDER 0-10%';
else if 0.8*purchase_price <= predval_final < 0.9*purchase_price then overun = 'UNDER 10-20%';
else if 0.7*purchase_price <= predval_final < 0.8*purchase_price then overun = 'UNDER 20-30%';
else if predval_final <= 0.7*purchase_price then overun = 'UNDER >30%';
run;

proc freq data=neural.logistic6_keep;
tables optimal;
run;

/* Generate statistics on the performance of the current Lightstone prediction */
proc freq data = neural.logistic6_keep;
tables ab_abs per_curr05 per_curr10 per_curr20 per_curr30 right_curr overunder overun;
run;

/* Get the file ready for use in neural network training in iml, i.e. code all */
/* the relevant variables correctly and keep only the variables that will be used */

data neural.use (keep = purchase_price
predval_rs p_ab_rs p_90_rs
predval_cs p_ab_cs p_90_cs
predval_comb p_ab_comb p_90_comb
lsigma_rs lsigma_cs lsigma_comb
churn comp_num_used  newmonthdiff_query
csflag1-csflag19 flag_used1-flag_used3 mod_seg1-mod_seg16
optimal1 - optimal3 obs_nr);
retain  purchase_price
predval_rs p_ab_rs p_90_rs
predval_cs p_ab_cs p_90_cs
```

```
predval_comb p_ab_comb p_90_comb
lsigma_rs lsigma_cs lsigma_comb
churn comp_num_used newmonthdiff_query
mod_seg1-mod_seg16 csflag1-csflag19 flag_used1-flag_used3
optimal1 - optimal3 obs_nr;
set neural.logistic6_keep (drop = optimal2-optimal5);

if substr(mod_seg,1,index(mod_seg,":")-1) = 'A' then mod_seg1 = 1;
else mod_seg1 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'B' then mod_seg2 = 1;
else mod_seg2 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'E1' then mod_seg3 = 1;
else mod_seg3 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'E2' then mod_seg4 = 1;
else mod_seg4 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'E3' then mod_seg5 = 1;
else mod_seg5 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'F' then mod_seg6 = 1;
else mod_seg6 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'G1' then mod_seg7 = 1;
else mod_seg7 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'G2' then mod_seg8 = 1;
else mod_seg8 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'G3' then mod_seg9 = 1;
else mod_seg9 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'H' then mod_seg10 = 1;
else mod_seg10 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'I' then mod_seg11 = 1;
else mod_seg11 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'J' then mod_seg12 = 1;
else mod_seg12 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'K' then mod_seg13 = 1;
else mod_seg13 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'L' then mod_seg14 = 1;
```

```
else mod_seg14 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'Z1' then mod_seg15 = 1;
else mod_seg15 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'Z2' then mod_seg16 = 1;
else mod_seg16 = 0;

if csflag = "ADJEA1 FH" then csflag1 = 1; else csflag1 = 0;
if csflag = "ADJEA1 FH TSHIP" then csflag2 = 1; else csflag2 = 0;
if csflag = "ADJEA1 SS" then csflag3 = 1; else csflag3 = 0;
if csflag = "ADJEA2 FH" then csflag4 = 1; else csflag4 = 0;
if csflag = "ADJEA2 FH TSHIP" then csflag5 = 1; else csflag5 = 0;
if csflag = "ADJEA2 SS" then csflag6 = 1; else csflag6 = 0;
if csflag = "ADJEA3 FH" then csflag7 = 1; else csflag7 = 0;
if csflag = "ADJEA3 FH TSHIP" then csflag8 = 1; else csflag8 = 0;
if csflag = "ADJEA3 SS" then csflag9 = 1; else csflag9 = 0;
if csflag = "EST ONLY" then csflag10 = 1; else csflag10 = 0;
if csflag = "FH EA SS" then csflag11 = 1; else csflag11 = 0;
if csflag = "OWN EA FH" then csflag12 = 1; else csflag12 = 0;
if csflag = "OWN EA FH TSHIP" then csflag13 = 1; else csflag13 = 0;
if csflag = "OWN EA FH SS" then csflag14 = 1; else csflag14 = 0;
if csflag = "SP FH" then csflag15 = 1; else csflag15 = 0;
if csflag = "SP FH TSHIP" then csflag16 = 1; else csflag16 = 0;
if csflag = "SP SS" then csflag17 = 1; else csflag17 = 0;
if csflag = "SS ONLY" then csflag18 = 1; else csflag18 = 0;
if csflag = "TSHIP SHRT SP" then csflag19 = 1; else csflag19 = 0;

if flag_used = "A: USED 0506" then flag_used1 = 1; else flag_used1 = 0;
if flag_used = "B: USED 0306" then flag_used2 = 1; else flag_used2 = 0;
if flag_used = "C: USED CHOSEN" then flag_used3 = 1; else flag_used3 = 0;

if optimal = 'COMB' then optimal1 = 1;
else optimal1 = 0;
if optimal = 'CS' then optimal2 = 1;
else optimal2 = 0;
```

