Credit Card Transactions, Fraud Detection, and Machine Learning: Modelling Time with LSTM Recurrent Neural Networks

by

Bénard Jacobus Wiese

A thesis submitted in fulfilment of the requirements for the degree of

Magister Scientiae

in the Department of Computer Science,

University of the Western Cape.

August 2007

Supervisor: Professor Christian W. Omlin

Credit Card Transactions, Fraud Detection, and Machine Learning: Modelling Time with LSTM Recurrent Neural Networks

BéNARD JACOBUS WIESE

Keywords

Credit Card
Transaction
Fraud
Support Vector Machine
Time series
Recurrence
Vanishing gradient
Long Short-Term Memory
Receiver Operating Characteristic
Pre-Processing

Abstract

In recent years, topics such as fraud detection and fraud prevention have received a lot of attention on the research front, in particular from plastic card issuers. The reason for this increase in research activity can be attributed to the huge annual financial losses incurred by card issuers due to fraudulent use of their card products. A successful strategy for dealing with fraud can quite literally mean millions of dollars in savings per year on operational costs.

Artificial neural networks have come to the front as an at least partially successful method for fraud detection. The success of neural networks in this field is, however, limited by their underlying design – a feedforward neural network is simply a static mapping of input vectors to output vectors, and as such is incapable of adapting to changing shopping profiles of legitimate card holders. Thus, fraud detection systems in use today are plagued by misclassifications and their usefulness is hampered by high false positive rates. We address this problem by proposing the use of a dynamic machine learning method in an attempt to model the time series inherent in sequences of same card transactions. We believe that, instead of looking at individual transactions; it makes more sense to look at sequences of transactions as a whole; a technique that can model time in this context will be more robust to minor shifts in legitimate shopping behaviour.

In order to form a clear basis for comparison, we did some investigative research on feature selection, pre-processing, and on the selection of performance measures; the latter will facilitate comparison of results obtained by applying machine learning methods to the biased data sets largely associated with fraud detection. We ran experiments on real world credit card transactional data using three machine learning techniques: a conventional feedforward neural network (FFNN), and two innovative

methods, the support vector machine (SVM) and the long short-term memory recurrent neural network (LSTM).

Declaration

I declare that *Credit Card Transactions, Fraud Detection, and Machine Learning: Modelling Time with LSTM Recurrent Neural Networks* is my own work, that it has not been submitted for any degree or examination in any other university, and that all the sources I have used or quoted have been indicated and acknowledged by complete references.

Bénard Jacobus Wiese
August 2007

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The aim of this chapter is to give a broad overview of the thesis. It starts off with the motivation behind the research conducted here, the premises on which the research was based, and a general problem statement. This is followed by the research hypothesis, technical goals, research methodology, and this thesis's contributions toward the fraud research community. Finally, a chapter by chapter overview of the thesis is given to form a bird's eye view of what lies ahead.

1.1. Motivation

"*Card companies continue to increase the effectiveness and sophistication of customer-profiling neural network systems that can identify at a very early stage unusual spending patterns and potentially fraudulent transactions* [6]".

There are several different factors that make card fraud research worthwhile. The most obvious advantage of having a proper fraud detection system in place is the restriction and control of potential monetary loss due to fraudulent activity. Annually, card issuers suffer huge financial losses due to card fraud and, consequently, large sums of money can be saved if successful and effective fraud detection techniques are applied. While card fraud losses against total turnover have actually declined in the past decade or so - undoubtedly due to card issuers actively fighting fraud - the total monetary loss due to card fraud has increased sharply during the same time due to an increase in the total number of cards issued and the resulting increase in card usage. Figure 1-1 shows that, during 2004 alone, an estimated £504.8 million were lost on UK-issued cards due to fraudulent activity, a staggering increase of 20% over the preceding year. In addition to this, newer fraud methods are emerging as fraud detection increases and chip cards become more widely used. These include money laundering on card transactions, identity or application fraud to obtain cards and hacking of card numbers from card processors.

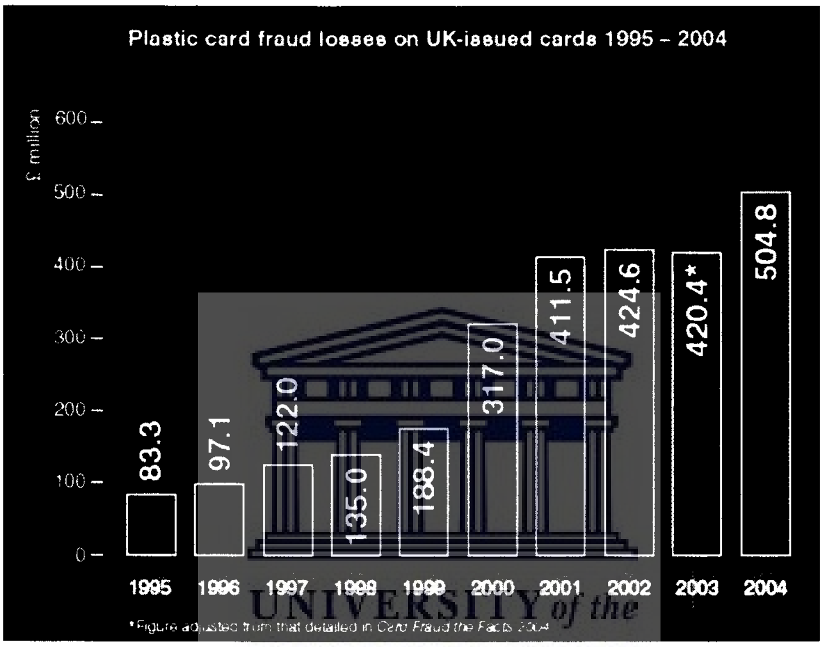

Figure 1-1: Losses due to plastic card fraud on UK-issued cards from 1995 to 2004 [6].

Ultimately, card transaction fraud detection deals with customer behaviour profiling in the sense that each card holder exhibits an ever evolving shopping pattern; it is up to the fraud detection system to detect evident deviations from these patterns. Fraud detection is therefore a dynamic pattern recognition problem as opposed to an ordinary static binary classification problem.

The data sets involved in card fraud detection are not only extremely large, but also quite complex. Fraud detection systems rely heavily on fast and complicated pre-processors to massage the data into formats compatible with machine learning algorithms. Older fraud detection systems lacked the ability to dynamically adapt to changing shopping behaviour, both in pre-processing and classification; more often than not, the large amount of data involved makes it computationally infeasible to use older systems for online transaction classification.

Because of the stigma associated with fraud, a successful fraud detection system or strategy is seen as an important advantage in the card issuing industry. Banks and card issuers are heavily engaged in research on this topic; however, the results are seldom published in the public domain which in turn only serves to hamper overall progress in card fraud research.

This thesis addresses some of the problems associated with detecting card fraud, and at least publishes comparative results on different neural network based machine learning methods. All in all, the combination of a dynamic pattern recognition problem on a complex, skewed, and immense data set makes for a very interesting research problem.

1.2. Premises

The data used in this study is real. It contains transaction sequences of real card holders generated through the use of real cards at real merchants. The data set is therefore naturally heavily biased towards the legitimate transaction class and contains relatively few fraudulent transactions - less than 1% of the data set is fraud. Since the data set is real, it also has significant noise. This has serious ramifications on fraud detection systems since these outliers have to be modelled in such a way that outliers due to noise are not mistaken for outliers due to fraud. A classifier that is incapable of making this distinction is useless because, although such a classifier might exhibit a high probability of fraud detection, it will simultaneously be plagued by a high false alarm rate. The data set used here is also quite large - more than 800 000 transactions in total, making data selection, feature selection and pre-processing all the more important.

1.3. Problem Statement

Given a sequence of transactions, can a classifier be used to model the time series inherent in the sequence to such an extent that deviations in card holder shopping behaviour can be detected regardless of the skewness and noise inherent in the data ? In addition our classifier must exhibit both a high probability of detection and a low false alarm rate during generalisation; otherwise, it will be practically useless. Since we are ultimately dealing with a dynamic problem here, the question also arises whether a dynamic neural network machine learning technique will outperform a static one. It can also be noted that the retraining of static networks to deal with changing shopping behaviour can have the effect that certain fraud patterns, that have not occurred in quite some time, cannot be detected by the retrained network when they do reoccur. A network that can learn when to forget information can be beneficial in such cases.

This study aims at answering these questions by providing a comparative analysis on the use of static feedforward neural networks, support vector machines and long short-term memory neural networks for card fraud detection.

1.4. Research Hypothesis

Our research hypothesis is that, given lists of sequential transactions to which proper feature selection and pre-processing techniques were applied, a long short-term memory recurrent neural network will outperform static machine learning methods such as feedforward neural networks and support vector machines by modelling the time series inherent in sequences pertaining to the same card as opposed to classifying individual transactions.

Our hypothesis leads to a number of research questions that we aim to answer in this thesis:

1. What features and pre-processing techniques are relevant for fraud detection ?
2. Can sequences of transactions reveal fraudulent use that remains unnoticed when looking at individual transactions ?
3. How can we model long sequences using machine learning methods ?
4. How can we deal with biased data sets ?
5. How good a performance can be achieved, and what is a good measure of performance when dealing with biased data sets ?

1.5. Technical Goals

To answer the questions resulting from our hypothesis, a number of technical goals first need to be achieved. First, we have to decide what features will be relevant for fraud detection. Second, and one of the most important steps, is to implement a pre-processor to process the features into some useable form. An insufficient pre-processor will ultimately lead to weak results. After pre-processing, a model needs to be constructed with which sequences of transactions can be encapsulated allowing the discrimination of legitimate from fraudulent transactions. Here, we have to bear in mind that these sequences are of variable length and that the data set will be necessarily heavily biased towards the legal transaction class; our model needs to account for this. Next, a measure of performance needs to be chosen that is applicable to measuring generalisation performance on biased data, which will allow comparative analysis on the performance of different classification techniques.

1.6. Methodology

First, an appropriate toolset needs to be obtained which can be used to achieve the above mentioned technical goals. This toolset should include a pre-processor, a feedforward neural network (FFNN), a long short-term memory recurrent neural network (LSTM), a support vector machine (SVM), and finally an algorithm to measure performance. Apart from the performance measure and SVM algorithm, all other tools used in this thesis will be bespoke implementations.

Since the data set is large, a more manageable subset of data has to be extracted which exhibits a predetermined class distribution - a ratio of roughly 99:1 between legitimate and fraudulent transactions. The pre-processor will be run on the resulting data set and the data will be split into equally sized training and test sets. No further processing such as noise correction or outlier removal will be done on the training or test data.

We will then run a series of experiments on the data sets using FFNN, LSTM and SVM, during which the resulting performances of each algorithm will be computed using the chosen performance measure. These results can then be analytically compared to see how the three algorithms compare to each other when applied to a non-trivial real world problem. This will either prove or disprove our original hypothesis.

1.7. Contributions

Since very few publications on card fraud detection exist in the public domain, this thesis should at least in part remedy that. It can be a reference for a complete overview of card fraud detection with neural network based machine learning methods. Results of research in other domains show that LSTM neural networks promise good results when applied to problems with an intrinsic dynamic nature; to our knowledge, this is the first time LSTM was applied in the card fraud detection domain.

Our results will show that it is worthwhile investigating time series modelling for fraud detection and that it might be a solution to the high false alarm rates that plaque similar systems today.

1.8. Overview

This section gives a short overview of the remainder of this thesis. In Chapter 2, the definition of card fraud and the problem of fraud detection is introduced and discussed. Chapter 3 contains a brief literature review of publications relevant to the thesis topic. Chapter 4 is a short introduction to neural networks, and sets the scene for the introduction of the two main methodologies used, namely support vector machines in Chapter 5 followed by recurrent neural networks (and more specifically long short-term memory) in Chapter 6. The performance measure used in this thesis is introduced in Chapter 7, along with a discussion on why we believe it is the most appropriate measure. Chapter 8 deals with all data related topics such as feature selection and pre-processing. Experiments with FFNN, SVM, and LSTM are conducted and results recorded and discussed in Chapter 9. Finally, our research conclusions are presented in Chapter 10 through a critical discussion of the experimental results, followed by a discussion of possible future research directions.

Chapter 2

FRAUD AND FRAUD DETECTION

2.1. Introduction

This chapter gives a short overview of fraud and its detection in the context of financial transactions initiated with the use of credit or charge cards. The first subsection starts with a definition of fraud followed by its discussion in the context of the card issuer industry. The second subsection deals with the detection of fraud and the importance of a successful fraud detection strategy.

2.2. Fraud

In the context of the card issuer industry, fraud can be defined as the actions undertaken by undesired elements to reap undeserved rewards, resulting in a direct monetary loss to the financial services industry. Here, we deal with the attempts by fraudsters to use stolen credit card and identity information to embezzle money, goods or services.

With card issuers constantly aiming to expand their operations by launching aggressive campaigns to gain bigger portions of the market share, the use of technology has become increasingly more prevalent in making it easier for people to transact and spend. This increase in ease of spending through the use of technology has, unfortunately, also provided a platform for increases in fraudulent activity. Fraud levels have consequently sharply risen since the 1990's, and the increase in credit card fraud is costing the card issuer industry literally billions of dollars annually. This has prompted the industry to come up with progressively more effective mechanisms to combat credit card fraud. A machine learner that encapsulates expert systems is an example of such a mechanism.

More recently, the issuing industry has taken a stance to prevent fraud rather than to put mechanisms in place to minimise its effects once it takes place, and major markets have therefore taken considerable steps towards becoming EMV (Europay-Mastercard-Visa) enabled. The idea behind EMV is to use chip cards and personal identification numbers (PIN) at point of sale devices rather than authorising transactions through the use of magnetic stripes and card holder signatures. Magstriped cards have the weakness that magnetic stripes can be easily copied and reprinted on fake cards - called card skimming - and card issuers believe that chip cards, being difficult to replicate, will limit loss due to card skimming. The question now is whether the necessity of fraud detection in the card issuing industry still exists. To answer this, one has to look at the effect that EMV enablement might have on fraud patterns globally.

With the shift to EMV, fraud liability will shift from the card issuers to non EMV-compliant merchants. With the onus on service establishments to ensure proper use of credit cards in their shops, a shift in fraud patterns is likely to occur. Chip and PIN, however, will by no means spell the end of credit card fraud. It is expected that "card-not-present" fraud will increase significantly because of chip and PIN. Card-not-present transactions take place when the physical card and card holder signature do not form part of the authorisation process, such as telephone and online purchases. Most major banks also expect ATM fraud to increase because of PIN exchange and handling in insecure environments. Card skim fraud, on the other hand, will probably migrate into countries which have not opted for EMV, such as the USA that shares borders with markets which are already EMV enabled, i.e. Canada and Mexico.

Fallback fraud is reportedly already on the increase in EMV enabled markets. Fallback happens when the chip on an EMV card is damaged and systems have to fall back on magstripe in an attempt to authorise the transaction. Some people claim that up to 45% of ATM transactions in EMV enabled markets have to fall back on magstripe, while other banks report absurd fallback figures of close to 100% in some cases. Fallback fraud has now become such a major problem to the extent that a certain global card issuer has indicated that they are considering banning fallback transactions entirely worldwide. These problems are obviously due to the relative immature state of EMV as it currently stands and will probably be solved in due time; however, any expectation that this type of fraud prevention will be enough to curb credit card fraud is overly optimistic, to say the least.

One cannot underestimate the determination of the fraudster; where there is a will, there is a way. Credit card fraud detection is therefore, at least for now, likely to stay.

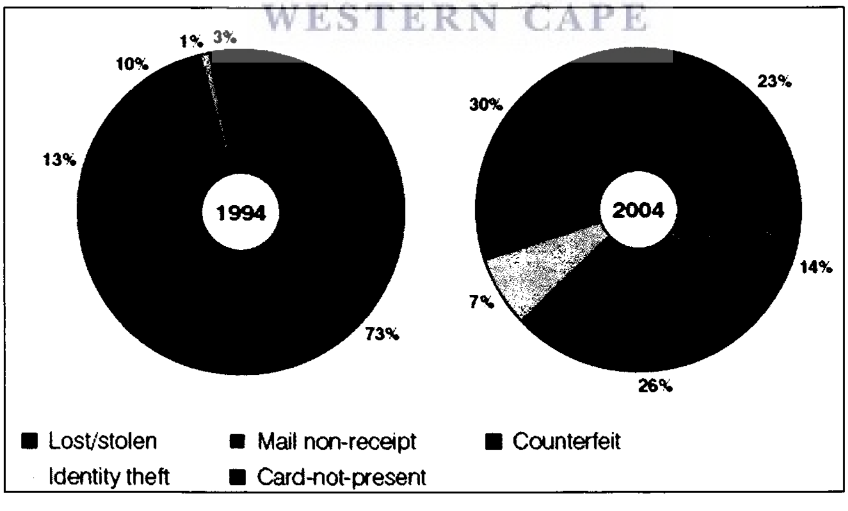

Figure 2-1: Migration of fraud patterns (fraud types) between 1994 and 2004 [6].

2.3. Fraud Detection

So, exactly what does fraud detection entail ? Quite simply put, fraud detection is the act of identifying fraudulent behaviour as soon as it occurs [2], which differs from fraud prevention where methods are deployed to make it increasingly more difficult for people to commit fraud in the first place. One would think that the principle of *prevention is better than cure* would also prevail here; but as discussed in the previous subsection, prevention is not always effective enough to curb the high fraud rate that plagues the credit card industry thus motivating the deployment of fraud detection mechanisms. As processing power increases, fraud detection itself might even become a prevention strategy in the future.

The fact that credit cards are used in uncontrolled environments, and because legitimate card holders may only realise that they have been taken advantage of weeks after the actual fraud event, makes credit cards an easy and preferred target for fraud. A lot of money can be stolen in a very short time, leaving virtually no trace of the fraudster. Detecting fraud as soon as possible after it occurs is therefore important to give the card issuer a chance to at least limit the damage. As soon as a transaction is flagged as a possible fraud, the card holder can be phoned to establish whether the transaction was legitimate or not and the card blocked if necessary. Fraud investigation and the chargeback process are costly and put a lot of strain on resources.

The quicker fraud can be detected the better; but the large amount of data involved - sometimes thousands of transactions per second - makes real-time detection difficult and sometimes even infeasible. Many banks attempt to detect fraud as soon as possible after it happened, as opposed to detecting it in realtime, because fraud detection can slow down an authorisation request to such an extent that it times out.

The development of new fraud detection systems is hampered by the fact that the exchange of ideas pertaining to this subject is severely limited, simply because it does not make sense to describe fraud detection and prevention techniques in great detail in the public domain [2]. Making details of fraud detection techniques public will give fraudsters exactly what they need to devise strategies to beat these systems. Fraud, especially in the context of the financial services industry, is seen as a very sensitive topic because of the stigma attached to potential monetary loss. Card issuers are therefore usually very hesitant to report annual fraud figures and better fraud detection strategies or systems than those of the competition are therefore advantageous; this gives even more reason for issuers to keep internal research results on fraud detection away from the public domain.

Banks add to the problem because their local banking authorities or local cultures make them reluctant to issue fraud figures on the premise that this will scare their customers into believing that banking systems are unsafe and not secure. Data sets and results are therefore rarely made public, making it difficult to assess the effectiveness of new fraud detection strategies and techniques.

Most fraud detection systems today are awkward to use and become ineffective with time because of changing shopping behaviour and migrating fraud patterns. During the holiday seasons, for example, shopping profiles change significantly and fraud detection systems cannot deal with changing behaviour because of their static nature and inability to dynamically adapt to changes in patterns. Thus, fraud detection systems suffer from unacceptable false alarm rates, making the probability of annoying legitimate customers much higher than that of actually detecting fraud. Systems based on Hidden Markov Models promise better results, but are useless for real-time use when transaction volumes become too high. In this thesis, we will investigate the use of a technique that is neural network based and therefore promises relative quick classification times, and also dynamic because it attempts to learn the underlying time series present in series of same card holder transactions.

Chapter 3

LITERATURE REVIEW

Published articles dealing with the method of and results on credit card fraud are few and far in between. Compared to the volume of publicly available research papers on fraud in other areas, such as telecommunications fraud and computer intrusion detection for instance, the number of papers available on card fraud is almost negligible. The possible reasons for this were already discussed in Chapter 2. The little available literature on this topic, however, forms the basis for a good starting point into credit card fraud research.

Bolton and Hand [2] gives a good overview on the fraud problem in general and the difficulties faced with fraud detection, including sections specifically dealing with credit card fraud. Brause *et al* [4] takes a hybrid approach by applying a combined rule-based and neural network system to credit card fraud detection. Their paper also gives a good description of what the goal of fraud detection should be and includes a section on how to possibly model the data generated by card transactions. Maes *et al* [21] analyse the card fraud problem by comparing results obtained from Bayesian belief networks with that of artificial neural networks. They also offer a thorough discussion on some of the problems associated with credit card fraud detection. Card fraud statistics are available from websites such as that of the Association for Payment Clearing Services (APACS). It remains difficult to analytically compare results from the different techniques used in the above mentioned papers because they use different data sets and mostly only publish summary statistics as opposed to actual results.

One of the most important steps in building a successful classifier for fraud detection is ensuring that the data is properly pre-processed after feature selection. Equally important is selecting an appropriate technique for measuring a classifier's performance during generalisation testing. While feature selection greatly depends on the whim of each researcher, some publications do offer rough guidelines on data preparation and performance measurement. Masters [22] gives a detailed description of data input preparation through discussions on different variable types, scaling and transformations, while Mena [23] gives some indications into data selection and preparation methodology and a case study of a real-time fraud detection system. As usual, Press, Teukolsky *et al* [25] contains helpful information on mean and standard deviation calculations which forms the basis of many a pre-processing technique.

Most of the papers on fraud mentioned above have sections dealing with the problem of performance measurement on skewed and biased data sets. Sections specific to the receiver operating characteristic can be found in Masters [22] and Harvey [13].

Literature on support vector machines is quite common. Vapnik *et al* [3] discuss the theory and practicalities behind the optimal margin classifier for separable training data - a discussion which is extended to the non-separable case by Cortes and Vapnik [9]. Kroon and Omlin [19] gives a good overview of support vector machine theory and application, including discussions on the kernel trick used to generalise the SVM to high-dimensional feature space in an attempt to transform non-separable data into separable data. Their paper on SVM application [18] is also useful to get a jump start in using the popular libSVM [8] toolkit. Burges [5] gives a very detailed and complete tutorial on support vector machines used for pattern recognition, and gives a more in depth account of the theory involved. The work of Keerthi and Lin [17] is useful for investigating what happens when an SVM displays tendencies towards asymptotic behaviour, for example when the separating hyper planes take on very large or small values. This helps in understanding how to prevent an SVM from overfitting or underfitting the training data. Lee *et al* [20] tackles the task of extending SVMs to multi-category problems, i.e. problems where the data set can be divided into more than two classes. Hsu *et al* [15] gives important information on successfully using SVMs in practice. They also discuss the cross-validation technique and propose a grid-search method for obtaining the most appropriate kernel parameters for a new classification problem.

Publications on feedforward and recurrent neural networks are also widespread. Mitchell [24] provides a good overview of the theory behind neural networks, showing how the normal perceptron of Acton [1] can be extended into a network to ultimately handle non-separable data. Robinson and Fallside [26] show how to adapt static neural networks to deal with dynamic patterns, i.e. recurrent neural networks. Williams and Zipser [27] give a more complete overview of recurrent neural networks including in-depth discussions on the two most common recurrent neural network training algorithms: backpropagation through time and real-time recurrent learning. Hochreiter and Schmidhuber [14] discuss the vanishing gradient problem which limits a recurrent neural network's ability to learn long-term dependencies in patterns, and propose a novel technique to address this problem: the long short-term memory recurrent neural network (LSTM). Gers, Schmidhuber *et al* [11] and Gers, Schraudolph *et al* [12] extend the LSTM network's capabilities by introducing forget gates and peephole connections, respectively.

Chapter 4

PERCEPTRONS AND NEURAL NETWORKS: A BRIEF OVERVIEW

4.1. Introduction

The artificial neural network machine learning method has always been applied with great success to a wide variety of learning problems. Artificial neural networks represent one of the more common approaches to implementing humanlike perception, thought and behaviour. They are called artificial neural networks because their general topology is based on the densely interconnected network of neurons that make up the human brain, and like human brain activity as explained by neurobiology, an artificial neural network functions by exciting certain chains of neurons depending on the inputs it receives.

This chapter is not intended to give an in-depth discussion on the workings of feedforward neural networks but rather a brief overview of neural networks at the hand of examining the perceptron and its use in multi-layer networks. This is done because both machine learning methodologies used later in this thesis are derived directly from the perceptron.

4.2. The Perceptron

The perceptron was the first attempt by Rosenblatt (1958) to model a neural network [22] and forms the basic building block of the more advanced multi-layer neural network. In its simplest form, the perceptron is a structure consisting of three layers:

1. an input layer for presenting input to the perceptron,
2. an association unit layer that acts as a feature detector, and
3. an output response layer.

A perceptron's task is quite simple: to produce an output by applying a threshold to the weighted sum of its inputs. Thus, it takes a real-valued input vector, calculates a linear combination of these inputs and then outputs a 1 if the result is greater than some threshold, or -1 otherwise [24]. Figure 4-1 illustrates the general structure of a perceptron.

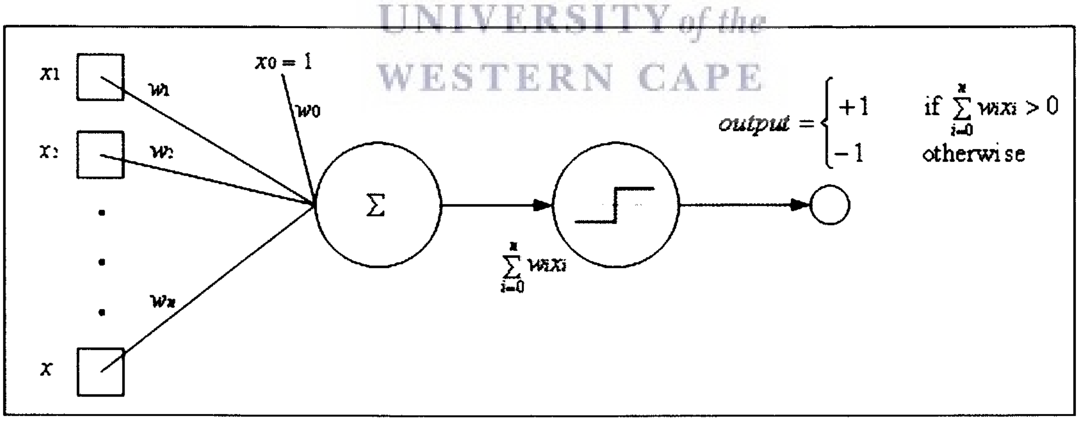
x1
w1
x0 = 1
w0
x2
w2
wn
x
Σ
$\sum_{i=0}^{n} w_i x_i$
$output = \begin{cases} +1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$

Figure 4-1: The general structure of a three-layer perceptron with its input, association, and output response layer (illustration adapted from [24]).

In the above figure the symbols x_1 to x_n represent the components of a real-valued input vector, while w_1 to w_n represent the weights associated with each of the n dimensions of the input vector. Weight w_0 is associated with a constant input (x_0) of value 1. The quantity (-w_0) is therefore a bias which the weighted sum of the input vector components must exceed in order for the perceptron to achieve activation. A perceptron's output is therefore computed using Equation (1).

$$o = \begin{cases} +1 & w_1x_1 + w_2x_2 + \ldots + w_nx_n > -w_0 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

4.2.1. Perceptron Training

So how does a perceptron learn to classify a set of inputs ? Since the values of the input variables x_1 to x_n are the input data, and the output of the perceptron depends on Equation (1), the only variables left that can be updated during perceptron training are the weights. The training process consists of iteratively applying the perceptron to each training example in the training set, and then updating the weight vector whenever a training example is misclassified. Small random values are used to first initialise the weights before training begins. Weights are updated using the perceptron training rule [24] which updates weight w_i associated with input x_i according to the rules in Equation (2) and (3).

$$w_i \leftarrow w_i + \Delta w_i \tag{2}$$

$$\Delta w_i = \eta(d - o)x_i \tag{3}$$

In Equation (3) the symbol d represents the target class or desired output of the training example while o represents the output generated by the perceptron after feeding it with the training example. The learning rate is represented by η and is normally set to a small, positive value which can be forced to decay as training progresses in order to limit the of effects local minima on training.

4.3. Linear Separability

The perceptron and its training rule as described in the previous section have one serious shortcoming: a perceptron can only be used for classification problems with linearly separable data; linear separability guarantees convergence of the training process.

So what then is linear separability ? A classification problem is linearly separable if the positive and negative exemplars in its data set can be completely separated with a hyperplane in *n*-dimensional space, where *n* is one less than the number of inputs to the perceptron, given that the constant unity bias input is counted as one of the dimensions. Consider the illustrations in Figure 4-2 which shows the decision surfaces of a two-input (x_0 and x_1) perceptron applied to two different classification problems. Example (a) is linearly separable since the positive target class (+) can be separated from the negative target class (-) by means of a straight line in two-dimensional space. Example (b) shows the decision surface of the infamous XOR problem. No straight line in two-dimensional space can separate its positive class from its negative class, and therefore the XOR problem is not linearly separable. A single perceptron will not be able to solve the XOR problem and neither will the perceptron training rule be guaranteed to converge to an overall solution.

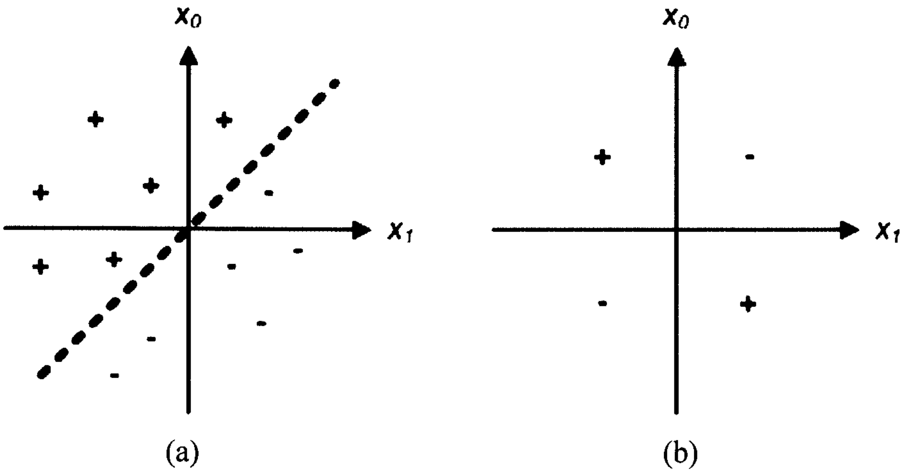

Figure 4-2: Linear separability. (a) A linearly separable problem, i.e. the binary class can be separated by a straight line in two dimensional space. (b) A representation of the XOR boolean function, which is not linearly separable.

4.4. Gradient Descent

A second training rule that was designed for linearly inseparable data sets is the delta rule. In cases where the data is inseparable, the delta rule uses gradient descent to converge to a best-fit approximation of the target function [24]. The delta rule and gradient descent forms the basis for the most common training algorithm for multi-layer neural networks, the backpropagation algorithm. Multi-layer networks are formed by organising multiple perceptrons into several layers to form a forward connected network. The most common networks usually consist of an input layer, a hidden layer and an output layer. A lot of literature is available on multi-layer networks and backpropagation and we will therefore not go into more detail on these two topics here.

4.5. Summary

This chapter briefly touched on the perceptron and its use in multi-layer neural networks. The perceptron forms the basis for more advanced network based machine learning methods, including support vector machines and recurrent neural networks. These two novel learning methods are still relatively new compared to the perceptron, but, at the same time, both these methods are propositioned to deliver extremely powerful classifiers by addressing some of the fundamental problems associated with multi-layer neural network training. The following two chapters deal with support vector machines and recurrent neural networks in detail.

Chapter 5

METHODOLOGY I: SUPPORT VECTOR MACHINES

5.1. Introduction

In 1992, Boser, Guyon and Vapnik proposed a training algorithm for optimal margin classifiers in which they showed that maximising the margin between training examples and class boundary amounts to minimising the maximum loss with regards to the generalisation performance of the classifier [3]. This idea was initially explored for two reasons:

1. binary class optimal margin classifiers achieve errorless separation of the training data, given that separation is possible;
2. outliers are easily identified by these types of classifiers and therefore do not affect the generalisation performance as is mostly the case in other classifiers, where performances are based on minimising the mean squared or other type of average error.

The first investigations into this type of algorithm were based on separable data sets, but, in 1995, Cortes and Vapnik extended the algorithm to account for linearly inseparable data [9]; attempts soon followed to also extend the results to multi-class classification problems. This type of learning machine was later dubbed the support vector machine (SVM).

As can be seen in this chapter, the SVM is a machine learning technique with a strong and sound theoretical basis. It is interesting to note that, in most cases, researchers claim that SVMs match or outperform neural networks in classification problems. In this thesis we will put these claims to the test on a non-trivial, real world problem.

The introduction to SVMs starts in Section 5.2 by first investigating the notion of an optimum margin as explained at the hand of the maximum margin hyperplane. Section 5.3 deals with direct and dual space representations of an arbitrary hyperplane while Section 5.4 introduces the concept of soft margins to handle linear inseparability. The nonlinear SVM is discussed in Section 5.5, and in conclusion a brief summary is presented in Section 5.6.

For the sake of simplicity and because the practical classification problem this thesis addresses is dichotomous in nature, we will limit our discussion to binary classification problems.

5.2. The Maximum Margin Hyperplane

During the supervised training process of a classifier, training vectors or patterns of the form $(\mathbf{x},y)$, where $\mathbf{x}$ is a set of input parameters or features and y denotes class membership, are repeatedly presented to the classifier in an attempt to learn a decision function $D(\mathbf{x})$, which can later be used to make classification decisions on previously unseen data. In the case of the optimal margin training algorithm, these decision functions have to be linear in their parameters, but are not restricted to linear dependencies in their input components x, and can also be expressed in either direct or dual space [3].

In direct space, the decision function has the following form:

$$D(\mathbf{x}) = \sum_{i=1}^{N} w_i \varphi_i(\mathbf{x}) + b \qquad (4)$$

This is identical to the perceptron decision function as discussed in Section 4.2 and shown in Equation (1), with the bias represented by b instead of w_0 and φ_i some function of $\mathbf{x}$. In Equation (1), φ_i is simply the identity function.

In the case of a binary outcome decision function applied to linearly separable data, the function can be represented in 2-dimensional space as a straight line splitting the input vectors into the two classes they might possibly belong to, as demonstrated in Figure 5-1.

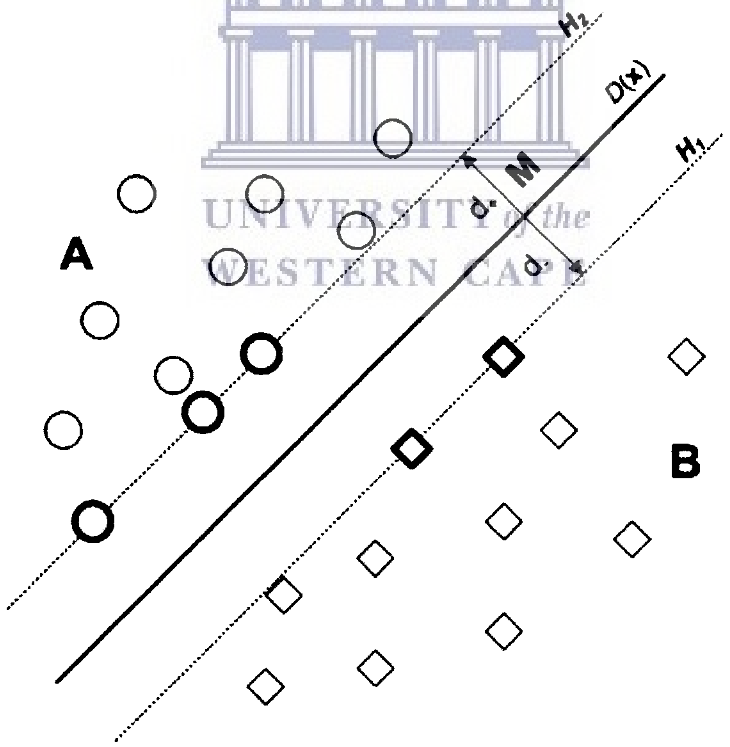

Figure 5-1: A binary outcome decision function in 2-dimensional space (adapted from [5]). The support vectors are shown with extra thick borders, with *H1* and *H2* denoting the separating hyperplanes on which they lie. The maximum margin is denoted as **M** and gives the maximum distance between the separating hyperplanes.

In its simplest form, the SVM algorithm will construct a hyperplane that completely separates (at least in the linear separable case) the data in such a way that $\mathbf{w}\cdot\mathbf{x} + b > 0$ for all points belonging to one class, and $\mathbf{w}\cdot\mathbf{x} + b < 0$ for all points in the other class [19]. In other words, for $D(\mathbf{x}) > 0$ pattern $\mathbf{x}$ belongs to class A and for $D(\mathbf{x}) < 0$ pattern $\mathbf{x}$ belongs to class B. In this context, $D(\mathbf{x})$ is referred to as the decision surface or separating hyperplane and all points $\mathbf{x}$ which lie on this decision surface satisfy the equation $\mathbf{w}\cdot\mathbf{x} + b = 0$.

Consider that the two possible classes to which an arbitrary pattern can belong are identified by labels y_i, where $y_i \in \{-1,1\}$. Furthermore, define $d_+(d_-)$ to be the shortest distance between the separating hyperplane and the closest positive (negative) pattern in the training set. We can then define the margin (denoted M in Figure 5-1) to be (d_+ + d-), a quantity that the support vector algorithm will attempt to maximise during training. In the linear separable case, all training data will adhere to the following two constraints:

$$\mathbf{w}\cdot\mathbf{x}_i + b \geq +1 \quad (\text{for } y_i = +1) \tag{5}$$

$$\mathbf{w}\cdot\mathbf{x}_i + b \leq -1 \quad (\text{for } y_i = -1) \tag{6}$$

A vector $\mathbf{w}$ and scalar b therefore exist such that

$$y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1 \geq 0, \; i \in \{1,\ldots,l\} \tag{7}$$

The points for which the equalities in Equations (5) and (6) hold lie on two separate hyperplanes parallel to the separating hyperplane, but on different sides as shown by H_1 and H_2 in Figure 5-1. These hyperplanes can be defined as H_1: $\mathbf{w}\cdot\mathbf{x}_i + b = 1$ and H_2: $\mathbf{w}\cdot\mathbf{x}_i + b = -1$, where $\mathbf{w}$ is normal to the hyperplanes in both cases. The perpendicular distance between H_1 and the origin and H_2 and the origin is therefore

$\frac{|1-b|}{\|\mathbf{w}\|}$ and $\frac{|-1-b|}{\|\mathbf{w}\|}$, respectively, and it then follows from this that $d_+ = d_- = \frac{1}{\|\mathbf{w}\|}$ and margin $M = \frac{2}{\|\mathbf{w}\|}$.

A support vector can now be defined as a training pattern which lies on either H_1 or H_2, and whose removal might change the size of the maximum margin. In Figure 5-1, the support vectors are shown as circles or squares with extra thick borders. We can find the set of hyperplanes H_1 and H_2 that gives the maximum margin by maximising the quantity $\frac{1}{\|\mathbf{w}\|}$ (or likewise minimising $\frac{1}{2}\|\mathbf{w}\|^2$). This is equivalent to solving the quadratic problem

$$\min_{\mathbf{w}} \|\mathbf{w}\|^2 \qquad (8)$$

under the constraints in (7).

Although the quadratic problem in (8) can be solved directly with numerical techniques, this approach becomes impractical when the dimensionality of the φ-space becomes large; furthermore, no information about the support vectors is gained when solving the problem in direct space [3]. These problems are addressed by transforming the problem in (8) from direct space into dual space.