```
if optimal = 'RS' then optimal3 = 1;
else optimal3 = 0;

if purchase_price = . then delete;
if predval_rs = . then delete;
if p_ab_rs = . then delete;
if p_90_rs = . then delete;
if lsigma_rs = . then delete;
if predval_cs = . then delete;
if p_ab_cs = . then delete;
  if p_90_cs = . then delete;
if lsigma_cs = . then delete;
if predval_comb = . then delete;
if p_ab_comb = . then delete;
if p_90_comb = . then delete;
if lsigma_comb = . then delete;
if newmonthdiff_query = . then delete;
run;

/*TAKE THE REMAINING DATASET TO IML TO TRAIN THE NEURAL NETWORK*/
ods html close;
proc iml;

/*Function to do standardization*/
start standardize(input_data,cols_to_standardize,standardized_data,mean,std);
mean = input_data[:,cols_to_standardize];
   cov=(t(input_data[,cols_to_standardize])*input_data[,cols_to_standardize]
            -nrow(input_data)*(t(mean)*mean))/(nrow(input_data)-1);
std = sqrt(vecdiag(cov));
standardized_data=(input_data[,cols_to_standardize]-
    shape(mean,nrow(input_data),ncol(mean)))/shape(t(std),nrow(input_data),ncol(mean));
finish standardize;
/* Finish standardize function */
```

```
use neural.use;
read all into data;
subset = data[1:10,];
print subset;
names = {'PP' 'PREDVAL_RS' 'P_AB_RS' 'P_90_RS'
'PREDVAL_CS' 'P_AB_CS' 'P_90_CS'
'PREDVAL_COMB' 'P_AB_COMB' 'P_90_COMB'
'LSIGMA_RS' 'LSIGMA_CS' 'LSIGMA_COMB'
'CHURN' 'COMP' 'MONTHDIFF'
'MOD_SEG1' 'MOD_SEG2' 'MOD_SEG3' 'MOD_SEG4' 'MOD_SEG5' 'MOD_SEG6' 'MOD_SEG7' 'MOD_SEG8'
'MOD_SEG9' 'MOD_SEG10' 'MOD_SEG11' 'MOD_SEG12' 'MOD_SEG13' 'MOD_SEG14' 'MOD_SEG15' 'MOD_SEG16'
'CSFLAG1' 'CSFLAG2' 'CSFLAG3' 'CSFLAG4' 'CSFLAG5' 'CSFLAG6' 'CSFLAG7' 'CSFLAG8' 'CSFLAG9'
            'CSFLAG10' 'CSFLAG11' 'CSFLAG12' 'CSFLAG13' 'CSFLAG14' 'CSFLAG15' 'CSFLAG16' 'CSFLAG17'
            'CSFLAG18' 'CSFLAG19'
'FLAGUSED1' 'FLAGUSED2' 'FLAGUSED3'
'OPTIMAL1' 'OPTIMAL2' 'OPTIMAL3'
'OBS_NR'
};
print subset[c=names];
colx = {2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
            34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54};
coly = {55 56 57};
X = data[,colx];
Y = data[,coly];
PP = data[,1];

/*STANDARDIZE THE RELEVANT INPUTS*/
call standardize(X,1:15,standardized_inputs,mean,std);
X = standardized_inputs || X[,16:53];

XY = X || Y;

subset = XY[1:10,];
print subset;
```

```
/*SPLIT INTO A MODELLING AND TESTING DATASET*/
percentage_testing = 0.2;
n = nrow(XY);
u = uniform(J(n,1,0));
do i = 1 to n;
if u[i] <= (1-percentage_testing) then modelling = modelling // i;
else testing = testing // i;
end;

modelling_x = XY[modelling,1:53];
modelling_y = XY[modelling,54:56];
testing_x = XY[testing,1:53];
testing_y = XY[testing,54:56];
testing_pp = PP[testing,1];

/* Define the basic parameters of the neural network */
n_epoch = 0;
n_obs = nrow(modelling_x);
n_input = ncol(modelling_x);
n_output = ncol(modelling_y);
n_hidden = 50;
alpha = 0.005;
wtrange = 0.05;
momentum = 0.4;

print "Starting training a neural network with the following parameters"
            "Training observations:" n_obs "Independent variables:" n_input "Hidden nodes:" n_hidden
            "Number of output nodes:" n_output "Learning rate:" alpha "Momentum rate:" momentum;

/*Initialize network*/
sequence = shape(0,n_obs,1);
bias_ih = shape(0,1,n_hidden);
wt_ih = shape(0,n_input,n_hidden);
```

```
bias_ho = shape(0,1,n_output);
wt_ho = shape(0,n_hidden,n_output);

delta_wt_ih_prev = shape(0,n_input,n_hidden);
delta_bias_ih_prev = shape(0,1,n_hidden);
delta_wt_ho_prev = shape(0,n_hidden,n_output);
delta_bias_ho_prev = shape(0,1,n_output);

row_name_wt_ih = t(char(1:n_input));
col_name_wt_ih = char(1:n_hidden);
row_name_wt_ho = t(char(1:n_hidden));
col_name_wt_ho = char(1:n_output);

do i = 1 to n_obs;
sequence[i] = i;
end;

/*Initialize connection weights*/
wt_ih = 2*wtrange*uniform(wt_ih) - J(nrow(wt_ih),ncol(wt_ih),wtrange);
wt_ho = 2*wtrange*uniform(wt_ho) - J(nrow(wt_ho),ncol(wt_ho),wtrange);
bias_ih = 2*wtrange*uniform(bias_ih) - J(1,ncol(bias_ih),wtrange);
bias_ho = 2*wtrange*uniform(bias_ho) - J(1,ncol(bias_ho),wtrange);

print wt_ih[r=row_name_wt_ih c=col_name_wt_ih]  wt_ho[r=row_name_wt_ho c=col_name_wt_ho]
bias_ih[c=col_name_wt_ih] bias_ho[c=col_name_wt_ho];

do t = 1 to 1000;

n_epoch = n_epoch + 1;
tss = 0;

do i=n_obs to 2 by -1;
   a=int(uniform(0)*i)+1;
   b=sequence[i];
```

```
   sequence[i]=sequence[a];
   sequence[a]=b;
end;

do i = 1 to n_obs;
pss = 0;

obs_nr = sequence[i];
input = modelling_x[obs_nr,];
output = modelling_y[obs_nr,];

z_in = bias_ih#J(1,ncol(wt_ih),1) + input*wt_ih;
z = (1+exp(-z_in))##(-1);
y_in = bias_ho#J(1,ncol(wt_ho),1) + z*wt_ho;
y_hat = exp(y_in)/sum(exp(y_in));

check = y_hat[,+];

error = output - y_hat;
delta_2 = error;

delta_wt_ho = t(z)*delta_2 + momentum#delta_wt_ho_prev;
delta_bias_ho = delta_2 + momentum#delta_bias_ho_prev;

delta_1 = (z#(1-z))#t(wt_ho*t(delta_2));

delta_wt_ih = t(input)*delta_1 + momentum#delta_wt_ih_prev;
delta_bias_ih = delta_1 + momentum#delta_bias_ih_prev;

wt_ih_old = wt_ih;
wt_ho_old = wt_ho;
delta_wt_ih_prev = delta_wt_ih;
delta_bias_ih_prev = delta_bias_ih;
delta_wt_ho_prev = delta_wt_ho;
```

```
delta_bias_ho_prev = delta_bias_ho;

bias_ih = bias_ih  + alpha#delta_bias_ih;
wt_ih  = wt_ih  + alpha#delta_wt_ih;
bias_ho = bias_ho  + alpha#delta_bias_ho;
wt_ho  = wt_ho  + alpha#delta_wt_ho;
end;



if ncol(testing_results) > 0 then do;
testing_results = remove(testing_results,1:(nrow(testing_results)*ncol(testing_results)));
end;

do i = 1 to nrow(testing_x);
input = testing_x[i,];
output = testing_y[i,];

z_in = bias_ih#J(1,ncol(wt_ih),1) + input*wt_ih;
z = (1+exp(-z_in))##(-1);
y_in = bias_ho#J(1,ncol(wt_ho),1) + z*wt_ho;
y_hat = exp(y_in)/sum(exp(y_in));

max = max(y_hat);
do k = 1 to ncol(y_hat);
if max = y_hat[k] then classification = k;
end;
do k = 1 to ncol(output);
correct_class = correct_class || (output[,k] = 1)*k;
end;

correct_class = correct_class[,+];
if classification = 1 then prediction = input[,7]*std[7] + mean[7];
else if classification = 2 then prediction = input[,4]*std[4] + mean[4];
else if classification = 3 then prediction = input[,1]*std[1] + mean[1];
```

```
if prediction >= 0.8*testing_pp[i] & prediction <= 1.2*testing_pp[i] then ab = 1;
else ab = 0;

testing_results = testing_results // (correct_class || classification || ab);
correct_class = remove(correct_class,1);

end;

mcr = 100*(1-(testing_results[,1] = testing_results[,2])[+]/nrow(testing_results));
ab_rate = testing_results[:,3];
n_test = nrow(testing_results);
mcr_mat = mcr_mat // (t || mcr || ab_rate);

if nrow(mcr_mat) > 1 then do;
if ab_rate > max(mcr_mat[1:(nrow(mcr_mat)-1),3]) then do;
best_iteration = t;
best_mcr = mcr;
best_wt_ih = wt_ih;
best_bias_ih = bias_ih;
best_wt_ho = wt_ho;
best_bias_ho = bias_ho;
end;
end;

end;

print "Total number of epocs:" n_epoch;
print "Epoch with lowest mcr:" best_iteration;
print "Input-->Hidden Bias" best_bias_ih;
print "Input-->Hidden Weight Matrix" best_wt_ih;
print "Hidden-->Output Bias" best_bias_ho;
print "Hidden-->Output Weight Matrix" best_wt_ho;
```

```
/* Use the best weights which is obtained from the independent test sample to give */
/* detailed statistics about the net */
if ncol(testing_results) > 0 then do;
testing_results = remove(testing_results,1:(nrow(testing_results)*ncol(testing_results)));
end;

do i = 1 to nrow(testing_x);
input = testing_x[i,];
output = testing_y[i,];

z_in = best_bias_ih#J(1,ncol(wt_ih),1) + input*best_wt_ih;
z = (1+exp(-z_in))##(-1);
y_in = best_bias_ho#J(1,ncol(wt_ho),1) + z*best_wt_ho;
y_hat = exp(y_in)/sum(exp(y_in));

max = max(y_hat);
do k = 1 to ncol(y_hat);
if max = y_hat[k] then classification = k;
end;
do k = 1 to ncol(output);
correct_class = correct_class || (output[,k] = 1)*k;
end;

correct_class = correct_class[,+];
if classification = 1 then prediction = input[,7]*std[7] + mean[7];
else if classification = 2 then prediction = input[,4]*std[4] + mean[4];
else if classification = 3 then prediction = input[,1]*std[1] + mean[1];
if prediction >= 0.8*testing_pp[i] & prediction <= 1.2*testing_pp[i] then ab = 1;
else ab = 0;

testing_results = testing_results // (correct_class || classification || ab);
correct_class = remove(correct_class,1);
end;
ab_result = testing_results[:,3];
```