5.3. Direct Space versus Dual Space

We use a Lagrangian formulation to transform the problem in Equation (8) from direct space into dual space. There are two main reasons for doing this:

1. The constraints in (7) are replaced by constraints on the Lagrangian multipliers themselves, which are a lot easier to handle;
2. In the Lagrangian formulation, the training exemplars will appear in the form of dot products between vectors, which allow us to generalise the training and test procedures to the nonlinear case.

In order to obtain the primal Lagrangian, we introduce a vector of Lagrange multipliers $\alpha = (\alpha_1, ..., \alpha_l) \geq 0$ to handle the constraints given in (7). The constraint equations are multiplied by these positive Lagrange multipliers and subtracted from the objective function to form the primal Lagrangian:

$$L_P \equiv \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{l} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{w} + b) + \sum_{i=1}^{l} \alpha_i \tag{9}$$

It follows from optimisation theory that the solution to the problem in (8) lies at the saddle point of the Lagrangian function L_P. To obtain the saddle point, we need to minimise L_P with respect to **w** and b and at the same time ensure that the derivatives of L_P with respect to all theα_i's vanish, subject to$\alpha_i \geq 0$. The factor 1/2 in (9) is included for cosmetic reasons since we will be taking derivates. We minimise L_P by taking partial derivates with respect to **w** and b and setting them to zero to ensure that the gradient vanishes:

$$\frac{\partial L_P(\mathbf{w}, b, \alpha)}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i = 0 \tag{10}$$

$$\frac{\partial L_P(\mathbf{w}, b, \alpha)}{\partial b} = -\sum_{i=1}^{l} \alpha_i y_i = 0 \tag{11}$$

Substituting Equations (10) and (11) back into the primal Lagrangian yields the dual Lagrangian (denoted by subscript *D* as opposed to *P* in Equation (9)):

$$L_D = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \qquad (12)$$

The Lagrangians in Equations (9) and (12) arise from the same objective function but under different constraints, and the solution is found by either minimising L_P or maximising L_D. The formulation of the problem as given in (12) is called the *Wolfe dual* problem. It can be seen in L_D that there is a Lagrange multiplier α_i for every training point $\mathbf{x}_i$. The training points for which $\alpha_i \geq 0$ satisfy $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0$. These points lie on either H_1 or H_2 and are called the support vectors of the solution. All other training points have $\alpha_i = 0$ and lie either on one of the hyperplanes (H_1, H_2) such that the equality in (7) holds, or on that side of H_1 or H_2 such that the strict inequality in (7) holds [5].

L_D is a quadratic form with all constraints linear in the α_i's and thus solving L_D is a standard quadratic programming problem solvable by numerous computing packages. Converting from direct space to dual space is analogous to changing the problem into an optimisation problem where the constraints are considerably easier to handle. Notice that by finding a solution for L_D, we can also find the $\mathbf{w}_i$'s by using equation (10). The only unknown parameter now remaining from the original optimisation problem is the threshold quantity *b*.

In order to calculate *b* we need to use the Karush-Kuhn-Tucker (KKT) complementarity conditions that play an important role in both the theory and practice of constrained optimisation. For the problem at hand, the relevant KKT complementarity conditions are [19]

$$\alpha_i \{y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1\} = 0, \quad \mathrm{i} \in \{1,\ldots,1\} \qquad (13)$$

For a complete list of all KKT conditions for this particular problem the reader can consult [5]. In the context of the SVM training algorithm, the KKT conditions are necessary and sufficient for **w**, b and α to be a solution.

Now let us define an α^0 which maximises L_D subject to its constraints, and parameters w^0 and b^0 as the parameters of the corresponding optimal hyperplane. It follows from (10) that $w^0 = \sum_{i=1}^{l} \alpha_i^0 y_i x_i$ [19]. In this case, α^0 will be nonzero and using the KKT complementary conditions in (13) we can therefore have

$$y_i(w^0 \cdot \mathbf{x}_i + b^0) = 1$$

so that

$$b^0 = y_i - w^0 \cdot \mathbf{x}_i \tag{14}$$

since $y_i = \pm 1 = \frac{1}{y_i}$. In order to obtain a more representative and numerically stable value for b^0, it is usually taken to be the mean of the values resulting from calculating Equation (14) for all $i \in (1,\ldots,l)$.

Once all the above mentioned values have been calculated, we can use the newly calculated separating hyperplane to classify a new training example, $\mathbf{x}_n$, by calculating on what side of the hyperplane it lies by using

$$\mathrm{sgn}(w^0 \cdot \mathbf{x}_n + b^0)$$

5.4. The Soft Margin Hyperplane

The main assumption under which all of the equations in the previous sections are derived is that the training data is linearly separable. This is, however, a severe oversimplification of real world data sets and the SVM algorithm will have little to no practical value if not extended to handle linearly inseparable problems. In feedforward neural networks, inseparable training sets with high dimensionality are normally sufficiently handled by minimising some training error in the context of some predefined error measure. The same principle can be applied to SVMs by introducing positive variables $\xi_i \geq 0,\ i = 1,\dots,l$ so that we have the following constraints

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i,\ i \in \{1,\dots,l\} \tag{15}$$

$$\xi \geq 0,\ \forall i \tag{16}$$

The positive variables ξ_i in (15) and (16) are called slack variables; they allow for some margin of error (i.e. slack) when deciding on which side of the optimal hyperplane a training exemplar lies. In fact, ξ_i must exceed unity for an error to occur and $\sum_i \xi_i$ represents an upper bound on the number of training errors [9][5]. The introduction of slack variables relaxes the original constraints in (7) somewhat. During training, we would like the separation error (and hence the ξ_i's) to be as small as possible and we therefore introduce a penalty on the objective function by changing it to

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l}\xi_i \tag{17}$$

where $C \in \Re$ is a cost parameter that represents the cost of making an error, or the extent to which violations of the original constraints (7) should be penalised [19]. Figure 5-2 depicts the use of slack variables.

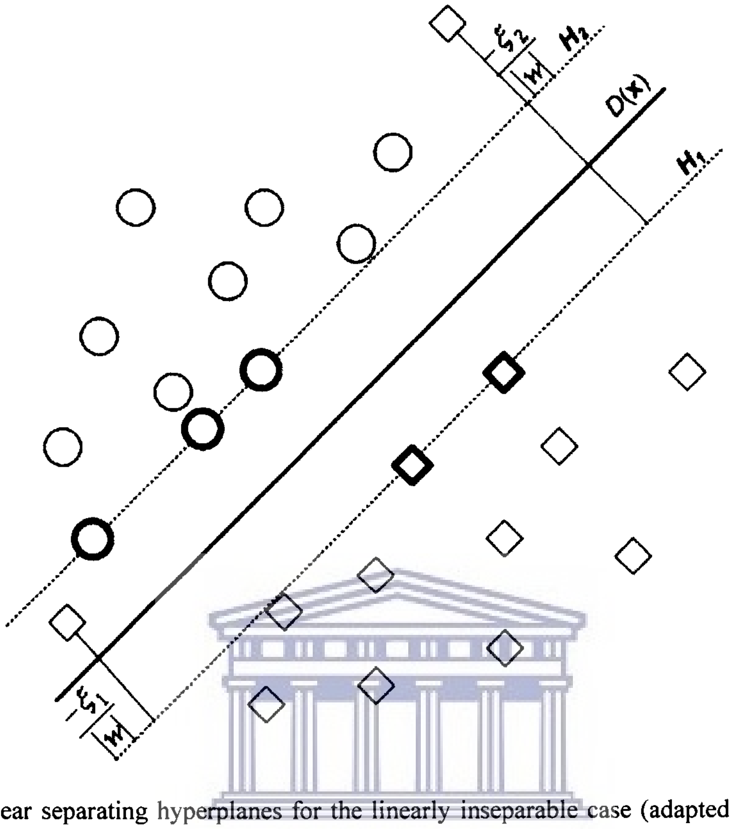

Figure 5-2: Linear separating hyperplanes for the linearly inseparable case (adapted from [5]). Data vectors on the wrong side of the decision surface are compensated for by the introduction of slack variables, whose magnitude need to exceed unity for an error to occur.

There is no straightforward method for selecting the value of the cost parameter C, but a number of techniques exist; the most popular method being grid-search [15] using cross-validation [9].

Equation (17) once again leads to a quadratic programming problem. An additional feature of the formulation in (17) is that the slack variables and their Lagrangians vanish in the Wolfe dual formulation [5]. In order to derive the quadratic programming problem, we once again introduce the constraints $\alpha \geq 0$, and the Lagrange multipliers $\eta = (\eta_1, \ldots, \eta_l) \geq 0$ to enforce the constraint $\xi \geq 0$. The primal Lagrangian in this case is

$$L_P = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i - \sum_i \alpha_i \{y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 + \xi_i\} - \sum_i \eta_i \xi_i \qquad (18)$$

Following the same path as in Section 5.3, we take partial derivates with respect to *w*, *b* and ξ_i, and setting them to zero we obtain the same two constraints as with the original optimal margin SVM,

$$\mathbf{w} = \sum_{i=1}^{l} \alpha_i y_i \mathbf{x}_i \tag{19}$$

and

$$\sum_{i=1}^{l} \alpha_i y_i = 0 \tag{20}$$

along with a new constraint

$$C = \alpha_i + \eta_i \tag{21}$$

Using the constraints in (19), (20) and (21), we derive the Wolfe dual

$$L_D = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \tag{22}$$

Equation (22) shows that the introduction of the slack variables has absolutely no effect on the dual objective function. The slack variables, however, change the constraints somewhat, which now becomes

$$0 \leq \alpha_i \leq C \tag{23}$$

and

$$\sum_i \alpha_i y_i = 0 \tag{24}$$

The difference between (23) and the original constraint on α is that the α_i's now also have an upper bound *C*. The introduction of slack variables therefore limits the search space of α [19]. As in Section 5.3, the KKT conditions for the primal problem can be used to obtain a value for the threshold *b*. A complete list of the KKT conditions for the primal Lagrangian of the linear inseparable problem can be found in [5].

5.5. Nonlinear Support Vector Machines

Another method for dealing with SVMs with decision functions that are nonlinear functions of their input vectors, and consequently deal with inseparable input data, is to transform the inseparable data in input space to a higher-dimensional feature space. The reason behind this transformation is based on Cover's theorem, which states that in cases where data is inseparable in a low-dimensional space, transformation of the data to a higher-dimensional feature space often yields linear separability [19].

The transformation of data points to a higher-dimensional feature space can yield an optimisation problem that is computationally infeasible to solve. Luckily, a technique called the kernel trick exists that can be used in such cases. The kernel trick allows us to solve the optimisation problem (i.e. calculate b and specify the hyperplanes in feature space to obtain a decision function), without ever having to directly transform the data points to feature space. Those that are interested in the details of the kernel trick can consult [19] which gives a good overview of the technique, the theorems it is based on, and its application to SVMs.

5.6. Summary

The aim of this chapter was to give a concise overview of the support vector machine classifier method. It is a relatively simple technique with a strong theoretical and mathematical basis, and most researchers claim that it exhibits above average performance when compared to neural network methods [3][5][9]. Some of the SVM's strengths can be summarised as follows:

1. it always achieves an errorless (complete) separation of the training data, given that the training set is linearly separable;
2. outliers or meaningless patterns are easily identified and eliminated;
3. the solution found by the SVM training algorithm is always a global minimum, unlike most neural network techniques that are plagued by problems with local minima.

On the other hand, the SVM algorithm does exhibit a number of weaknesses:

1. the SVM algorithm is not easily extendable to multi-class problems, which is still in large part an unsolved problem - especially in the context of the computational burden that a multi-class approach introduces [20];
2. the SVM has severe limitations in speed and size in both training and test phases [5]. Training on large datasets with millions of support vectors is also still an unsolved problem;
3. a lot has been written about the choice of the kernel function, and most researchers see it as a huge limitation since once it is chosen, only the error penalty *C* is left as a user-definable parameter.

In this thesis we will empirically compare the SVM to another novel machine learning technique, the long short-term memory recurrent neural network, discussed in the next chapter.

Chapter 6

METHODOLOGY II: RECURRENT NEURAL NETWORKS

6.1. Introduction

While feedforward neural networks can solve a vast range of classification problems with relative ease, they can fail miserably when applied to problems with an underlying temporal nature. This is quite simply because they represent a static mapping of an input to an output vector and can therefore not learn to represent the dynamically changing states that are necessary to successfully model temporal problems. A good example of a problem with an underlying temporal nature is fraud detection, where it helps to take the past shopping behaviour of a card holder into account when deciding whether or not a new transaction belongs to the fraud domain. These problems are sometimes also referred to as sequence classification problems, where the input to a network is a sequence of feature vectors and the desired output is the correct classification [27].

Sequence classification problems also introduce the notion of time, where the presentation to the network of each input vector of a particular sequence takes place during separate and distinct time steps. It is this underlying time or temporal structure in sequence classification problems that necessitates the replacement of the static feedforward neural network with a classification tool that exhibits nonlinear dynamical behaviour. The recurrent neural network is such a tool.

6.2. Recurrent Neural Networks

As mentioned in the previous section, a recurrent neural network (RNN) is a classification tool that exhibits highly nonlinear dynamical behaviour and has some internal state at each time step of a classification. RNNs can be divided into three different architecture types, two of which are partially recurrent neural networks called Jordan or Elman RNNs, and the third is the fully recurrent neural network. Figure 6-1 (a) shows the Jordan RNN, proposed by Jordan in 1986 [16], where the output layer feeds back into a type of input element called a context unit. Figure 6-1 (b) shows a typical Elman RNN, proposed by Elman in 1990 [10], where the hidden layer neurons have feedback connections to context unit input elements.

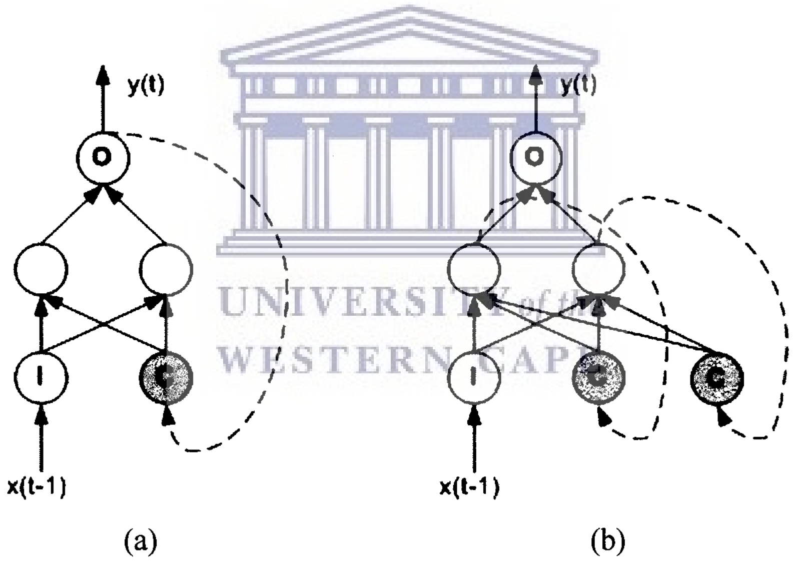

Figure 6-1: (a) Jordan RNN and (b) Elman RNN.

Figure 6-2 shows a fully recurrent neural network, in which every unit has feedback connections to all units within the same layer and all preceding units. The feedback connections that are present in all these RNN network types are what ultimately makes it possible for RNNs to build memory of time series events that took place an arbitrary number of time steps earlier in during sequence classification.

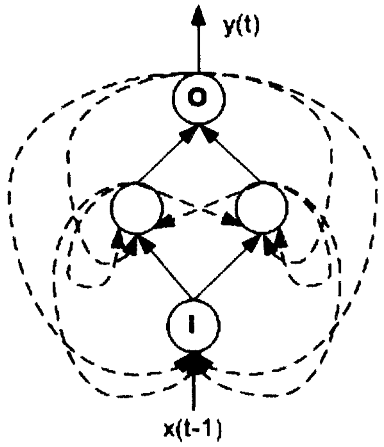

Figure 6-2: A fully recurrent neural network.

In most cases, partially recurrent networks are preferred over fully recurrent ones, simply because they are much easier to deal with in terms of time and space complexity. Architecture selection ultimately depends on the problem at hand. Elman RNNs, for instance, will perform better on certain problems than Jordan RNNs because, since it does not contain any feedback connections in its hidden layer, the latter cannot build up a memory of inputs that are not directly reflected in the output.

In this thesis, we will use some type of RNN architecture to perform a supervised temporal learning task. So, how do one train recurrent neural networks ? The next two subsections briefly explore two widely used and documented recurrent neural network learning techniques, namely backpropagation-through-time and real-time recurrent learning.

6.2.1. Backpropagation-Through-Time

Backpropagation-through-time (BPTT) is a simple extension of the standard backpropagation learning algorithm for feedforward neural networks, which aims at computing error gradient information in order to do gradient descent. The BPTT algorithm can easily be explained at the hand of a recurrent neural network unfolded in time.

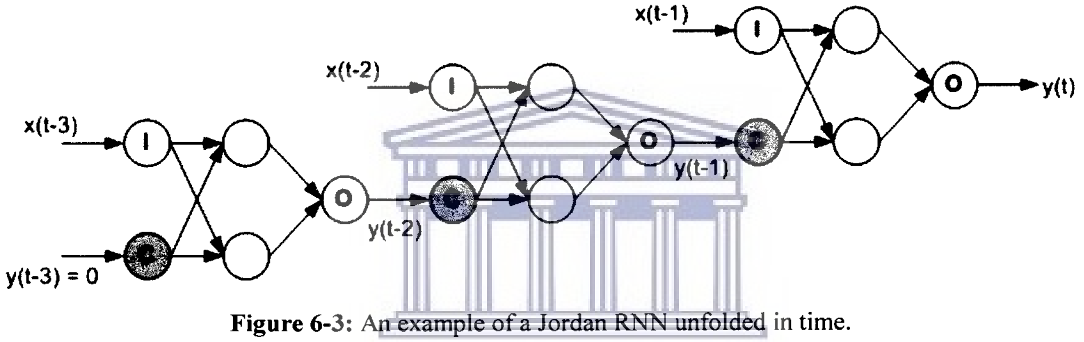

Figure 6-3: An example of a Jordan RNN unfolded in time.

Figure 6-3 shows a simple Jordan RNN unfolded over three time steps. Unfolding an RNN over time leads to a conceptual static feedforward neural network, with external inputs flowing into the network at each time step while the initial input is propagated forward through the network. In other words, at each time step of a sequence classification, a copy of the network is made and linked to the copy of the previous time step through the feedback connection(s). At the end of the sequence, an error signal can be computed and weight deltas calculated by propagating the error backwards through the unfolded network. After calculating weight deltas for all time steps (i.e. copies of the network), each weight in the original network is updated with the sum of the weight's deltas in all copies of the network.

As is the case for conventional static feedforward neural networks (Chapter 4), the activation or output value of non-input unit *i* in a recurrent network is given by

$$y_i(t) = f_i(net_i(t)), \tag{25}$$

where f_i is unit *i*'s differentiable squashing function and

$$net_i(t) = \sum_j w_{ij} y_j(t-1), \tag{26}$$

is the network's weighted input to unit *i* at time step *t*.

Suppose we use conventional mean squared error as the recurrent network's performance measure, and suppose output unit *i*'s target at time *t* is $d_i(t)$ while its actual output is $y_i(t)$, then unit *i*'s error signal at time *t* is

$$\delta_i(t) = f_i^{'}(net_i(t))(d_i(t) - y_i(t)), \tag{27}$$

And the backpropagated error signal for some non-output unit *j* is given by

$$\delta_j(t) = f_j^{'}(net_j(t)) \sum_i w_{ik} \delta_i(t+1).. \tag{28}$$

The total contribution of unit *l* to the update of weight w_{jl}, which connects unit *l* to unit *j*, is then $\alpha \delta_j(t) y_l(t-1)$, where α is the learning rate [27].