```
mcr = 100*(1-((testing_results[,1] = testing_results[,2])[+]/nrow(testing_results)));
n_test = nrow(testing_results);
print "Testing AB: " ab_result;
print "Misclassification rate is:" mcr "%";
print "Number of testing set observations:" n_test;

/*Now use the trained neural network on the entire dataset*/
do i = 1 to nrow(X);

input = X[i,];
output = Y[i,];

z_in = best_bias_ih#J(1,ncol(wt_ih),1) + input*best_wt_ih;
z = (1+exp(-z_in))##(-1);
y_in = best_bias_ho#J(1,ncol(wt_ho),1) + z*best_wt_ho;
y_hat = exp(y_in)/sum(exp(y_in));

max = max(y_hat);
do k = 1 to ncol(y_hat);
if max = y_hat[k] then classification = k;
end;
do k = 1 to ncol(output);
correct_class = correct_class || (output[,k] = 1)*k;
end;

correct_class = correct_class[,+];
if classification = 1 then prediction = input[,7]*std[7] + mean[7];
else if classification = 2 then prediction = input[,4]*std[4] + mean[4];
else if classification = 3 then prediction = input[,1]*std[1] + mean[1];

if prediction >= 0.7*data[i,1] & prediction <= 1.3*data[i,1] then neural30 = 1;
else neural30 = 0;
if prediction >= 0.8*data[i,1] & prediction <= 1.2*data[i,1] then neural20 = 1;
else neural20 = 0;
```

```
if prediction >= 0.9*data[i,1] & prediction <= 1.1*data[i,1] then neural10 = 1;
else neural10 = 0;
if prediction >= 0.95*data[i,1] & prediction <= 1.05*data[i,1] then neural05 = 1;
else neural05 = 0;



final_results = final_results //
            (correct_class || classification || neural05 || neural10 || neural20 || neural30);
pp = data[i,1];
obs = data[i,58];
record = obs || pp || correct_class || classification || prediction || y_hat;
output_data = output_data // record;

correct_class = remove(correct_class,1);
end;

per05 = final_results[:,3];
per10 = final_results[:,4];
per20 = final_results[:,5];
per30 = final_results[:,6];
mcr = 100*(1-((final_results[,1] = final_results[,2])[+]/nrow(final_results)));
n_all = nrow(final_results);
print "% within 5% " per05;
print "% within 10% " per10;
print "% within 20% " per20;
print "% within 30% " per30;
print "Misclassification rate is:" mcr "%";
print "Number of observations:" n_all;

/* Need to output the weights for future use of the neural network */
std = t(std);
create mean from mean;
append from mean;
```

```
create std from std;
append from std;

create bias_ih from bias_ih;
append from bias_ih;

create wt_ih from wt_ih;
append from wt_ih;

create bias_ho from bias_ho;
append from bias_ho;

create wt_ho from wt_ho;
append from wt_ho;

names1 = {'OBS_NR' 'PURCHASE_PRICE'
'CORRECT CLASS' 'PREDICTED CLASS' 'FINAL PREDICTION' "P_COMB" "P_CS" "P_RS"};

create output_bestpick from output_data[c=names1];
append from output_data;

quit;


/* Now that the neural network is trained and the best neural network is saved, */
/* we can do analysis */
proc format;
value preds 1 = 'COMB'
  2 = 'CS'
3 = 'RS';
run;

proc sql;
create table results_linked as
```

```
select a.*,b.predval_rs,b.predval_cs,b.predval_comb,b.predval_final,b.pred_method,mod_seg
from neural.output_bestpick a left join neural.logistic6_keep b
on a.obs_nr = b.obs_nr;
quit;

data neural.results_linked;
set results_linked;
format correct_class predicted_class preds.;
length overunder $10 overun $40;
length predicted_class2 $4;
if 0.8*purchase_price <= final_prediction <= 1.2*purchase_price then ab_neural = 1;
else ab_neural = 0;
if 0.8*purchase_price <= predval_final <= 1.2*purchase_price then ab_curr = 1;
else ab_curr = 0;

/* Modified classification rule which works with the posterior probabilities */
predicted_class_modified = 'COMB';
if p_rs > 0.5 then predicted_class_modified = 'RS';
else if p_cs > 0.5 then predicted_class_modified = 'CS';

predicted_val_modified = predval_comb;
if p_rs > 0.5 then predicted_val_modified = predval_rs;
else if p_cs > 0.5 then predicted_val_modified = predval_cs;

if 0.95*purchase_price <= predicted_val_modified <= 1.05*purchase_price then ab_neural_05 = 1;
else ab_neural_05 = 0;
if 0.9*purchase_price <= predicted_val_modified <= 1.1*purchase_price then ab_neural_10 = 1;
else ab_neural_10 = 0;
if 0.8*purchase_price <= predicted_val_modified <= 1.2*purchase_price then ab_neural_20 = 1;
else ab_neural_20 = 0;
if 0.7*purchase_price <= predicted_val_modified <= 1.3*purchase_price then ab_neural_30 = 1;
else ab_neural_30 = 0;

if predicted_val_modified > purchase_price then overunder = 'OVER'; else overunder = 'UNDER';
```

```
if 1*purchase_price <= predicted_val_modified < 1.1*purchase_price
                                            then overun = 'OVER 0-10%';
else if 1.1*purchase_price <= predicted_val_modified < 1.2*purchase_price
                                            then overun = 'OVER 10-20%';
else if 1.2*purchase_price <= predicted_val_modified < 1.3*purchase_price
                                            then overun = 'OVER 20-30%';
else if predicted_val_modified >= 1.3*purchase_price
                                            then overun = 'OVER >30%';
else if 0.9*purchase_price <= predicted_val_modified < 1*purchase_price
                                            then overun = 'UNDER 0-10%';
else if 0.8*purchase_price <= predicted_val_modified < 0.9*purchase_price
                                            then overun = 'UNDER 10-20%';
else if 0.7*purchase_price <= predicted_val_modified < 0.8*purchase_price
                                            then overun = 'UNDER 20-30%';
else if predicted_val_modified <= 0.7*purchase_price
                                            then overun = 'UNDER >30%';

run;

/* Look at performance statistics from neural network with the modified classification rule */
proc freq data=neural.results_linked;
tables ab_neural ab_curr ab_neural_05 ab_neural_10 ab_neural_20 ab_neural_30 overunder overun;
run;


/* Test the neural network approach on a totally independent dataset */
data june_sys_data;
set logistic6_jun;
if timestamp >= '01JUN09'd;
run;

data june_sys_data;
set june_sys_data;
```

```
length overunder $10;
length overun $40;
if predval_rs ^=. & predval_cs ^= .;

sigma_rs = sqrt(sigma1_sq);
sigma_cs = sqrt(sigma2_sq);

lsigma_rs = log(sigma_rs);
lsigma_cs = log(sigma_cs);
lsigma_comb = log(sigma_comb);

/*Defines the prediction that is the closest to the entered purchase price*/
best_error = min(abs(error_cs),abs(error_rs),abs(error_comb));
if best_error = abs(error_comb) then absolute_best = 'COMB';
else if best_error = abs(error_cs) then absolute_best = 'CS';
else if best_error = abs(error_rs) then absolute_best = 'RS';

if absolute_best = 'COMB' then absolute_best_pred = predval_comb;
if absolute_best = 'RS' then absolute_best_pred = predval_rs;
if absolute_best = 'CS' then absolute_best_pred = predval_cs;

optimal = absolute_best;
OBS_NR = _n_;

if 0.7*purchase_price <= predval_final <= 1.3*purchase_price then per_curr30 = 1;
else per_curr30 = 0;

if 0.8*purchase_price <= predval_final <= 1.2*purchase_price then per_curr20 = 1;
else per_curr20 = 0;

if 0.9*purchase_price <= predval_final <= 1.1*purchase_price then per_curr10 = 1;
else per_curr10 = 0;

if 0.95*purchase_price <= predval_final <= 1.05*purchase_price then per_curr05 = 1;
```

```
else per_curr05 = 0;

if predval_final > purchase_price then overunder = 'OVER'; else overunder = 'UNDER';

if 1*purchase_price <= predval_final < 1.1*purchase_price then overun = 'OVER 0-10%';
else if 1.1*purchase_price <= predval_final < 1.2*purchase_price then overun = 'OVER 10-20%';
else if 1.2*purchase_price <= predval_final < 1.3*purchase_price then overun = 'OVER 20-30%';
else if predval_final > 1.3*purchase_price then overun = 'OVER >30%';
else if 0.9*purchase_price <= predval_final < 1*purchase_price then overun = 'UNDER 0-10%';
else if 0.8*purchase_price <= predval_final < 0.9*purchase_price then overun = 'UNDER 10-20%';
else if 0.7*purchase_price <= predval_final < 0.8*purchase_price then overun = 'UNDER 20-30%';
else if predval_final <= 0.7*purchase_price then overun = 'UNDER >30%';

if purchase_price = . then delete;
if predval_rs = . then delete;
if p_ab_rs = . then delete;
if p_90_rs = . then delete;
if lsigma_rs = . then delete;
if predval_cs = . then delete;
if p_ab_cs = . then delete;
  if p_90_cs = . then delete;
if lsigma_cs = . then delete;
if predval_comb = . then delete;
if p_ab_comb = . then delete;
if p_90_comb = . then delete;
if lsigma_comb = . then delete;
if newmonthdiff_query = . then delete;

run;

/* Performance statistics of the Lightstone method on this new independent testing set */
proc freq data = june_sys_data;
tables per_curr05 per_curr10 per_curr20 per_curr30 overunder overun;
run;
```