In short, the BPTT algorithm can be summarised as follows:

1. An input sequence is fed into the network and the network unfolded over time, with the error being calculated according to some chosen error measure at each time step.
2. At the end of an input sequence, the error is injected back into the network, and the delta for each weight at each time step is consequently calculated.
3. Each weight is updated with the sum of its deltas over all time steps.
4. The initial state of the network is reset and the above process repeated for each training exemplar.

In order to use BPTT to train a network, we need to store a copy of the network's weights at each time step, which amounts to storing a total of wh numbers for a network with w weights unfolded over h time steps.

6.2.2. Real-Time Recurrent Learning

An alternative approach to propagating error information backwards over time, is to propagate activity gradient information forward [27]. This leads to a learning algorithm called real-time recurrent learning (RTRL) where the gradient information is computed on the fly as inputs are presented to the network.

By adding time to Equation (1) we can define the input to a recurrent network unit as

$$y_k(t) = f_k(\sum_{l \in U \cup I} w_{kl} x_l(t)) \tag{29}$$

where U denotes the set of indices k such that x_k is the output of a unit in the network, I the set of indices k for which x_k is an external input, and f_k represents unit k's squashing function.

As in [27], we can define the error of an arbitrary output unit k in a recurrent network as

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise} \end{cases} \tag{30}$$

where *T(t)* denotes the set of indices at time t for which a teacher signal (i.e. output unit target value) exists.

The negative[1] of the total network error at time t can then be defined as

$$E(t) = -\frac{1}{2} \sum_{k \in U} (e_k(t))^2 \,. \tag{31}$$

[1] As in [27] the training objective is changed from minimising the overall network error to maximising the negative of the overall network error in order to get rid of the minus sign in equation (33), which makes subsequent derivations easier to deal with..

Let us now define a quantity that measures the sensitivity of the output value of unit k at time t to a small increase in the value of weight w_{ij},

$$p_{ij}^{k}(t)=\frac{\partial y_k(t)}{\partial w_{ij}}. \tag{32}$$

Differentiating Equation (31) with regards to w_{ij} using Equation (30), and substituting (32) yields

$$\frac{\partial E(t)}{\partial w_{ij}}=-\sum_{k\in U}e_k(t)\left(-\frac{\partial y_k(t)}{\partial w_{ij}}\right)=\sum_{k\in U}e_k(t)p_{ij}^{k}. \tag{33}$$

Furthermore, differentiating Equation (25) using Equation (29) gives

$$\begin{aligned}p_{ij}^{k}(t+1)&=\frac{\partial}{\partial w_{ij}}\left(f_k\left(\sum_{l\in U}w_{kl}x_l(t)\right)\right)\\&=f_k'(y_k(t))\left[\sum_{l\in U}w_{kl}\frac{\partial x_l(t)}{\partial w_{ij}}+\sum_{l\in U}\frac{\partial w_{kl}}{\partial w_{ij}}x_l(t)\right]\\&=f_k'(y_k(t))\left[\sum_{l\in U}w_{kl}p_{ij}^{l}(t)+\delta_{ik}x_j(t)\right]\end{aligned} \tag{34}$$

using the Kronecker delta δ_{ik} to simplify the equation in the last step. Here, the summation is only done for indices in U since units pertaining to indices in I do not have inputs themselves.

Since it is normally assumed that the initial state of a network has no functional dependence on its weights [27], we have

$$p_{ij}^{k}(t_0)=\frac{\partial y_k(t_0)}{\partial w_{ij}}=0 \tag{35}$$

for the first time step t_0.

Equations (34) and (35) can be used to compute the quantity $p_{ij}^{k}(t)$ at each time step, and combining this with the error vector through Equation (33) yields the negative error gradient. The negative error gradient information can in turn be used to update the weights in the network in real time because the $p_{ij}^{k}(t)$ values are available at each time step.

All the $p_{ij}^{k}(t)$ values need to be stored and updated, but the space requirements do not grow with each time step as with BPTT.

6.3. The Vanishing Gradient Problem

The current state in a recurrent network depends on all previous time steps of a particular classification attempt. Most RNN learning algorithms fail on problems with long minimum time lags between input signals and their resulting error signals. When an error signal is injected back into a network - and hence flowing back in time - it tends to either blow up or vanish. An error that blows up leads to oscillating weights and unpredicted network behaviour, while a vanishing error will quite obviously cause training to slow down or completely stop during a network's attempts to learn to bridge long time lags.

To gain a better understanding of why the error tends to suffer from exponential decay over long time lags, it is worthwhile to take a look at Hochreiter's analysis [14]. Apply BPTT to a fully recurrent network whose non-input indices range from 1 to *n*. The error signal computed at an arbitrary unit *a* at time step *t* is propagated back through time for *q* time steps to arbitrary unit *b*, causing the error to be scaled by the factor

$$\frac{\partial\delta_b(t-q)}{\partial\delta_a(t)} = \begin{cases} f_b'(net_b(t-1))w_{ab} & q=1 \\ f_b'(net_b(t-q))\sum_{i=1}^{n}\frac{\partial\delta_i(t-q+1)}{\partial\delta_a(t)}w_{ib} & q>1 \end{cases}. \tag{36}$$

Setting $i_0 = a$ and $i_q = b$, one can derive the following equation through proof by induction:

$$\frac{\partial \delta_b(t-q)}{\partial \delta_a(t)} = \sum_{i_1=1}^{n} \cdots \sum_{i_q=1}^{n} \prod_{j=1}^{q} f'_{i_j}(net_{i_j}(t-j))w_{i_j i_{j-1}} \tag{37}$$

where the sum of the n^{q-1} terms, $\prod_{j=1}^{q} f'_{i_j}(net_{i_j}(t-j))w_{i_j i_{j-1}}$, represents the total error flowing back from unit a to unit b.

It then follows that, if

$$f'_{i_j}(net_{i_j}(t-j))w_{i_j i_{j-1}} > 1 \tag{38}$$

for all j, then the product term in Equation (37) will grow exponentially with increasing q. In other words the error will blow up and conflicting error signals arriving at unit b can lead to oscillating weights and unstable learning [14]

On the other hand, if

$$f'_{i_j}(net_{i_j}(t-j))w_{i_j i_{j-1}} < 1 \tag{39}$$

for all j, then the product term will decrease exponentially with q and the error will vanish, making it impossible for the network to learn anything or slowing learning down to an unacceptable speed.

6.4. Long Short-Term Memory

So how do we avoid vanishing error signals in recurrent neural networks ? The most obvious approach would be to ensure constant error flow through each unit in the network. The implementation of such a learning method, however, might not be as obvious or straightforward. Hochreiter and Schmidhuber addressed the issue of constant error flow in [14] by introducing a novel machine learning method they dubbed long short-term memory (LSTM).

6.4.1. The Constant Error Carrousel

Concentrating on a single unit j and looking at one time step, it follows from Equation (36) that j's local error back flow at time t is given by $\delta_j(t) = f_j'(net_j(t))\delta_j(t+1)w_{jj}$, with w_{jj} the weight which connects the unit to itself. It is evident from Equations (38) and (39) that, in order to enforce constant error flow through unit j, we need to have

$$f_j'(net_j(t))w_{jj} = 1.$$

Integrating the equation above gives

$$\int \partial f_j(net_j(t)) = \int \frac{1}{w_{jj}} \partial_{net_j}$$
$$\therefore f_j(net_j(t)) = \frac{net_j(t)}{w_{jj}}.$$

We learn two things from the above equation:

1. f_j has to be linear; and
2. unit j's activation has to remain constant, i.e.
   $y_j(t+1) = f_j(net_j(t+1)) = f_j(w_{jj}y_j(t)) = y_j(t)$.

This is achieved by using the identity function $f_j : f_j(x) = x, \forall x$, and by setting w_{jj}=1. Hochreiter and Schmidhuber refer to this as the constant error carrousel (CEC). The CEC performs a memorising function, or to be more precise, it is the device through which short-term memory storage is achieved for extended periods of time in an LSTM network.

6.4.2. Input and Output Gates

The previous subsection introduced the CEC as an arbitrary unit j with a single connection back to itself. Since this arbitrary unit j will also be connected to other units in the network, the effect that weighted inputs have on the unit has to be taken into account. Likewise, the effect that unit j's weighted outputs have on other units in the network has to be closely scrutinised.

In the case of incoming weighted connections to unit j, it is quite possible for these weights to receive conflicting update signals during training, which in turn makes learning difficult because the same weight is used to store certain inputs and ignore others [14]. In the same way, output weights originating at unit j can receive conflicting weight update signals because the same weights can be used to retrieve j's contents sometimes, and prevent j's output to flow forward through the network at other times.

The problem of conflicting update signals for input and output weights is addressed in the LSTM architecture with the introduction of multiplicative input and output gates. The input gate of a memory cell is taught when to open and when to close, thereby controlling when network inputs to a memory cell are allowed to adjust its memory contents; an input gate therefore also helps to protect the memory cell contents from being disturbed by irrelevant input from other network units or memory cells. In the same way, an output gate is taught when to open and close, thereby controlling access to the memory cell's contents. An output gate therefore helps to protect the memory contents of other memory cells from being disturbed by irrelevant output from its memory cell.

6.4.3. The LSTM Memory Block

An LSTM network unit containing a CEC, an input gate, and an output gate, is called a *memory cell.* Figure 6-4 below depicts the standard architecture of an LSTM memory cell.

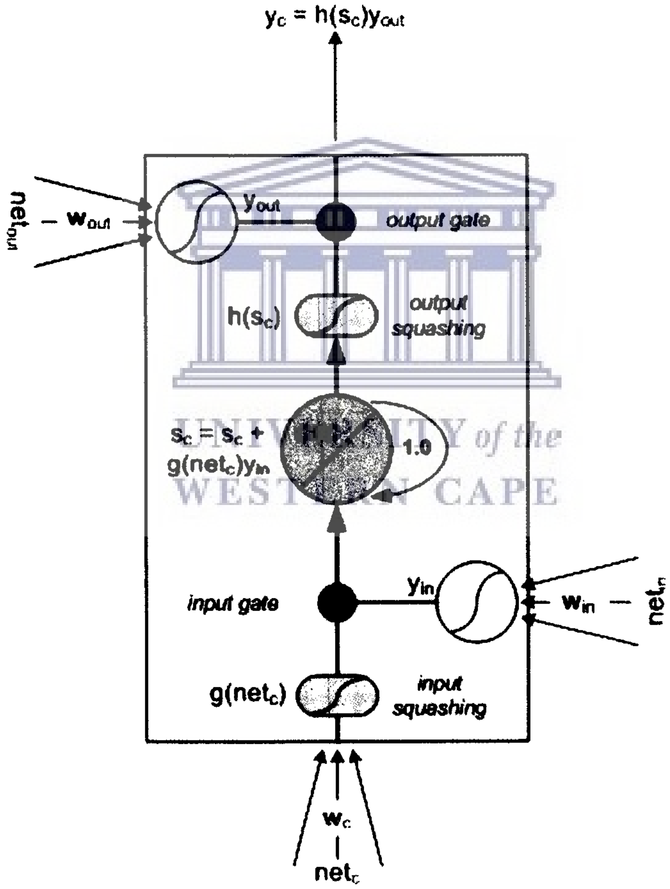

Figure 6-4: An LSTM memory cell.

One or more memory cells that share input and output gates between them are grouped together in a unit called the *memory block.* Each cell in a memory block has its own CEC at its core to ensure constant error flow through the cell in the absence of input or error signals, thereby solving the vanishing gradient problem. The activation of a CEC is called the internal cell state [11].

6.4.4. Forward Pass with LSTM

Borrowing from [11], we define the following indexes: j indexes memory blocks; v indexes memory cells in block j such that c_j^v denotes the v-th cell of the j-th memory block; w_{lm} denotes a weight connecting unit m to unit l; and m indexes source units as applicable.

A memory cell has three major sets of inputs: standard cell input, input gate input and output gate input. The cell input to arbitrary memory cell c_j^v at time t is

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m} y_m(t-1). \tag{40}$$

The input gate activation y^{in} is

$$y_{in_j} = f_{in_j}(\sum_m w_{in_j m} y_m(t-1)) \tag{41}$$

while output gate activation is computed as

$$y_{out_j} = f_{out_j}(\sum_m w_{out_j m} y_m(t-1)). \tag{42}$$

The squashing function f used in the gates is a standard logistic sigmoid with output range [0,1]:

$$f(x) = \frac{1}{1+e^{-x}}. \tag{43}$$

Assuming that the internal state of a memory cell at time t=0 is $s_{c_j^v}(0) = 0$, the internal state of memory cell c at time t is calculated by adding the squashed, gated input to the internal cell state of the last time step, $s_c(t$-1):

$$s_{c_j^v}(t) = s_{c_j^v}(t-1) + y_{in_j}(t)g(net_{c_j^v}(t)) \quad \text{for } t > 0, \tag{44}$$

where the input squashing function g is given by

$$g(x) = \frac{4}{1+e^{-x}} - 2. \tag{45}$$

The cell output y^c can then be calculated by squashing the internal state s_c and multiplying the result by the output gate activation,

$$y_{c_j^v}(t) = y_{out_j}(t)h(s_{c_j^v}(t)), \tag{46}$$

where the output squashing function h is given by

$$h(x) = \frac{2}{1+e^{-x}} - 1. \tag{47}$$

It was later suggested that the output squashing function h can be removed from Equation (46) because no empirical evidence exists that it is needed [12]. Equation (46) is thus changed to

$$y_{c_j^v}(t) = y_{out_j}(t)s_{c_j^v}(t). \tag{48}$$

Finally, the activations of the output units (indexed by k) can be calculated as

$$y_k(t) = f_k(net_k(t)), \quad net_k(t) = \sum_m w_{km} y_m(t-1). \tag{49}$$

6.4.5. Forget Gates

The standard LSTM architecture described above, although powerful, has its limitations. According to [11], the cell state s_c often tends to grow linearly during presentation of a time series, which might lead to saturation of the output squashing function h if the network is presented with a continuous input stream. Saturation of h will reduce the LSTM memory cell to a regular BPTT unit, causing it to lose its ability to memorise. Furthermore, saturation of h will cause its derivative to vanish, which will in turn block any error signals from entering the cell during back propagation. Cell state growth can be limited by manually resetting the state at the start of each new sequence. This method, however, is not practical in cases where the sequence has no discernable end, or where no external teacher exists for subdividing the input sequence into sub sequences. A sequence of credit card transactions is one such an example.

Gers, Schmidhuber and Cummins [11] solved the cell state saturation problem by introducing a third gate into the LSTM memory block architecture. This third gate is designed to learn to reset cell states when their content becomes useless to the network. In a way it forces the cell to forget what it had memorised during earlier time steps, and is therefore called a *forget gate*. Figure 6-5 on the next page shows the extended LSTM memory cell with a forget gate.

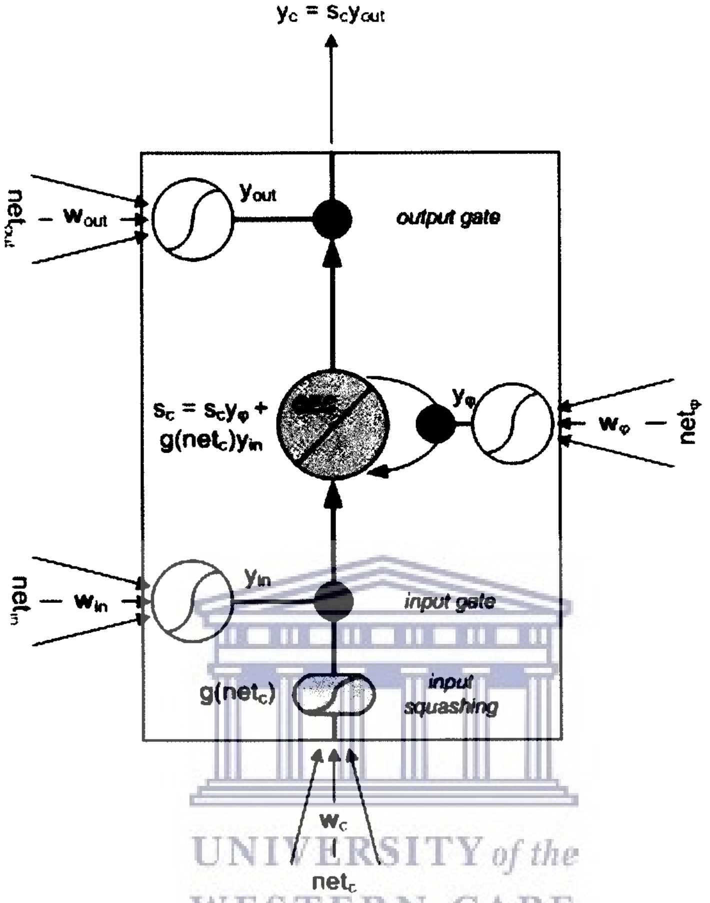

Figure 6-5: An LSTM memory cell with a forget gate.

The only change that the addition of a forget gate introduces to the forward pass of LSTM is in Equation (44), where the squashed, gated cell input is now added to the forget gated cell state of the previous time step, instead of just the basic cell state:

$$s_{c_j^v}(t) = y_{\varphi_j} s_{c_j^v}(t-1) + y_{in_j}(t) g(net_{c_j^v}(t)) \quad \text{for } t > 0, \tag{50}$$

where y_φ is the forget gate activation and is calculated (like the activations of the other gates) by squashing the weighted input to the gate:

$$y_{\varphi_j} = f_{\varphi_j}\left(\sum_m w_{\varphi_j m} y_m(t-1)\right). \tag{51}$$

Once again, f_φ is the standard logistic function of Equation (43).

It was suggested in [11] to initialise input and output gate weights with negative values, and forget gate weights with positive values, in order to ensure that a memory cell behaves like a normal LSTM cell during the beginning phases of training, as not to forget anything until it has learned to do so.

6.4.6. Peephole Connections

Another addition to the LSTM memory cell architecture is the notion of peephole connections, first introduced by Gers, Schraudolph and Schmidhuber in 2002 [12]. Peephole connections basically connect a memory cell CEC with the memory block gates through additional waited connections. This was done to give the gates some feedback on the internal cell states which they are suppose to control.

Peephole connections are not used in any of the learning tasks studied in this thesis; their use in initial experiments did not show significant improvement in generalisation performance and for the sake of simplicity further investigations with peephole connections were therefore discontinued. The interested reader can get more information on peephole connections and the resulting modified LSTM forward pass in [12].

6.4.7. LSTM Training – A Hybrid Approach

The LSTM backward pass makes use of truncated versions of both the BPTT and RTRL algorithms to calculate the weight deltas for a particular time step. This hybrid approach to RNN learning was first suggested by Williams (1989) and later described by Schmidhuber (1992). In the context of LSTM learning and the BPTT and RTRL algorithms, truncation means that errors are cut off once they leak out of a memory cell or gate although they do serve to change incoming weights [11].

The standard BPTT algorithm is used to calculate the weight deltas for output unit weights while truncated BPTT is used for output gate weights. A truncated version of RTRL is used to calculate deltas for cell, input gate, and forget gate weights. The rationale behind truncation is that it makes learning computationally feasible without significantly decreasing network training and generalisation performance. The remainder of this section describes the gradient-based learning algorithm for LSTM networks with forget gates. More complete versions can be found in [11], [12] or [14].

As is normal with gradient-based learning, the aim is to minimise the objective function E by changing each connecting weight in the network by some value at each time step. Once again the error generated at the output layer at time t is taken to be the mean squared error,

$$E(t) = \frac{1}{2}\sum_{k}(t_k(t) - y_k(t))^2 \tag{52}$$

with t^k the target output and y^k the actual activation of output unit k.

As mentioned earlier, truncated BPTT is used to calculate the weight change amounts of output unit weights and output gate weights. Therefore, taking an arbitrary output weight w_{km} (following the notation used in [12]) which connects unit m to output unit k, the weight change is given by

$$\Delta w_{km}(t) = \alpha\delta_k(t)y_m(t-1), \quad \delta_k(t) = -\frac{\partial E(t)}{\partial net_k(t)} \tag{53}$$

where α is the learning rate.