```
data neural.test (keep = purchase_price
predval_rs p_ab_rs p_90_rs
predval_cs p_ab_cs p_90_cs
predval_comb p_ab_comb p_90_comb
lsigma_rs lsigma_cs lsigma_comb
churn comp_num_used  newmonthdiff_query
csflag1-csflag19 flag_used1-flag_used3 mod_seg1-mod_seg16
optimal1 - optimal3 obs_nr);
retain  purchase_price
predval_rs p_ab_rs p_90_rs
predval_cs p_ab_cs p_90_cs
predval_comb p_ab_comb p_90_comb
lsigma_rs lsigma_cs lsigma_comb
churn comp_num_used newmonthdiff_query
mod_seg1-mod_seg16 csflag1-csflag19 flag_used1-flag_used3
optimal1 - optimal3 obs_nr;
set june_sys_data (drop = csflag2);

if substr(mod_seg,1,index(mod_seg,":")-1) = 'A' then mod_seg1 = 1;
else mod_seg1 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'B' then mod_seg2 = 1;
else mod_seg2 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'E1' then mod_seg3 = 1;
else mod_seg3 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'E2' then mod_seg4 = 1;
else mod_seg4 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'E3' then mod_seg5 = 1;
else mod_seg5 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'F' then mod_seg6 = 1;
else mod_seg6 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'G1' then mod_seg7 = 1;
else mod_seg7 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'G2' then mod_seg8 = 1;
```

```
else mod_seg8 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'G3' then mod_seg9 = 1;
else mod_seg9 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'H' then mod_seg10 = 1;
else mod_seg10 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'I' then mod_seg11 = 1;
else mod_seg11 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'J' then mod_seg12 = 1;
else mod_seg12 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'K' then mod_seg13 = 1;
else mod_seg13 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'L' then mod_seg14 = 1;
else mod_seg14 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'Z1' then mod_seg15 = 1;
else mod_seg15 = 0;
if substr(mod_seg,1,index(mod_seg,":")-1) = 'Z2' then mod_seg16 = 1;
else mod_seg16 = 0;

if csflag = "ADJEA1 FH" then csflag1 = 1; else csflag1 = 0;
if csflag = "ADJEA1 FH TSHIP" then csflag2 = 1; else csflag2 = 0;
if csflag = "ADJEA1 SS" then csflag3 = 1; else csflag3 = 0;
if csflag = "ADJEA2 FH" then csflag4 = 1; else csflag4 = 0;
if csflag = "ADJEA2 FH TSHIP" then csflag5 = 1; else csflag5 = 0;
if csflag = "ADJEA2 SS" then csflag6 = 1; else csflag6 = 0;
if csflag = "ADJEA3 FH" then csflag7 = 1; else csflag7 = 0;
if csflag = "ADJEA3 FH TSHIP" then csflag8 = 1; else csflag8 = 0;
if csflag = "ADJEA3 SS" then csflag9 = 1; else csflag9 = 0;
if csflag = "EST ONLY" then csflag10 = 1; else csflag10 = 0;
if csflag = "FH EA SS" then csflag11 = 1; else csflag11 = 0;
if csflag = "OWN EA FH" then csflag12 = 1; else csflag12 = 0;
if csflag = "OWN EA FH TSHIP" then csflag13 = 1; else csflag13 = 0;
if csflag = "OWN EA FH SS" then csflag14 = 1; else csflag14 = 0;
if csflag = "SP FH" then csflag15 = 1; else csflag15 = 0;
if csflag = "SP FH TSHIP" then csflag16 = 1; else csflag16 = 0;
```

```
if csflag = "SP SS" then csflag17 = 1; else csflag17 = 0;
if csflag = "SS ONLY" then csflag18 = 1; else csflag18 = 0;
if csflag = "TSHIP SHRT SP" then csflag19 = 1; else csflag19 = 0;

if flag_used = "A: USED 0506" then flag_used1 = 1; else flag_used1 = 0;
if flag_used = "B: USED 0306" then flag_used2 = 1; else flag_used2 = 0;
if flag_used = "C: USED CHOSEN" then flag_used3 = 1; else flag_used3 = 0;



if optimal = 'COMB' then optimal1 = 1;
else optimal1 = 0;
if optimal = 'CS' then optimal2 = 1;
else optimal2 = 0;
if optimal = 'RS' then optimal3 = 1;
else optimal3 = 0;

if purchase_price = . then delete;
if predval_rs = . then delete;
if p_ab_rs = . then delete;
if p_90_rs = . then delete;
if lsigma_rs = . then delete;
if predval_cs = . then delete;
if p_ab_cs = . then delete;
  if p_90_cs = . then delete;
if lsigma_cs = . then delete;
if predval_comb = . then delete;
if p_ab_comb = . then delete;
if p_90_comb = . then delete;
if lsigma_comb = . then delete;
if newmonthdiff_query = . then delete;
run;

/* Put this testing set through the neural network that was trained on the Jan-May data */
ods html close;
```