Differentiating Equation (52) and substituting into Equation (53) yields the externally injected error

$$\delta_k(t) = f_k^{'}(net_k(t))[t_k(t) - y_k(t)] \tag{54}$$

Since standard BPTT is also used to calculate the weight changes for output gate weights, the delta of a weight connecting an output gate of arbitrary memory block *j* to unit *m* is given by

$$\Delta w_{out_j m}(t) = \alpha \delta_{out_j}(t) y_m(t), \tag{55}$$

$$\delta_{out_j}(t) \overset{tr}{=} f'_{out_j}(net_{out_j}(t)) \left(\sum_{v=1}^{S_j} s_{c_j^v}(t) \sum_k w_{kc_j^v} \delta_k(t) \right). \tag{56}$$

Again following the notation used in [12], $\overset{tr}{=}$ indicates that the error is being truncated.

It is clear from Equations (33), (34) and (35) (Section 6.2.2) that RTRL requires the calculation of partial derivates at each time step in order to ultimately compute the necessary weight update deltas for those time steps. Since, in the context of LSTM, we use RTRL to calculate the weight deltas for cell input weights, input gate weights and forget gate weights, the following partial derivates have to be updated at each time step:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \overset{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} y_{\varphi j}(t) + g'(net_{c_j^v}(t)) y_{in_j}(t) y_m(t-1), \tag{57}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} \overset{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} y_{\varphi j}(t) + g(net_{c_j^v}(t)) f'_{in_j}(net_{in_j}(t)) y_m(t-1), \tag{58}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} \overset{tr}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} y_{\varphi j}(t) + s_{c_j^v}(t-1) f'_{\varphi_j}(net_{\varphi_j}(t)) y_m(t-1). \tag{59}$$

Following from Equation (35), the above partials are set to zero for the first time step ($t = 0$).

The partials from Equations (57), (58) and (59) can now be used to calculate the weight deltas for cell input weights, input gate weights and forget gate weights respectively:

$$\Delta w_{c_j^v m}(t) = \alpha \, e_{s_j^v}(t) \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}}, \tag{60}$$

$$\Delta w_{in_j m}(t) = \alpha \sum_{\mathrm{v}=1}^{\mathrm{S_j}} e_{s_j^v}(t) \frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}}, \tag{61}$$

$$\Delta w_{\varphi_j m}(t) = \alpha \sum_{\mathrm{v}=1}^{\mathrm{S_j}} e_{s_j^v}(t) \frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}}. \tag{62}$$

The quantity $e_{s_j^v}$ represents the internal state error for each memory cell and is calculated as follows:

$$e_{s_j^v}(t) \overset{tr}{=} y_{out_j}(t) \left(\sum_k w_{kc_j^v} \delta_k(t) \right). \tag{63}$$

Notice that for the input and forget gate weight deltas in Equations (61) and (62), the effect of all memory cells in a memory block is taken into account by using the sum of the internal state errors. This is done because the memory cells in a memory block share the input, output and forget gates between them.

6.5. Summary

This chapter dealt explicitly with recurrent neural networks as a possible solution to the problem of detecting fraudulent credit card transactions. The motivation behind this is that recurrent neural network training is a tried and tested technique for dealing with sequence classification problems, of which the classification of chains of time ordered credit card transactions is a perfect real world example. A serious shortcoming of recurrent neural networks, namely the vanishing gradient problem, was highlighted and a possible solution in the form of LSTM recurrent neural networks was described. Here we are interested in modelling very long time series, and in [14] and [11] it was shown that LSTM, in particular, is exceedingly good at modelling long-term dependencies in sequence classification tasks.

Ultimately, this thesis will compare the performance of support vector machines with that of long short-term memory recurrent neural networks as applied to the non-trivial problem of fraud transaction detection. However, before this can be done a criteria for evaluating the performance of each methodology needs to be selected. The next chapter looks at the performance evaluation of these two machine learning techniques as applied to a binary classification problem in more detail.

Chapter 7

EVALUATING PERFORMANCE

7.1. Introduction

This chapter introduces two techniques that will be used to evaluate the generalisation performance of the networks used in the experiments in this thesis. The popular mean squared error is discussed first along with its advantages and disadvantages. It is also shown why the mean squared error is not a good measure of performance when dealing with skew data sets and binary classifiers. The discussion on mean squared error is followed by a discussion of an alternative performance measure that is more applicable to fraud detection and binary classifiers in general – the receiver operating characteristic curve.

7.2. Mean Squared Error

The mean squared error (MSE) is probably the most common error measure used in both the training of a wide range of neural networks, and the calculation of neural network generalisation performance measures. The MSE of a neural network is calculated by simply summing the sum of the square of the difference between the actual output and expected output values, and dividing the result by the number of outputs n and training exemplars m:

$$MSE_{net} = \frac{1}{mn}\sum_{d}\sum_{k}(d_k - y_k)^2 \tag{64}$$

As usual, indices k denotes output neurons.

When used for training, the factor $1/n$ in Equation (64) is often simplified to 1/2 in order to make subsequent calculations more elegant[2]. The ease with which MSE's derivative can be calculated is one of the reasons for its popularity in the machine learning field. Other reasons include the fact that MSE emphasise large errors more than smaller ones. In contrast to the positive features of MSE, it remains a pure mathematical construct which fails to take the cost of classification errors into account. Furthermore, it also fails to clearly distinguish minor errors from serious errors.

When dealing with skewed data sets which exhibit distributions heavily in favour of a particular class, a performance measure obtained using MSE might be misleading or even nonsensical. Take for example a credit card transaction data set containing data divided into two classes: class **A** representing legitimate transactions and class **B** representing fraudulent transactions. As is normal with transactional data sets, it is expected that the occurrence of class **A** will be in the majority by a large margin, possibly representing in excess of 99.9% of the data set with class **B** representing a mere 0.01%. Let us further assume that we have a really simple classifier that constantly labels input vectors as belonging to class **A**, regardless of the content of the vectors. Also assume that the output squashing function always rounds the output of the classifier to 0.9 (to represent class **A**). In such a case the success rate of the classifier will be 99.9% and its MSE will be negligible; but in spite of this "brilliant" generalisation performance, the classifier would not have identified one single fraudulent transaction and will hence be completely useless. If we go further and attach a cost to misclassifying fraudulent transactions, the shortcomings of MSE becomes even more apparent. The next section introduces an alternative to MSE which is particularly useful in measuring the performance of binary classifiers.

[2] Recall that the error gradient information for weights are calculated by differentiating the error measure with regards to the weights, which will ultimately make the factor 1/2 vanish.

7.3. Receiver Operating Characteristic

In the 1950's, a major theoretical advance was made by combining detection theory with statistical decision theory, and the importance of measuring two aspects of detection performance was realised. It was realised that one must measure the conditional probability that the observer decides the condition is present, the so-called hit rate, or that the observer decides the condition is present when it is actually not, called the false alarm rate. This can be represented in the format shown in Table 7-1, called the outcome probability table. This table is often used to visualise the probability of the four possible classification outcomes of binary classifiers, or in our case a fraud detection system. In Table 7-1 the rows represent the actual labels [legitimate,fraud] that an arbitrary transaction might have, while the columns represent the labels that a classifier might assign to such a transaction.

Assigned **Actual**	legitimate	fraud
legitimate	$P(\text{correct}\mid\text{legal})$	$P(\text{false alarm}\mid\text{legal})$
fraud	$P(\text{fraud not detected})$	$P(\text{correct}\mid\text{fraud})$

Table 7-1: The outcome probability table [4].

According to [4], a high correct classification probability is represented by

$$P(\text{correct}) = P(\text{correct}\mid\text{fraud})P(\text{fraud}) + P(\text{correct}\mid\text{legal})P(\text{legal}). \tag{65}$$

Put differently, Equation (65) simply states that the classifier will be at its best when it succeeds in maximising both the number of correctly classified fraudulent transactions and correctly classified legal transactions. This is obviously achieved by minimising the number of misclassifications, or in other words minimising the weighted sum

$$C = c_1 P(\text{fraud not detected}) + c_2 P(\text{false alarm}\mid\text{legitimate}). \tag{66}$$

Since it is difficult to determine c_1 and c_2 in practice, it makes more sense to attempt to maximise the number of fraudulent transactions detected while minimising the false alarm rate.

Table 7-2 shows a simplified version of the classification outcome matrix containing descriptions and abbreviations of the four possible outcomes of a classification with a binary target class. These abbreviations and terminology will be used in the derivations to follow.

Assigned **Actual**	legitimate	fraud
legitimate	True Negative (TN)	False Positive (FP)
fraud	False Negative (FN)	True Positive (TP)

Table 7-2: A simplified outcome/decision matrix.

In binary classification problems, a classifier simply attempts to predict whether a given condition is present or not for each item in a data set. In our case, it attempts to predict whether a transaction is genuine or fraudulent, or whether, given a range of earlier transactions, the next transaction in a sequence is legitimate or not. Decisions on whether a condition is present or absent are, in most cases, based on the activation level that a single output neuron achieves during classification. A low activation points towards the condition being absent, while a high activation level indicates that the condition is indeed present. Two types of errors are possible with the decision scheme set out in Table 7-2. A *type I* error, or false positive, is made when the classifier decides that the condition is present when it is not, and a *type II* error, or false negative, is made when the classifier decides that the condition is absent when it is in fact present.

In order to distinguish between high and low activations in the continuous activation spectrum of a single output neuron, a threshold is normally chosen and any activation lower than the threshold is then perceived as an indication that the condition in question is absent, while any activation equal to or above the threshold suggests that the condition is probably present. Figure 7-1 illustrates the high and low activation probabilities of an arbitrary output neuron. In statistics, the curve on the left representing the probability that the condition is absent is called the null hypothesis, while the curve on the right measuring the probability of the condition being present is called the alternative hypothesis [22]. It is important to note that Figure 7-1 shows a significant overlap between the null and alternative output probabilities, i.e. a somewhat grey decision area where the decision outcome is not certain beyond any doubt. This is usually the case in real world problems where perfect performance is unachievable.

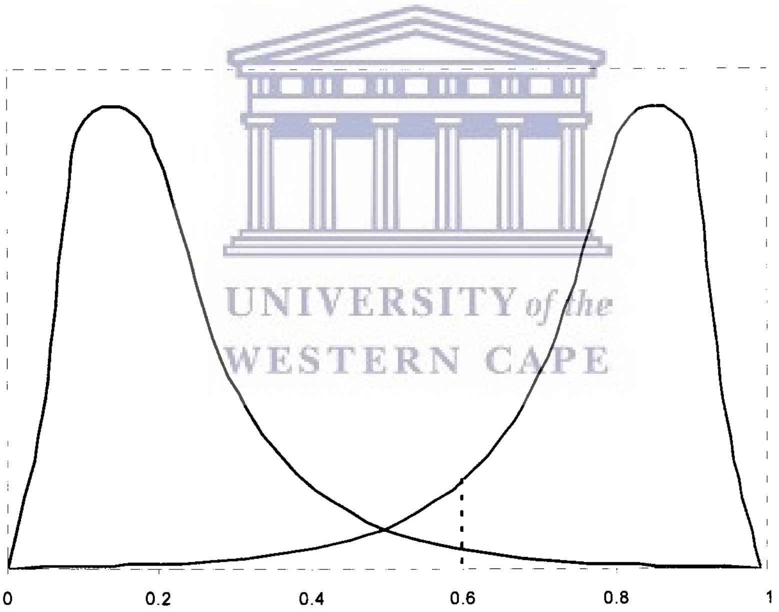

Figure 7-1: Output neuron probability distributions under null and alternative hypothesis.

More often than not, and for no apparent good reason, the threshold is set to exactly 0.5 in theoretical experiments. Square in the middle of neuron activation range is not necessarily the optimum place for an output neuron's activation threshold, and better generalisation performance might be achieved by shifting the threshold higher or lower.

Let us assume for argument's sake that a decision threshold has been set at 0.6 for our fraud detection network, as shown by the dotted line in Figure 7-1. This means that if an activation of at least 0.6 is achieved when propagating a previously unseen transaction (or sequence of transactions) forward through the network, we will assume that the condition is present (i.e. the transaction is fraudulent). If the neuron only achieves an activation level lower than this threshold, we will assume that the transaction is legitimate.

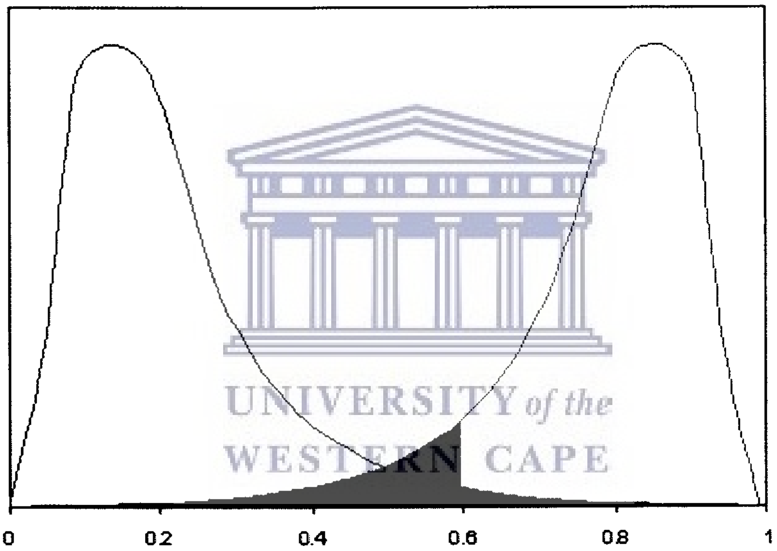

Figure 7-2: Decision outcome error probability.

The probability of making a type I error is given by the area under the left curve to the right of the threshold, and is shown as the blue area in Figure 7-2. On the other hand, the probability of making a type II error is given by the area under the right curve to the left of the threshold, shown as the red area in Figure 7-2. It is obvious from the above that the higher the threshold is set, the smaller the chance becomes of making a type I error (false positive), but the bigger the chance of making a type II error (false negative).

The probability of making a false positive decision is also called the *false alarm rate*: *FAR = P(false alarm|legitimate)*. In terms of Table 7-2, the false alarm rate can be derived as:

$$\begin{aligned} \text{FAR} &= \frac{\#\text{ false alarms}}{\#\text{ all alarms}} \\ &\geq \frac{\#\text{ false alarms}}{\#\text{ all alarms}} \frac{\#\text{ all alarms}}{\#\text{ all legals}} \quad {}^{3} \\ &= \frac{\#\text{ false alarms}}{\#\text{ all legals}} \\ &= \frac{FP}{FP + TN} \end{aligned} \tag{67}$$

Another quantity of interest is the *true positive rate* (or *hit rate*), which is the probability of making a true positive decision (i.e. *1 – P(fraud not detected)*), and is given by

$$\begin{aligned} \text{TPR} &= 1 - \frac{\#\text{ false negatives}}{\#\text{ all alarms}} \\ &= \frac{TP}{TP + FN} \end{aligned} \tag{68}$$

Together, these two quantities can give a good indication of a binary classifier's generalisation performance. Plotting these two quantities against each other for increasing thresholds from 0 to 1, with the false alarm rate on the x-axis and the true positive rate on the y-axis, yields a curve called the *Receiver Operating Characteristic* (ROC) curve (see Figure 7-3) .

The ROC curve was first used in the Signal Detection Theory field as an indication of a radar operator's ability to distinguish between noise, friendly ships, and enemy ships when viewing blips on a radar screen. In the 1970's, the ROC curve found its

[3] Allowable since $1 \geq \frac{\#\text{ all alarms}}{\#\text{ all legals}}$ in most realistic cases [4].

way into the medical sciences where it was found useful for interpreting medical test results. Because of its success in describing binary classification performance, it was bound to find its way into the machine learning field too.

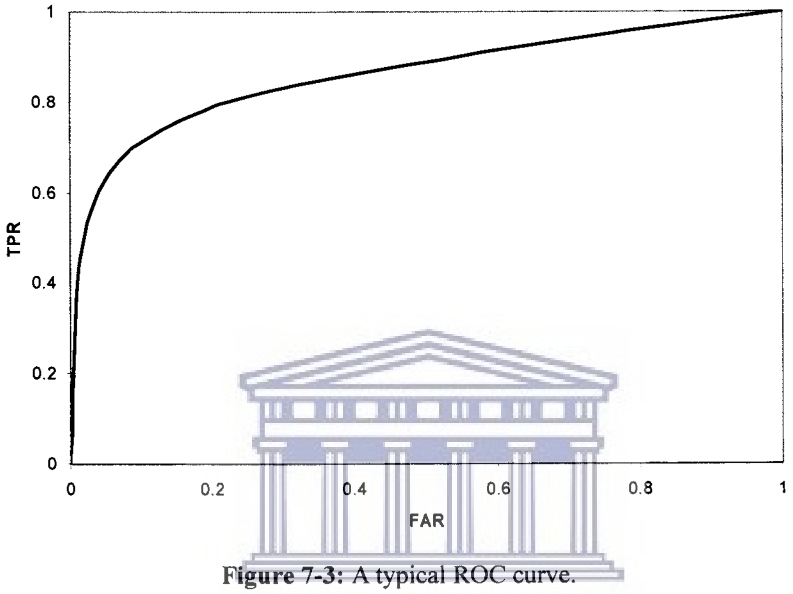

Figure 7-3: A typical ROC curve.

Originally, the classical concept of a detection threshold predicts a linear relationship between the FAR and TPR [13]. This can be shown by defining a quantity that captures the probability that the neuron activation will exceed a set threshold,

$$p = \frac{TPR - FAR}{1 - FAR}. \tag{69}$$

Rearranging (69) leads to the linear relationship between FAR and TPR predicted by the high threshold model:

$$TPR = p + (1 - p) \cdot FAR \tag{70}$$

The bowed-shaped ROC curve of Figure 7-3 contradicts this linear relationship predicted by the high threshold model, and this is one of the reasons why the high threshold model was eventually abandoned and replaced with signal detection theory

[13]. In signal detection theory, the sensory process has no threshold, but rather a continuous output based on random Gaussian noise with which the signal combines when it is present. The relationship between FAR and TPR can be expressed as z-scores of the unit, normal Gaussian probability distribution

$$z(TPR) = \mu_s \cdot \frac{\sigma_n}{\sigma_s} + \frac{\sigma_n}{\sigma_s} \cdot z(FAR) \tag{71}$$

with μ_s the signal-plus-noise distribution, σ_s its standard deviation, and σ_n the standard deviation of the noise.

Signal detection theory as applied to detection sensitivity is outside the scope of this thesis, but it is necessary to note a few points:

1. The ROC curve predicted by signal detection theory is anchored at the (0,0) and (1,1) points (Figure 7-3);
2. When μ_s is greater than zero, the ROC curve takes on a bowed-shape;
3. When μ_s is zero the ROC curve is a straight diagonal line connecting (0,0) and (1,1).

For the purpose of measuring performance in this thesis we are only interested in the total area under the ROC curve, called the area-under-curve (AUC). The area under the ROC curve gives an indication of a classifier's ability to generalise and the bigger this AUC value, the better the classifier. A perfect classifier will have a ROC curve that resembles a right angle with an enclosed area of unity.

7.4. Summary

This chapter discussed two widely used error measures, the mean squared error (MSE) and the receiver operating characteristic (ROC). It was shown that the area under the ROC curve is a more applicable performance measure when dealing with binary classifiers and heavily skewed data sets. The results of experiments conducted in Chapter 9 will include both the MSE and ROC curve area as performance measures.

We now have two methods for fraud detection, and a performance measure applicable to our specific problem. The final step before our methodologies can be put to the test is processing the data set to make it compatible with our learning algorithms. Feature selection and pre-processing is discussed in the following chapter.

Chapter 8

DATA SETS, FEATURE SELECTION AND PRE-PROCESSING

8.1. Introduction

This chapter deals with the tedious task of selecting appropriate features from a real world data set and applying proper pre-processing techniques to them prior to their presentation to a network for classification.

In the context of this thesis a feature represents a unit of data chosen from the list of available data columns present in a data set, and will ultimately form one or more dimensions of a network input vector after the feature has been sufficiently pre-processed. Pre-processing is the task of taking raw features and converting them into values that are suitable for network presentation and classification. Each row in a data set translates to a network input vector when feature selection and pre-processing steps are applied to it.

The different features present in a data set can normally be categorised into four classes according to the type and degree of information they contain [22]. The remainder of this chapter deals with the selection of features from a data set for fraud detection purposes (Section 8.2), the different categories they can be divided into according to their type (Sections 8.3 and 8.4), and the pre-processing techniques used to convert the features in each category to numerical values that can be used as inputs to neural networks.

8.2. Feature Selection

Selecting a salient set of features from a data set and applying the correct pre-processing techniques to them more often than not means the difference between success and failure when building a neural network for classifying real world data. Apart from obvious transaction features such as the amount, what other data features should be selected ?

Available transaction information differs from card issuer to card issuer, and details on this type of information is seen as proprietary and is therefore never published. An even bigger problem inherent in data sets used for fraud detection is the high number of symbolic fields present in the transactional data, while neural networks and other classifier algorithms can only deal with numeric data. The reason for this is that authorisation message exchange between point-of-sale terminals and card management platforms were simply not designed with fraud detection in mind. People might be tempted to simply ignore symbolic fields and only present the numerical data present in the data set to the classifier, but the information contained in symbolic fields can potentially be just as important in correctly classifying credit card transactions as the information contained in the numeric fields. Take for example the transaction date; most conventional fraud detection and expert systems will ignore this field because it is seen as unimportant information [4]. However, a fraud system that attempts to model a time series will obviously find this information very useful. Another reason for using the transaction date might be to model the changes in human shopping behaviour as time progresses. Conventional fraud detection systems seem to have difficulty in coping with changing shopping profiles, especially during holiday seasons.

Although the information contained in authorisation messages might be seen as limited when attempting fraud classification, some statistical values can be calculated to serve as additional features in the input vector. An example of such a statistical value used in conventional fraud classifiers is the transaction velocity. In the context of fraud detection, the velocity value of an account at any given point in time is calculated by counting the number of transactions done on the account during a pre-specified timeframe. The rationale behind this is that if a card is, for example, used to buy five different flight tickets or do ten purchases at different jewellery stores in a short period of time, the activity should be regarded as highly suspicious.

Different velocities can be calculated by including only transactions done at certain types of merchants in one velocity calculation, and including all transactions in another. Merchants can be grouped into industry for the above mentioned velocity calculation by using the standard industry code (SIC) present in every authorisation message. The SIC itself, although symbolic, can also be included in the input vector as a feature. The use of time-based velocity statistics in the conventional fraud detection systems in use today gives a clue to the importance of time when dealing with credit card fraud. After all, the introduction of statistical velocity values ultimately introduces a dynamic component into an otherwise static classification methodology.

Unfortunately, due to reasons of confidentiality, the exact features selected and their value ranges cannot be presented here. It will suffice to state, however, that the most important features used here included the transaction date, transaction amount, cardholder age, account age, months to card expiry, cardholder country, standard industry code of merchant, and two velocity counts.

The following sections deal with identifying correct pre-processing methods for each feature once the selection process is finished. Each feature is a variable with distinct content and data ranges, and it is this information that is central to classifying each feature into one of the four major variable classes.

8.3. Nominal and Ordinal Variables

Variables are considered nominal when they cannot be measured in a quantative way and only contain information to whether they form part of a pre-specified category or not. The only mathematical test that can be done on the nominal variables of a specific category is whether they are equal or not. One nominal variable cannot be considered greater or smaller than another nominal variable of the same type.

Nominal variables are usually presented to a neural network by using as many neurons as the number of distinct values within the category to which the variable belong. One of the neurons is then activated to present a specific value in the category, while all the remaining neurons remain deactivated. This is called *one-of-n* or *one-hot* encoding. This approach works well when the range of values the variable can take on is small, like for instance gender that can only take on one of two values (male or female). When the number of values increases though, this method runs into some difficulties:

1. When the number of distinct values are large, the vectors obtained by applying *one-of-n* encoding are so similar that a network might have difficulty in learning the difference between different categories of the same nominal variable [22].
2. Most nominal variables in real world data can have an extremely large number of categories, like for instance postal code, country code, standard industry code, etc. Representing such variables using *one-of-n* encoding is computationally infeasible most of the time.

An alternative approach that seems to work quite well is to compute a ranking score for each category value a nominal variable of a specified type can have. The ranking score can then be further pre-processed and used as input to a neural network.

The ranking scores for a particular nominal variable are computed by simply counting the number of occurrences of each category within the data set and sorting the resulting counts. The category with the least number of occurrences will then be ranked first, while the category that occurs the most will be ranked last.

Variables that have a true implicit order between categories, but whose physical categorical values have no meaning other than establishing order, are called ordinal variables. Since computing ranking scores for nominal variables almost seems to turn them into ordinal variables, one might be tempted to use the ranking score directly as a dimension of the input vector by encoding the value into one neuron, which is allowable for ordinal variables. The problem with this is that, by encoding the ranking score into a single neuron, it is implied that an order relationship between distinctive categories are present where there is in fact none. Therefore, the ranking scores computed for experiments in this thesis are first converted to binary numbers and then encoded into a fixed number of neurons for each variable. The number of neurons required for binary encoding can be computed using Equation (72), where *c* denotes the number of possible categories the nominal variable can take on.

$$n = \left\lceil \frac{\log c}{\log 2} \right\rceil \tag{72}$$

8.4. Interval and Ratio Variables

Interval variables are numeric in nature, have definite values, and have order relationships between different values of the same variable. Interval variables are usually presented to a neural network using one neuron. Features such as transaction velocity and date - when broken up into days, months and years - are nominal variables. Ratio variables are the same as interval variables, except that a value of zero in a ratio scale is a true zero, while it is arbitrary on an interval scale. The transaction amount is an example of a ratio variable.

In order to present interval or ratio variables to a neural network, they must first be normalised to prevent them from saturating the input neurons. In other words, they must be scaled to adhere with the activation limits of the input neurons of a particular network type. The simplest way to normalise interval or ratio variables is to calculate the minimum (X_{min}) and maximum (X_{max}) values that a variable x can take on in both the test and training set, and then use Equation (73) to scale the value to fall within the network input range, $[I_{min}, I_{max}]$. The network input range is arbitrarily chosen, and is set to [-1,1] for all the experiments in this thesis.

$$y = \left(\frac{x - X_{min}}{X_{max} - X_{min}}\right)\left(I_{max} - I_{min}\right) + I_{min} \tag{73}$$

Equation (74) shows an alternative approach for normalising a value. This statistical approach can be used when the set of values of a feature has the tendency to cluster around some particular value and therefore exhibits an approximately normal distribution [25].

$$y = \frac{x - \mu}{\sigma} \tag{74}$$

Equation (74) states that a value x can be normalised by subtracting its *mean* μ and dividing the result by its *standard deviation* σ. The main reason why Equation (74) is usually preferred over Equation (73), is because it removes all effects that offset and measurement scale can have on a feature's values [22].

Equations (75) and (76) below show the formulae used to compute the mean and standard deviation of a feature.

$$\mu = \frac{1}{N}\sum_{j=1}^{N} x_j \tag{75}$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(xj - \mu)^2} \tag{76}$$

Figure 8-1 shows an illustration of what a normal distribution might look like.

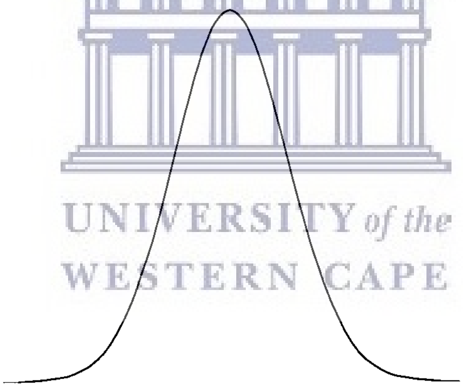

Figure 8-1: A normal distribution with mean 10 and standard deviation 2.

The only downside to using Equations (73) or (74) for normalisation purposes is the fact that they do not deal particularly well with outliers in a data set. Outliers will cause the bigger part of the data to be scaled into a relatively small section of the network input range, causing these values to have much less of an impact on training and generalisation then they should.

Figure 8-2 shows a histogram of the *transaction amount* feature. It is clear from the histogram that the information content of the variable is completely distorted and does not resemble anything even close to a normal distribution. Most neural networks will have a hard time learning anything from a feature such as the one depicted here, unless it is first transformed using a compression transformation to stabilise its variance.

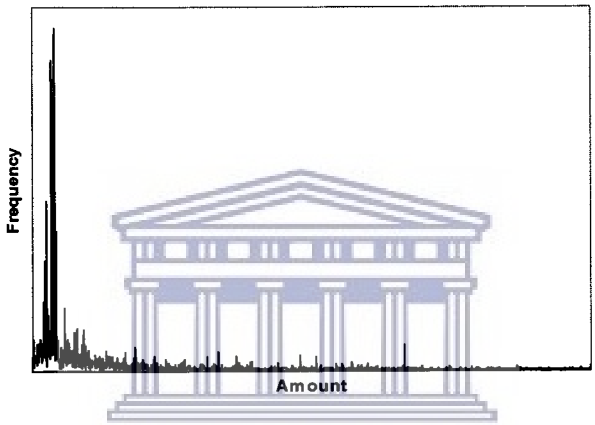

Figure 8-2: A histogram of the *transaction amount* feature prior to pre-processing.

The logarithm is one of the more commonly used transformations and also the one used in the pre-processing steps for the experiments in this thesis. Figure 8-3 shows the histogram of the transaction amount feature after being passed through the logarithmic compression transformation.

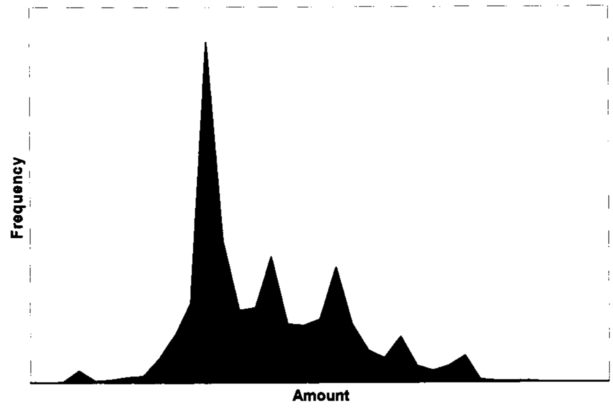

Figure 8-3: A histogram of the *transaction amount* feature after applying a simple logarithmic compression transformation.

Equation (77) shows an improved transform first suggested by J. Tukey (cited in [1]), and Figure 8-4 shows the effect this transformation has on the distribution of the transaction amount feature.

$$y = \arcsin\left(\sqrt{\frac{x}{n+1}}\right) + \arcsin\left(\sqrt{\frac{x+1}{n+1}}\right) \tag{77}$$

The transform in Equation (77) did not seem to have much of an effect on the training and generalisation performance in the experiments of this thesis, and was therefore not used in the final version of the pre-processing algorithm.

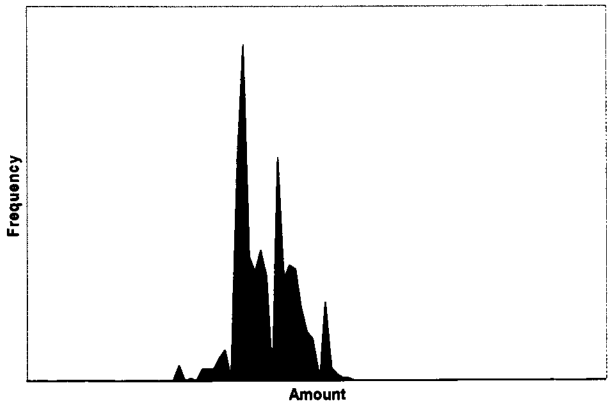

Figure 8-4: A histogram of the *transaction amount* feature after applying the transform in equation (77).

8.5. Summary

After transformation and normalisation, the transaction amount feature still does not quite resemble a normal distribution, and from Figure 8-4 it is clear that its distribution is plagued by a fair amount of skewness and kurtosis; the skewness characterises the degree of asymmetry of a distribution around its mean, while the kurtosis measures the peakedness or flatness of a distribution [25]. Nevertheless, the methods discussed in this chapter had the desired effect on features and significantly improved both the training and generalisation performance of the classification algorithms used in the experiments in this thesis.

The next chapter puts the theory presented in the past five chapters to the test in a series of experiments on our real world data set.

Chapter 9

DETECTING CREDIT CARD FRAUD

9.1. Introduction

In this chapter, the machine learning methodologies discussed in Chapter 5 and Chapter 6 are put to the test on a real world data set containing a mixture of legitimate and fraudulent credit card transactions. In addition to using support vector machines (SVM) and long short-term memory neural networks (LSTM), experiments are also run using a normal static feedforward neural network (FFNN) to put the results into perspective by providing a solid, well-known basis for comparison.

In the first experiment, we attempt to detect fraud using an FFNN and SVM. The data set used for this experiment contains a total of 30876 transactions evened out over 12 months.

The second experiment is done with the same data, but with the transactions time-ordered to test whether LSTM can manage to learn the time series inherent in the data set. The resulting transaction sequences are of variable length; the sequence length distributions of the training and test data sets are shown in Figure 9-1.

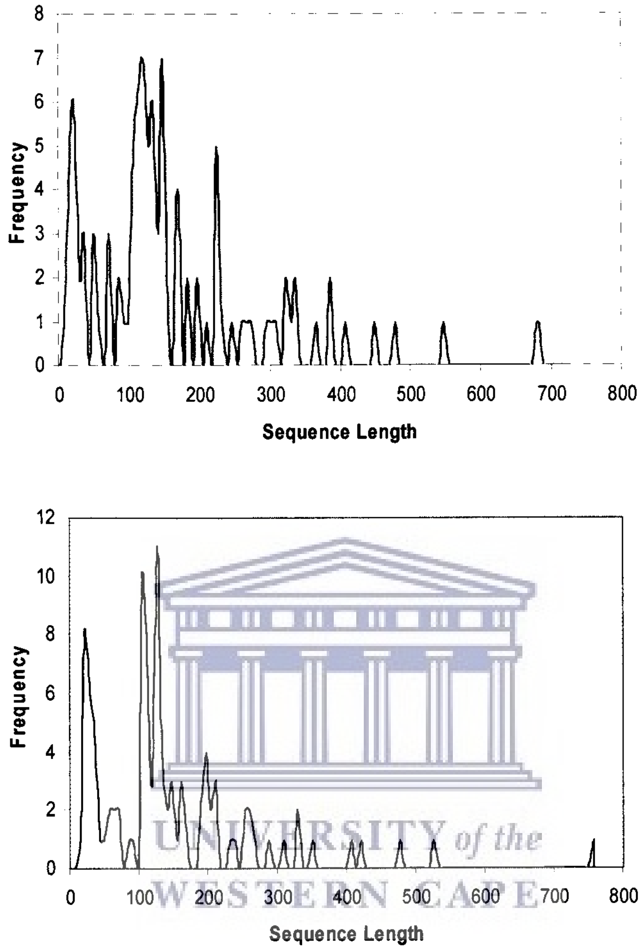

Figure 9-1: Sequence length distributions for the training set (top) and test set (bottom).

In each of the experiment subsections, the set up is explained and the results presented, after which a discussion follows. The implementations of the machine learning algorithms used in the experiments are also discussed at the start of each subsection. The performance measurement tool *PERF Release 5.10* [7] was used in each case to calculate the various performance metrics reported.

9.2. Experiment I: Fraud Detection with FFNN and SVM

9.2.1. Experimental Set Up

The original data set was manipulated by decreasing the number of legitimate transactions to exhibit a class distribution of 99 to 1 (99% legitimate vs. 1% fraud). This was done to reduce the overall size of the data set, making it more manageable for use in our experiments. This is also the distribution used in other card fraud detection literature [4]. The distribution of the original data set was of course even more skewed, containing less than 0.1% fraud. The result of the manipulation was a data set of 30876 transactions which was split into two equal parts, one for training and another for testing. The transactions contained in these two sets are equally spaced out over the 12 months of an arbitrary year, and were pre-processed using the techniques described in Chapter 8 to finally obtain a 41-dimensional input (feature) vector and a corresponding class label for each transaction.

The training and generalisation test cycle is run a total of 30 times for the FFNN to ensure that a statistically meaningful sample is obtained and to ensure that poor performance due to local minima is ruled out. For SVM on the other hand, since its performance depends on what values the kernel parameters and cost function are set to (and since SVMs do not exhibit the randomness inherent in neural network learning), time was rather spend estimating the best values for these parameters than repeating the same experiment 30 times.

9.2.2. Feedforward Neural Network

The feedforward neural network machine learning algorithm used in this experiment is a bespoke implementation in C++ which is executable in both the WIN32 and Linux environments. Nothing extra was done in an attempt to improve the performance of the algorithm - it is a standard static feedforward neural network.

Network Topology, Parameters, and Training

With each individual problem that neural networks are applied to, a lot of thought naturally goes into what network topology is the best to use and what values the network parameters should be set to during training. Figure 9-2 shows the feedforward neural network topology selected for this experiment. A fully connected network was used containing 41 input neurons (a one-to-one mapping to the 41 features representing each transaction), one hidden layer with 20 neurons and one output neuron. During initial experiments the network topology in Figure 9-2 achieved the best results; it was therefore also the topology used in the final FFNN experiments. High activation of the output neuron is an indication that the condition we are looking for is present, which in this case means that the transaction is fraudulent.

Only one hidden layer was used because no real improvement in generalisation performance could be seen when using two. It is also stated in [22] that no theoretical reason exists to ever use more than two hidden layers, while more than one hidden layer is almost never beneficial. Note that Figure 9-2 only shows a subset of the connections between the input and hidden layer because the diagram becomes unintelligible otherwise.

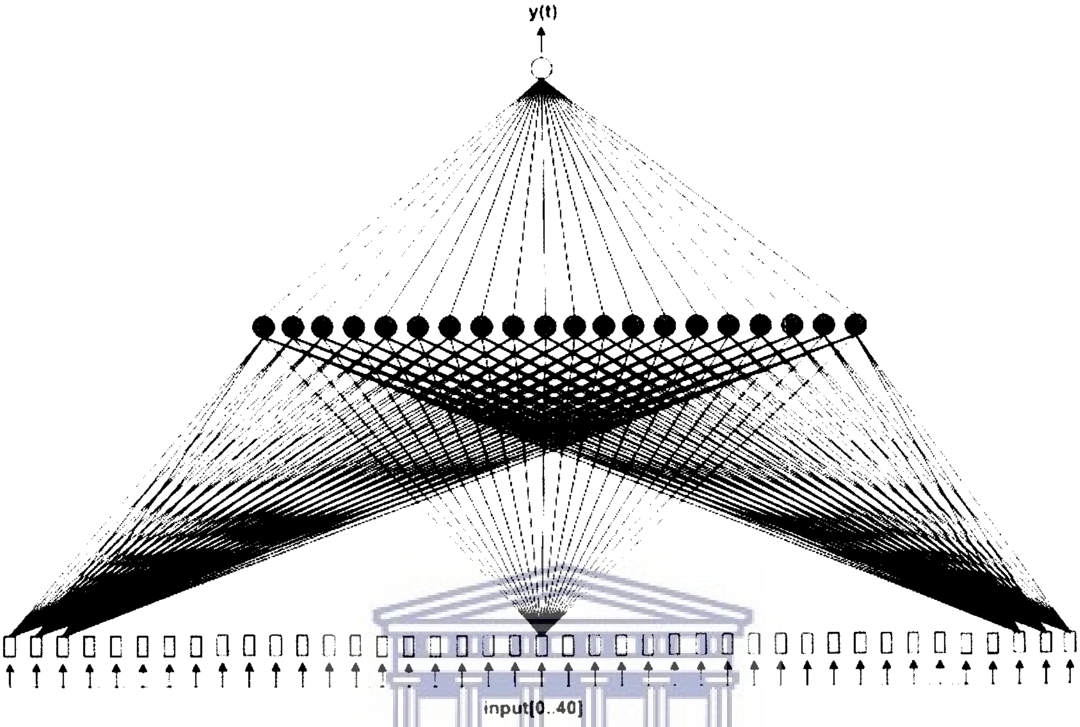

Figure 9-2: FFNN topology for fraud detection.