```
proc iml;

use neural.test;
read all into data;
subset = data[1:10,];
print subset;
names = {'PP' 'PREDVAL_RS' 'P_AB_RS' 'P_90_RS'
'PREDVAL_CS' 'P_AB_CS' 'P_90_CS'
'PREDVAL_COMB' 'P_AB_COMB' 'P_90_COMB'
'LSIGMA_RS' 'LSIGMA_CS' 'LSIGMA_COMB'
'CHURN' 'COMP' 'MONTHDIFF'
'MOD_SEG1' 'MOD_SEG2' 'MOD_SEG3' 'MOD_SEG4' 'MOD_SEG5' 'MOD_SEG6' 'MOD_SEG7' 'MOD_SEG8'
'MOD_SEG9' 'MOD_SEG10' 'MOD_SEG11' 'MOD_SEG12' 'MOD_SEG13' 'MOD_SEG14' 'MOD_SEG15' 'MOD_SEG16'
'CSFLAG1' 'CSFLAG2' 'CSFLAG3' 'CSFLAG4' 'CSFLAG5' 'CSFLAG6' 'CSFLAG7' 'CSFLAG8' 'CSFLAG9'
            'CSFLAG10' 'CSFLAG11' 'CSFLAG12' 'CSFLAG13' 'CSFLAG14' 'CSFLAG15' 'CSFLAG16' 'CSFLAG17'
            'CSFLAG18' 'CSFLAG19'
'FLAGUSED1' 'FLAGUSED2' 'FLAGUSED3'
'OPTIMAL1' 'OPTIMAL2' 'OPTIMAL3'
'OBS_NR'
};
print subset[c=names];
colx = {2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
            32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54};
coly = {55 56 57};
X = data[,colx];
Y = data[,coly];
PP = data[,1];

/*STANDARDIZE THE RELEVANT INPUTS*/
use neural.mean;
read all into mean;

use neural.std;
read all into std;
```

```
standardized_data = (X[,1:15] - shape(mean,nrow(X),ncol(mean)))/shape(t(std),nrow(X),ncol(mean));
X = standardized_data || X[,16:53];

XY = X || Y;

subset = XY[1:10,];
print subset;

use neural.bias_ih;
read all into bias_ih;

use neural.wt_ih;
read all into wt_ih;

use neural.bias_ho;
read all into bias_ho;

use neural.wt_ho;
read all into wt_ho;

do i = 1 to nrow(X);

input = X[i,];
output = Y[i,];

z_in = bias_ih#J(1,ncol(wt_ih),1) + input*wt_ih;
z = (1+exp(-z_in))##(-1);
y_in = bias_ho#J(1,ncol(wt_ho),1) + z*wt_ho;
y_hat = exp(y_in)/sum(exp(y_in));

max = max(y_hat);
do k = 1 to ncol(y_hat);
if max = y_hat[k] then classification = k;
```

```
end;
do k = 1 to ncol(output);
correct_class = correct_class || (output[,k] = 1)*k;
end;

correct_class = correct_class[,+];
if classification = 1 then prediction = input[,7]*std[7] + mean[7];
else if classification = 2 then prediction = input[,4]*std[4] + mean[4];
else if classification = 3 then prediction = input[,1]*std[1] + mean[1];

if prediction >= 0.7*data[i,1] & prediction <= 1.3*data[i,1] then neural30 = 1;
else neural30 = 0;
if prediction >= 0.8*data[i,1] & prediction <= 1.2*data[i,1] then neural20 = 1;
else neural20 = 0;
if prediction >= 0.9*data[i,1] & prediction <= 1.1*data[i,1] then neural10 = 1;
else neural10 = 0;
if prediction >= 0.95*data[i,1] & prediction <= 1.05*data[i,1] then neural05 = 1;
else neural05 = 0;



final_results = final_results //
            (correct_class || classification || neural05 || neural10 || neural20 || neural30);
pp = data[i,1];
obs = data[i,58];
record = obs || pp || correct_class || classification || prediction || y_hat;
output_data = output_data // record;

correct_class = remove(correct_class,1);
end;

per05 = final_results[:,3];
per10 = final_results[:,4];
per20 = final_results[:,5];
per30 = final_results[:,6];
```

```
mcr = 100*(1-((final_results[,1] = final_results[,2])[+]/nrow(final_results)));
n_all = nrow(final_results);
print "% within 5% " per05;
print "% within 10% " per10;
print "% within 20% " per20;
print "% within 30% " per30;
print "Misclassification rate is:" mcr "%";
print "Number of observations:" n_all;

names1 = {'OBS_NR' 'PURCHASE_PRICE''CORRECT CLASS'
    'PREDICTED CLASS' 'FINAL PREDICTION' "P_COMB" "P_CS" "P_RS"};

create output_bestpicktest from output_data[c=names1];
append from output_data;

QUIT;

/* Link on some of the necessary fields from the original file to do comparative analysis */
proc sql;
create table output_bestpicktest  as
select a.*,b.predval_rs,b.predval_cs,b.predval_comb,b.predval_final,b.pred_method,mod_seg
from output_bestpicktest a left join june_sys_data b
on a.obs_nr = b.obs_nr;
quit;

data output_bestpicktest;
set output_bestpicktest;
format correct_class predicted_class preds.;
length overunder $10 overun $40;

if 0.8*purchase_price <= final_prediction <= 1.2*purchase_price then ab_neural = 1;
else ab_neural = 0;

if 0.8*purchase_price <= predval_final <= 1.2*purchase_price then ab_curr = 1;
```