Training of the network was done using standard backpropagation with both the learning rate and momentum set to 0.3. Training was continued for only 100 epochs and transactions were picked randomly from the training set for each iteration through the network. In the context of this experiment, an epoch is equal to presenting 15438 transactions to the network, i.e. the size of the training set. It was noted that the network achieved its best generalisation performance at roughly 100 epochs. If training was continued past this mark, the network showed a tendency towards overfitting the training data with a resulting decrease in generalisation performance. After training, the test data set is presented to the network for classification and the probability of fraud for each transaction is recorded in order to compute the ROC curve, AUC and RMS.

Results

Figure 9-3 shows the ROC curves of 30 trials with the feedforward neural network algorithm. The shape of the curves, i.e. their deviation from a straight diagonal line and tendency to bow towards the (0,1) corner of the graph, is a clear indication that the network actually learned something. The spread of the 30 test set curves also show the amount of randomness normally associated with neural network training.

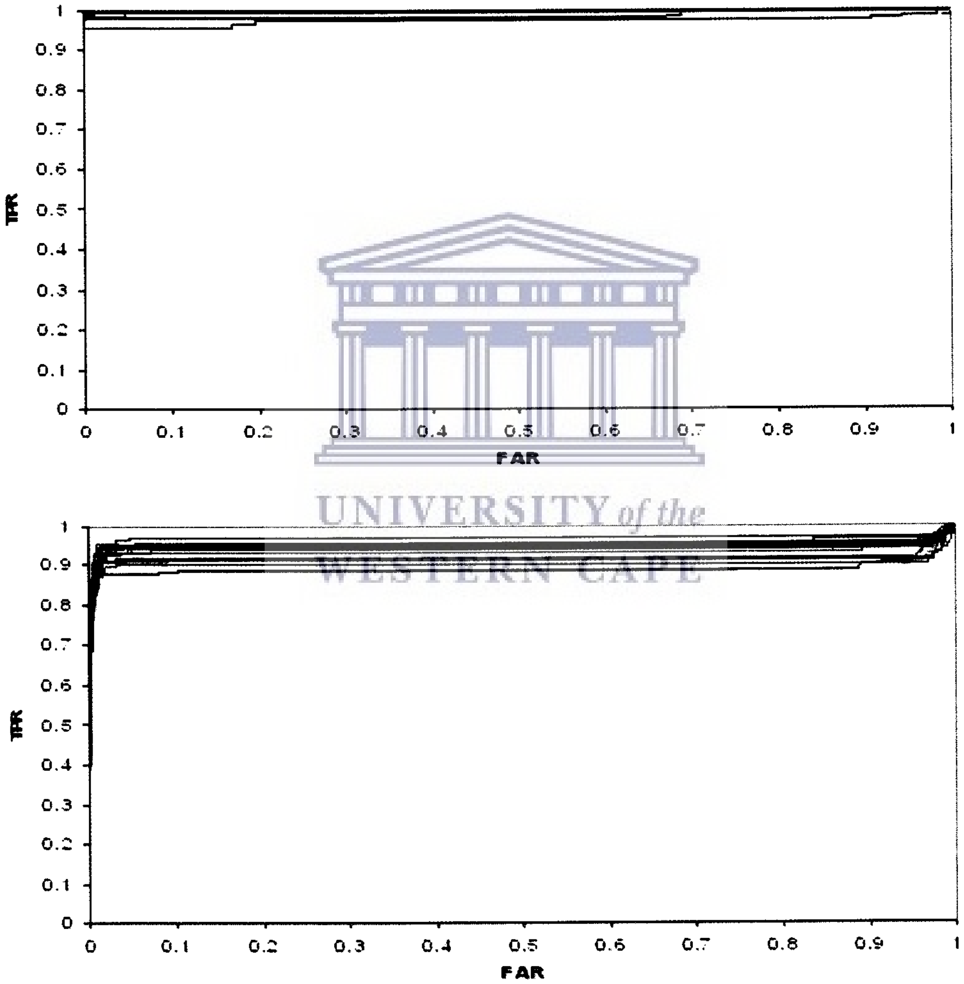

Figure 9-3: 30 ROC curves measuring FFNN fraud detection performance on the training set (top) and test set (bottom).

The ROC curves computed on the training set (top of Figure 9-3) are almost all completely square. This tells us that near errorless separation of the two classes were achieved during training. The curves computed on the test set (bottom of Figure 9-3) show less of a tendency to bow towards the (0,1) corner resulting in a smaller AUC value, which means the neural network performs somewhat worse on the test set as expected.

Table 9-1 shows the top 5 area-under-curve (AUC) values recorded during the training and test stages of the experiment respectively, including their corresponding root mean squared error (MSE) values, the minimum and average, and 95% AUC confidence interval calculated using the results of all 30 trials. For the test set, the maximum AUC recorded during the 30 repetitions of the experiment is 0.96745 and the minimum 0.88609. Keeping in mind that the data set used is not conducive to good machine learning performance, the FFNN actually performed quite well. Training times were short and generalisation with the network extremely quick, achieving a classification rate of approximately 46500 transactions/second.[4]

Rank #	**Training Set**		**Test Set**	
	AUC	**MSE**	**AUC**	**MSE**
1	1.00000	0.10550	0.96745	0.11543
2	0.99999	0.10093	0.95638	0.11311
3	0.99998	0.10125	0.95615	0.11265
4	0.99997	0.10094	0.95612	0.11915
5	0.99997	0.10113	0.95543	0.11764
Minimum	0.97143	0.10190	0.88609	0.12107
Average	0.99508	0.10145	0.93730	0.11626
95% Confidence Interval	0.99280 – 0.99735		0.93015 – 0.94444	

Table 9-1: Summary of training and test results for FFNN.

[4] Classification rates reported here are indicative. Classification rates were calculated on an Intel Celeron D 2.66 Ghz processor, and excluded pre-processing of the feature vectors.

Table 9-3 lists all results over the 30 training and 30 test trails, in the order in which they were recorded. As discussed in section 7.2, MSE is somewhat irrelevant when used in the context of heavily skewed data sets, be we include it here nevertheless for the sake of completeness.

Trail #	Training Set		Test Set	
	AUC	MSE	AUC	MSE
1	0.99991	0.10108	0.93743	0.11767
2	0.99987	0.10313	0.94426	0.11687
3	0.99999	0.10093	0.91333	0.11626
4	0.99313	0.10057	0.91402	0.11442
5	0.99361	0.10022	0.96745	0.11543
6	0.99992	0.09656	0.92031	0.11194
7	0.98633	0.10370	0.93755	0.11949
8	0.99998	0.10125	0.95006	0.11650
9	0.99312	0.10131	0.95638	0.11311
10	0.99316	0.10054	0.93790	0.11708
11	1.00000	0.10550	0.95001	0.11945
12	0.99317	0.10211	0.95543	0.11764
13	0.98645	0.10500	0.88609	0.12107
14	0.99997	0.10080	0.94355	0.11604
15	0.99997	0.10113	0.93278	0.11619
16	0.99315	0.10112	0.95026	0.11407
17	0.99313	0.10093	0.90104	0.11903
18	0.99989	0.09794	0.95481	0.11244
19	0.99996	0.10113	0.95014	0.11435
20	0.99325	0.10247	0.94903	0.11672
21	0.99938	0.10100	0.93815	0.11456
22	0.99355	0.10252	0.93255	0.11680
23	0.99936	0.10288	0.94342	0.11755
24	0.99320	0.10248	0.91404	0.11765
25	0.97143	0.10190	0.89038	0.11780
26	0.99997	0.10094	0.94929	0.11394
27	0.99994	0.10002	0.94903	0.11548
28	0.99382	0.10039	0.95615	0.11265
29	0.98384	0.10359	0.95612	0.11915
30	0.99981	0.10037	0.93792	0.11654

Table 9-2: Full list of training and test results for FFNN.

9.2.3. Support Vector Machines

The SVM has been around for a while and many tried and tested off-the-shelf packages are freely available with which SVM researchers can experiment. In this particular case, Chang and Lin's popular *libSVM* [8] implementation was used which includes a wide variety of tools to make dealing with SVM classification much simpler for the novice user.

Parameters and Training

To a great extent the performance of an SVM depends on what kernel is used and what the values of the kernel parameters and cost are set to. Since SVM training guarantees to always find the maximum margin achievable for a given set of parameters, it is fruitless to conduct multiple trial runs with the same parameters and time should rather be spent finding the most appropriate kernel and its parameters. Unfortunately, no real satisfying heuristic exists with which to compute a kernel's parameters and in most cases the best one can do is to guess. The search for the best parameters, however, can be done in a structured manner. It was already mentioned in Section 5.4 that a combination of cross-validation and grid-search is a popular method for obtaining a best guess estimate of the cost parameter (*C*) and the kernel parameter (γ) when dealing with radial basis function (RBF) kernels. Cross-validation can also help to prevent over fitting of the training set [15]. The cross-validation and grid-search methods were utilised here to obtain a best guess estimate of these parameters.

Cross-validation involves splitting the training set into two parts. Training is then done on one part and testing on the other. In *v*-fold cross-validation, the training set is divided into *v* parts and the classifier sequentially trained on *v*-1 of the subsets, and then tested on the remaining subset. This way, each subset is tested once and each training instance is predicted once, and the cross-validation accuracy is therefore the percentage of training examples correctly classified.

Grid-search is a straightforward and simple technique. To estimate the best parameters for a given classification problem and kernel, a growing sequence of parameter values are used for training and the ones giving the best cross-validation accuracy is picked. In case of the RBF kernel, the parameter values consist of (C, γ) pairs.

Cross-validation and grid-search were done using the parameter selection tool *grid.py* that comes packaged with *libSVM*. Figure 9-4 shows the output of *grid.py* as applied to fraud detection during one of the parameter selection runs.

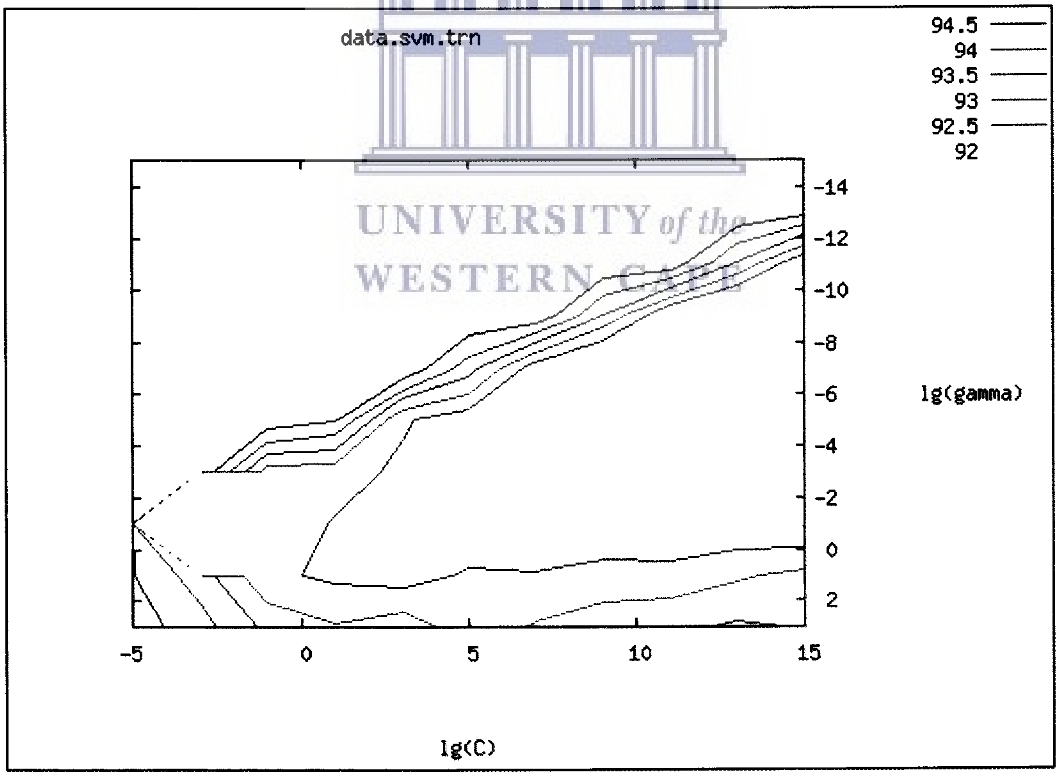

Figure 9-4: Grid-search results for SVM parameter selection.

Since the class distribution of the training set is skewed and biased towards the legitimate transaction class, the cost of misclassification amongst the two classes are different. On the one hand, one might want to cost the misclassification of fraudulent transactions more because

1. not detecting them directly translates into monetary loss, and
2. fraud transactions in the data set are sparsely represented.

On the other hand, one might also want to cost the misclassification of a legitimate transaction more since customer dissatisfaction can be more harmful than fraud in some cases. In cases where appropriate cost parameters are not introduced for skewed data sets, underfitting of the data is likely to occur. In such cases, the classifier will assign the legal transaction label to all transactions, making it impossible to detect any fraud. During the parameter grid-search for this experiment, different ratios of cost for the two classes were tried and the best cost ratio was found to be 1:10, with mistakes on the fraud class being penalised 10 times more than mistakes on the legitimate class.

The different kernel types tested included linear, polynomial, RBF, and sigmoid kernels.

Results

Figure 9-5 shows the ROC curves of various trial runs with different kernels, kernel parameters and cost ratios. The best generalisation performance on the test set was achieved by using an RBF kernel with $\gamma = 11$, cost $C = 0.05$ and cost ratio 1:10 as already mentioned.

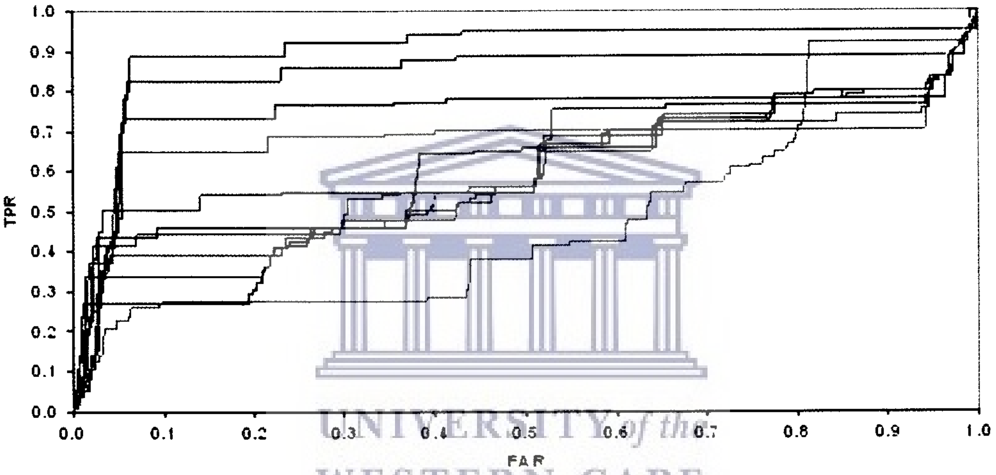

Figure 9-5: ROC curves for various SVM kernels, parameters, and cost.

The AUC for this curve was found to be 0.89322. The SVM achieved a classification rate of roughly 213 transaction/second.[5]

[5] This was calculated in a WIN32 environment, and according to [18] the Windows version of libSVM sometimes suffers inexplicable delays during training. This might also be true for classification.

9.3. Experiment II: Time Series Modelling with LSTM

A bespoke C/C++ implementation of LSTM (called *LSTMFraud*) was used in this experiment. The implementation is executable in both the WIN32 and Linux environments and was benchmarked against the Extended Reber Grammar problem [11] to ensure that it at least matches the performance of the original implementations of Sepp Hochreiter and Felix Gers used in the initial LSTM experiments [14][11]. The algorithm was implemented and customised rather than using one of the downloadable implementations because we felt that a much better understanding of the algorithm can be gained by actually implementing it.

All of the important parameters are customisable in *LSTMFraud*, for instance the number of memory blocks, memory cells per block, hidden neurons, output neurons, the learning rate, rate decay, and also whether input-output shortcuts and peephole connections should be used. Various test runs were executed in order to find the best combination of these parameters prior to executing the 30 trial runs of the experiment.

Network Topology, Parameters, and Training

The use of LSTM by nature results in a near fully connected network. Depending on implementation preference and parameter settings, almost every single output signal flows into the inputs of every single neuron entity present in the network. Figure 9-6 shows the LSTM topology used for this experiment. Two memory blocks with two cells each were used. Once again, not all connections can be shown due to reasons of intelligibility; for instance the below figure does not show feedback connections between cell outputs and memory block and cell inputs.

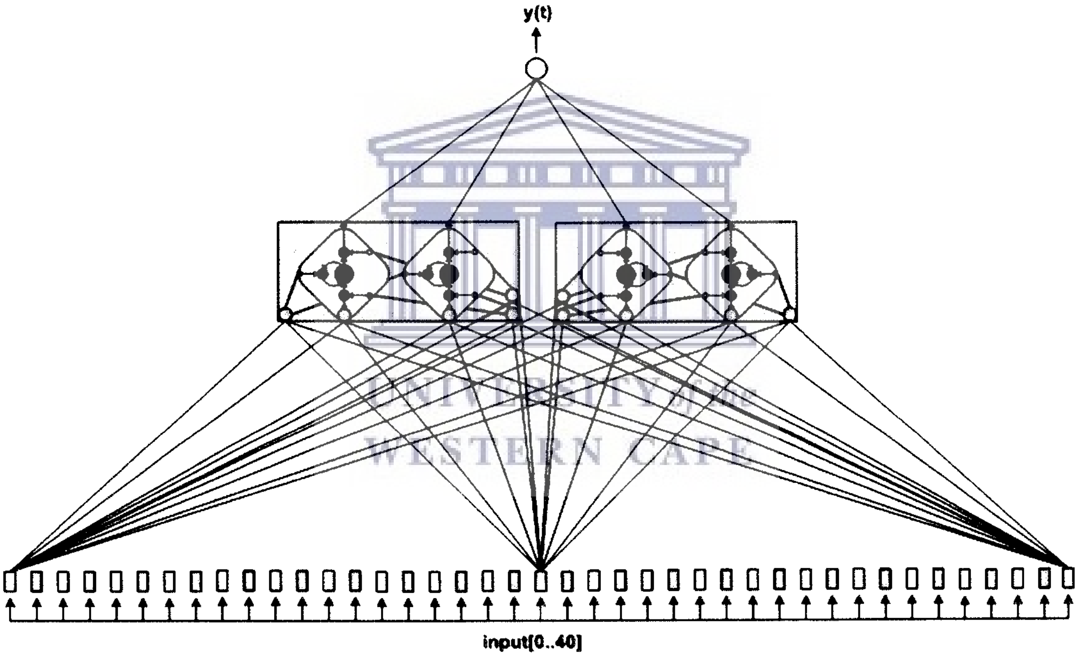

Figure 9-6: LSTM network topology for fraud detection.

Each trial run consisted of feeding pre-processed randomly selected transaction sequences into the network for 100 epochs, where an epoch is the size of the training set (in this case the training set contaned 16263 transactions). Once again, although LSTM does not seem to suffer as much from problems with overfitting as FFNNs do, 100 epochs were chosen because this resulted in the best generalisation performance during initial experimentation. The network was not manually reset between sequences or epochs, and no learning rate decay, peephole connections or input-output shortcuts were used. The learning rate for this expirement was set to 0.3. During training, whenever a misclassification occurred, training on the transaction sequence was immediately restarted. This was repeated until all transactions of a sequence were correctly classified, after which a new sequence was then randomly selected from the training set for the next iteration.

At the end of each training sequence, the fraud probability for each transaction in both the training and test set was recorded in order to draw the ROC curves and calculate the AUC and MSE. During generalisation testing, the network is reset after each transaction sequence. The transaction sequences used in both training and testing were of variable length.

A hybrid network with one feedforward hidden layer was also tested, but no real improvement in generalisation performance was noted (see Figure 9-7). The motivation for its use was that FFNN performed well on the data set, and that a combined FFNN-LSTM approach might lift the generalisation performance somewhat. The problem with this approach is that static FFNN neurons learn much quicker than their LSTM counterparts, and their weights will therefore start overfitting the training data long before the LSTM memory cells are fully trained. One possible solution to this problem is to have separate learning rates for the FFNN and LSTM parts of the hybrid network; however, it remains difficult to obtain the correct ratio between these two rates. The training method employed here, i.e. stopping the presentation of a sequence once a misclassification occurs and then re-presenting it, is not really applicable to FFNN training either, where it is better to present as much of the data set as possible during each epoch.

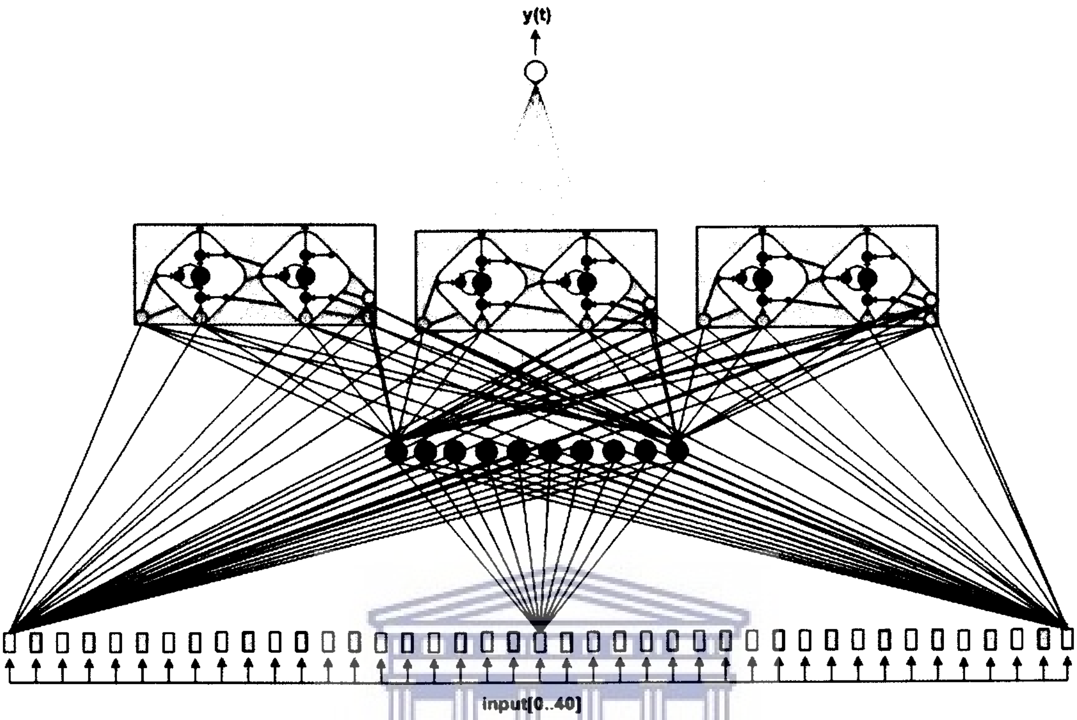

Figure 9-7: A hybrid LSTM with one hidden layer.

Results

Figure 9-8 shows the generalisation ROC curves of the 30 trials with LSTM. Their deviation from a straight diagonal between (0,0) and (1,1) and consequent tendency to bow towards the (0,1) corner of the graph once again shows that the algorithm actually learned something during training and that it performs far better than a constant "not fraud" diagnosis would, despite the overwhelming bias towards the legitimate transaction class in the test data set.

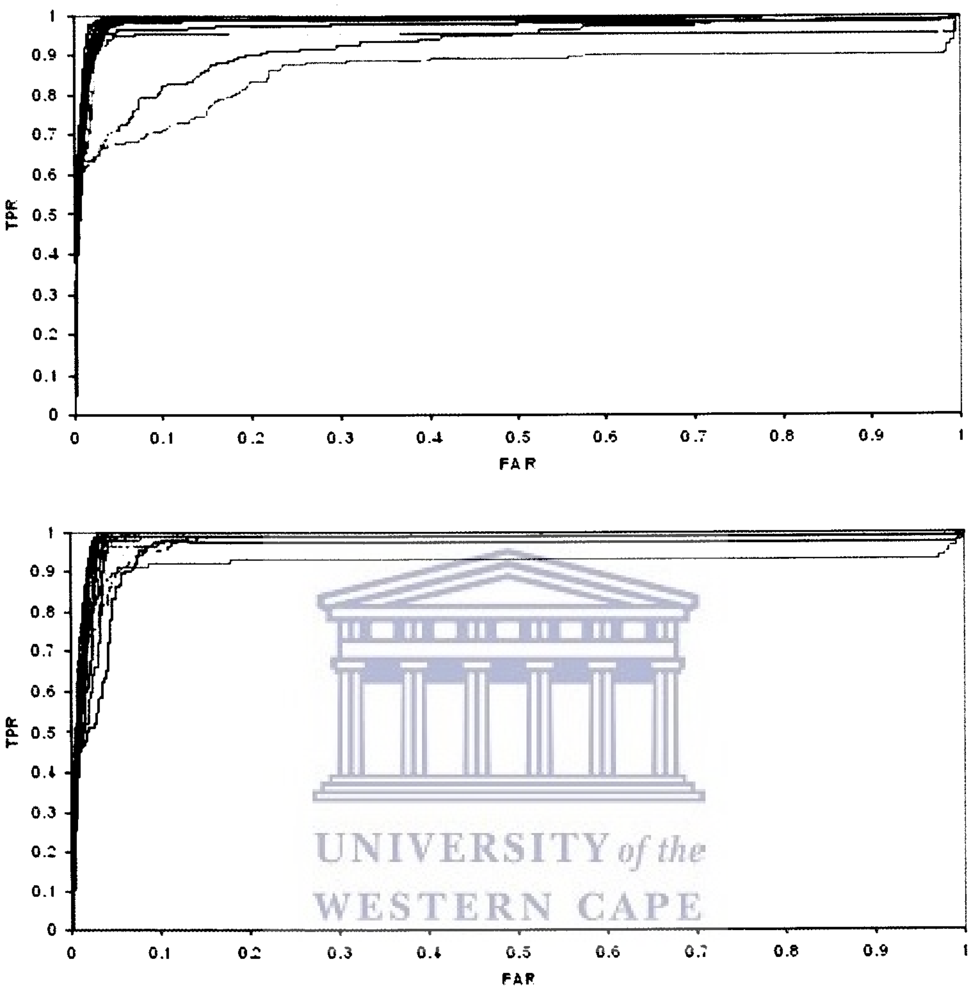

Figure 9-8: 30 ROC curves measuring LSTM fraud detection performance on the training set (top) and test set (bottom).

The spread of the ROC curves again shows the level of randomness inherent in neural network training where initial weight values are randomised leading to a different gradient search start position in weight space each time the network is re-initialised prior to training. In addition, the transaction sequences are also randomly picked from the training set for each iteration through the network.

Table 9-3 below shows the top 5 AUC values recorded during the training and testing stages of the LSTM time series experiment, including their corresponding MSE values, and also the minimum, average, and 95% AUC confidence interval calculated using the results of all 30 trials.

Rank #	**Training Set**		**Test Set**	
	AUC	**MSE**	**AUC**	**MSE**
1	0.99524	0.12660	0.99216	0.12886
2	0.99397	0.12969	0.99145	0.12802
3	0.99367	0.12749	0.99115	0.10950
4	0.99359	0.12725	0.99091	0.12435
5	0.99333	0.13107	0.99060	0.13729
Minimum	0.85868	0.12942	0.91903	0.12938
Average	0.97571	0.12873	0.98221	0.13128
95% Confidence Interval	0.96326 – 0.98817		0.97716 - 0.98727	

Table 9-3: Summary of training and test results for LSTM.

The maximum AUC recorded on the test set during the 30 trails is an amazing 0.99216, and the minimum 0.91903. Training time for LSTM is the slowest of the machine learning methods tested here, but classification is fairly quick with a recorded classification rate of approximately 2690 transactions/second.

Table 9-4 below lists all results for LSTM over the 30 training and 30 test trails, in the order in which they were recorded.

Trail #	Training Set		Test Set	
	AUC	MSE	AUC	MSE
1	0.99264	0.12684	0.99091	0.12435
2	0.99145	0.13085	0.98844	0.14216
3	0.99085	0.12265	0.98929	0.12163
4	0.99333	0.13107	0.98890	0.12571
5	0.94763	0.12837	0.96308	0.12674
6	0.98994	0.13235	0.98889	0.13450
7	0.92630	0.13210	0.97610	0.12552
8	0.98680	0.12702	0.98313	0.12830
9	0.99202	0.11228	0.99115	0.10950
10	0.98512	0.12873	0.98993	0.13407
11	0.96602	0.13294	0.98687	0.12673
12	0.99041	0.12514	0.97613	0.13880
13	0.99367	0.12749	0.98877	0.13715
14	0.95212	0.13233	0.98596	0.12686
15	0.98996	0.12883	0.98380	0.14431
16	0.98638	0.13916	0.96679	0.13910
17	0.99183	0.13011	0.99216	0.12886
18	0.99266	0.13376	0.98144	0.13506
19	0.85868	0.12942	0.91903	0.12938
20	0.86001	0.12564	0.97470	0.13521
21	0.99141	0.12703	0.98910	0.12236
22	0.99397	0.12969	0.99145	0.12802
23	0.99524	0.12660	0.98191	0.13336
24	0.98505	0.12755	0.98854	0.13442
25	0.98913	0.12851	0.99060	0.13729
26	0.99359	0.12725	0.98934	0.12809
27	0.97382	0.12941	0.96552	0.13253
28	0.98631	0.12788	0.98768	0.13090
29	0.99249	0.13030	0.98966	0.13696
30	0.99256	0.13066	0.98717	0.14065

Table 9-4: Full list of training and test results for LSTM.

9.4. Discussion

All three of the algorithms tested here achieved remarkably good separation between the legitimate and fraud classes in the test data set. Figure 9-9 shows the ROC curves of the best performing SVM, FFNN and LSTM instances and Table 9-5 lists the AUC values of these ROC curves, along with the classification rates for each algorithm.

As postulated, LSTM on time-ordered data outperformed the other two methods, a remarkable feat if one takes into account that both FFNN and SVM performed quite well too, with FFNN getting AUC values in the upper 0.96 range. What is surprising about the above results, though, is that FFNN actually outperformed SVM in this case. It is quite possible that the optimum SVM kernel and kernel parameters were not used for this particular data set; searching for these parameters is a time consuming task and we did the best we could with cross-validation and grid-search in the time available. Nevertheless, SVM still achieved a respectable AUC of 0.89322.

Training times were the shortest with FFNN, followed by SVM and LSTM. Classification was also the quickest with FFNN (46500 transactions/second), followed by LSTM (2690 transactions/second) and then SVM (213 transactions/second).

Method	**Max AUC**	**Average AUC**	**Classification Rate (transactions/sec)**
SVM	0.89322	n/a	213
FFNN	0.96745	0.93730	46500
LSTM	0.99216	0.98221	2690

Table 9-5: Comparison of SVM, FFNN and LSTM.

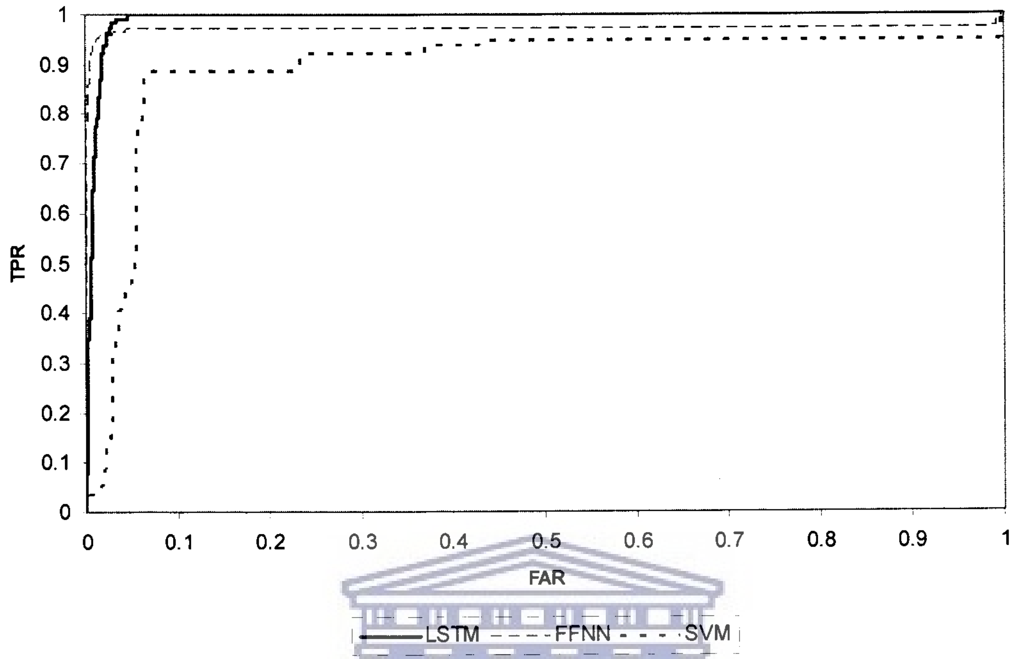

Figure 9-9: ROC curves of best performing LSTM, FFNN and SVM classifiers.

It is interesting to note that the average AUC computed for LSTM on the test set was actually higher than on the training set. Having a closer look at the full list of results (Table 9-4) one sees that in most cases performance was either better on the training set, or more or less the same as on the training set; for trails 7, 19 and 20, however, performance on the test set were better by such a large margin that it had an excessively positive effect on the average performance of the classifier. Significantly better performance on the test set in some instances are certainly unusual, but we will not investigate it here since we feel it will distract from the main objective of this thesis.

Another question that might arise is whether the difference in classification performance of the two closest competitors (FFNN and LSTM) are statistically significant. Without delving too deep into this subject, one can make two observations which certainly builds a case for statistical significance of the results obtained here:

1. The two AUC 95% confidence intervals calculated on the 30 test trails for FFNN and LSTM respectively, do not overlap.
2. Using the Student's t-test, we obtained t = -9.89380. The probability of obtaining this value by pure chance, was calculated as p = 4.60989e-014 ≈ 0. It is widely accepted that if this probability is less than 0.05, the difference in results is significant. In our case p was so close to zero that the observed difference between FFNN and LSTM classifier results can only be deemed very significant.

9.5. Summary

In this chapter, the two main methodologies for fraud detection proposed in this thesis were tested and compared using a data set containing real-world credit card transactions. Experiments were also conducted with a well known machine learning method, the feedforward neural network, to form a solid basis for comparison between the various algorithms. In the next and final chapter we draw some conclusions and discuss some possibilities for future research.

Chapter 10

CONCLUSIONS AND FUTURE RESEARCH

10.1. Conclusion

In this thesis, we set out to show what can be gained in generalisation performance by modelling the time series inherent in sequences of credit card transactions as opposed to dealing with individual transactions. To support these claims, we proposed two fraud transaction modelling methodologies for analytical comparison: support vector machines and long short-term memory recurrent neural networks. In addition, we also ran experiments with the well known feedforward neural network in order to lay a solid foundation for our comparative study.

The results of the experiments conducted in Chapter 9 largely confirmed our initial hypothesis, and LSTM proved to be a highly successful method for modelling time series. Our claim was further strengthened by the fact that the fraud detection systems in use today, use statistics such as the transaction velocity [23] and distance mapping techniques between successive transactions to aid them in discriminating legitimate from fraudulent transactions, which in essence introduce a concept of time and dynamics into otherwise static methodologies anyway. What makes the success of LSTM more remarkable is that the time series modelled here are of variable length and different for each card member. LSTM therefore modelled a set of completely diverse time series and still managed to outperform two very popular machine

BIBLIOGRAPHY

[1] Acton, F.S. (1959). Analysis of straight-line data. *Dover Publications* (1994).

[2] Bolton, R.J. & Hand, D.J. (2002). Statistical Fraud Detection: A Review. *Statistical Science,* **17(3)**, 235-255.

[3] Boser, B.E., Guyon, I.M. & Vapnik, V.N. (1992). A training algorithm for optimal margin classifiers. Haussler, D. (Ed). *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory,* 144-152.

[4] Brause, R., Langsdorf, T. & Hepp, M. (1999). Credit Card Fraud Detection by Adaptive Neural Data Mining. http://Citeseer.ist.edu/brause99credit.html.

[5] Burges, C.J.C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, **2**, 121-167.

[6] Card Fraud: The Facts. (2005). *The definitive guide on plastic card fraud and measures to prevent it.* APACS. http://www.apacs.org.uk

[7] Caruana, R. (2004). The PERF Performance Evaluation Code, http://kodiak.cs.cornell.edu/kddcup/software.html.

[8] Chang, C. & Lin, C. (2001). LIBSVM: a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[9] Cortes, C. & Vapnik, V. (1995). Support-Vector networks. *Machine Learning,* **20(3)**, 273-297.

[10] Elman, J.L. (1990). Finding structure in time. *Cognitive Science Journal.* **14(2),** 179-211.

[11] Gers, F.A., Schmidhuber, J. & Cummins, F. (1999). Learning to Forget: Continual Prediction with LSTM. *Technical Report IDSIA-01-99.*

[12] Gers, F.A., Schraudolph, N.N. & Schmidhuber, J. (2002). Learning Precise Timing with LSTM Recurrent Networks. *Journal of Machine Learning Research (2002)* **3**, 115 – 143.

[13] Harvey, L.O. Jr (2003). Detection Sensitivity and Response Bias. *Psychology of Perception.* Psychology 4165, Department of Psychology, University of Colorado.

[14] Hochreiter, S. & Schmidhuber, J. (1996). Long Short-Term Memory. *Technical Report FKI-207-95, Version 3.0.*

[15] Hsu, C., Chang, C. & Lin, C. (2003). A Practical Guide to Support Vector Classification. *Technical Report, Department of Computer Science and Information Engineering, National Taiwan University.*

[16] Jordan, M.I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 531-546.

[17] Keerthi, S.S. & Lin, C. (2003). Asymptotic Behaviors of Support Vector Machines with Gaussian Kernel. *Neural Computation (2003)* **15,** 1667-1689.

[18] Kroon, S. & Omlin, C.W. (2003). Getting to grips with Support Vector Machines: Application. *South African Statistical Journal (2004)* **28(2)**, 93-114.

[19] Kroon, S. & Omlin, C.W. (2004). Getting to grips with Support Vector Machines: Theory. *South African Statistical Journal (2004)* **28(2),** 159-172.

[20] Lee, Y., Lin, Y. & Wahba, G. (2001). Multicategory Support Vector Machines. *Technical Report TR1040, Department of Statistics, University of Wisconsin.*

[21] Maes, S., Tuyls, K., Vanschoenwinkel, B., & Manderick, B. (2002). Credit Card Fraud Detection Using Bayesian and Neural Networks. *Proceedings of the 1st International NAISO Congress on Neuro Fuzzy Technologies,* Havana, Cuba, 2002.

[22] Masters, T. (1993). Practical neural network recipes in C++. *Academic Press.*

[23] Mena, J. (2003). Investigative data mining for security and criminal detection. *Butterworth-Heinemann.*

[24] Mitchell, T.M. (1997). Machine learning. *MIT Press and The McGraw-Hill Companies, Inc.*

[25] Press, W.H., Teukolsky, S.A., Vetterling, W.T. & Flannery, B.P. (1992). Numerical recipes in C. *Cambridge University Press.*

[26] Robinson A.J. & Fallside, F. (1998). Static and dynamic error propagation networks with application to speech coding. *Neural Information Processing System,*. Anderson, D.Z. (Ed). American Institute of Physics.

[27] Williams, R.J. & Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. *Back-propagation: Theory, Architectures and Applications.*