```
else ab_curr = 0;

predicted_class_modified = 'COMB';
if p_rs > 0.6 then predicted_class_modified = 'RS';
else if p_cs > 0.6 then predicted_class_modified = 'CS';

predicted_val_modified = predval_comb;
if p_rs > 0.6 then predicted_val_modified = predval_rs;
else if p_cs > 0.6 then predicted_val_modified = predval_cs;

if 0.95*purchase_price <= predicted_val_modified <= 1.05*purchase_price then ab_neural_05 = 1;
else ab_neural_05 = 0;
if 0.9*purchase_price <= predicted_val_modified <= 1.1*purchase_price then ab_neural_10 = 1;
else ab_neural_10 = 0;
if 0.8*purchase_price <= predicted_val_modified <= 1.2*purchase_price then ab_neural_20 = 1;
else ab_neural_20 = 0;
if 0.7*purchase_price <= predicted_val_modified <= 1.3*purchase_price then ab_neural_30 = 1;
else ab_neural_30 = 0;

if predicted_val_modified > purchase_price then overunder = 'OVER'; else overunder = 'UNDER';

if 1*purchase_price <= predicted_val_modified < 1.1*purchase_price
                                            then overun = 'OVER 0-10%';
else if 1.1*purchase_price <= predicted_val_modified < 1.2*purchase_price
                                            then overun = 'OVER 10-20%';
else if 1.2*purchase_price <= predicted_val_modified < 1.3*purchase_price
                                            then overun = 'OVER 20-30%';
else if predicted_val_modified > 1.3*purchase_price
                                            then overun = 'OVER >30%';
else if 0.9*purchase_price <= predicted_val_modified < 1*purchase_price
                                            then overun = 'UNDER 0-10%';
else if 0.8*purchase_price <= predicted_val_modified < 0.9*purchase_price
                                            then overun = 'UNDER 10-20%';
else if 0.7*purchase_price <= predicted_val_modified < 0.8*purchase_price
```

```
                                        then overun = 'UNDER 20-30%';
else if predicted_val_modified <= 0.7*purchase_price
                                        then overun = 'UNDER >30%';
run;

/* Look at the performance measures from the neural network using the  */
/* modified classification rule on the independet testing set */
proc freq data=output_bestpicktest;
tables ab_curr ab_neural ab_neural_05 ab_neural_10 ab_neural_20 ab_neural_30
overunder overun;
run;

/*************************************************************************/
```

A.5 Results from neural networks run on Lightstone application

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	20,87	39,48	66,39	81,99
2	20,57	39,21	66,86	82,50
3	20,97	39,66	67,04	82,32
4	20,59	39,41	66,63	82,38
5	20,77	39,63	67,08	82,36

Table A.2: Neural network number 1 for combining predictions.

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	20,68	39,24	66,52	82,42
2	20,96	39,70	66,90	82,26
3	21,16	39,75	66,60	82,21
4	20,76	39,38	66,61	82,10
5	20,97	39,78	66,96	82,34

Table A.3: Neural network number 2 for combining predictions

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	20,77	39,89	66,84	82,39
2	20,82	39,61	66,77	82,19
3	20,28	38,36	65,60	81,11
4	20,56	38,90	66,21	81,94
5	20,36	38,53	65,89	81,73

Table A.4: Neural network number 3 for combining predictions.

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	20,62	39,53	66,75	82,94
2	19,65	37,95	65,47	82,12
3	20,43	38,86	66,40	81,77
4	20,29	38,13	66,32	81,48
5	20,36	38,74	66,00	81,39

Table A.5: Neural network number 4 for combining predictions.

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	20,34	38,48	64,97	80,93
2	20,12	38,17	64,80	80,67
3	20,24	38,35	64,51	80,36
4	20,38	38,70	65,06	80,95
5	20,16	38,24	64,78	80,51

Table A.6: Neural network number 1 for picking optimal prediction.

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	20,61	39,11	65,58	81,14
2	20,88	39,57	66,07	81,50
3	20,69	39,27	65,66	81,01
4	20,77	39,46	65,92	81,37
5	20,78	39,45	65,90	81,37

Table A.7: Neural network number 2 for picking optimal prediction.

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	21,32	39,92	66,22	81,36
2	21,13	39,75	66,02	81,33
3	21,15	39,81	66,13	81,22
4	21,08	39,64	65,83	81,05
5	21,32	40,14	66,65	81,71

Table A.8: Neural network number 3 for picking optimal prediction.

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	20,96	39,66	66,12	81,30
2	20,91	39,53	66,16	81,40
3	20,77	39,23	66,03	81,37
4	21,30	40,06	66,31	81,40
5	20,96	39,62	66,05	81,32

Table A.9: Neural network number 4 for picking optimal prediction.

Run number	% Within 5%	% Within 10%	% Within 20%	% Within 30%
1	22,19	40,89	67,07	81,87
2	21,62	40,13	66,82	81,88
3	22,48	41,15	67,13	81,99
4	22,21	40,73	67,04	81,93
5	22,41	41,22	67,47	82,27

Table A.10: Neural network number 5 for picking optimal prediction.

Run number	% Within 5%	% Within 10%	% 20%	% Within 30%
1	21,89	40,76	67,03	81,95
2	22,01	40,83	67,01	81,79
3	22,05	40,97	67,21	82,02
4	21,90	40,82	67,09	81,90
5	21,95	40,54	66,79	81,71

Table A.11: Neural network number 6 for picking optimal prediction.