Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF MATHEMATICS

Machine Learning and Artificial Neural Networks Applied to Forecasting Energy Commodity Prices

Author: Jonah Humphreys (CID: 01971479)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2020-2021

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Acknowledgements

Above all else I would like to thank my friends and family for their unrivalled support throughout the duration of the MSc course. They provided comfort and strength through a difficult year and for that I will always be grateful.

I would also like to thank my supervisors Dr Joaquín Narro and Dr Thomas Cass for their advise during the research project, providing new takes and suggestions from a vast pool of knowledge. Alongside the staff I am also incredibly appreciative for the resources Imperial College has provided, not only during the project but throughout entirety of the course.

Abstract

In this thesis we explore the use of support vector machines and neural networks in forecasting time series in commodities markets. We tackle the binary classification task of predicting the direction of day-ahead price changes in front month Brent Crude Oil futures contracts. Initially we begin by building theoretically intuitive models, utilising support vector machines with both unlagged and then later lagged features so as to explore the long term dependencies of the data. Following this, we develop more advanced feed-forward neural networks that focus on capturing long and short term memories from the time series. In short we implement recurrent neural networks with various combinations of simple, LSTM and GRU layers in order to capture the importance of both short and long-term memory. We assess the models based on their performance over unseen test data across the metrics 'Accuracy', 'F_1 Score', 'Precision' and 'Recall'. We find that the SVM models perform significantly better than all artificial neural networks, with the best performance of 62.1% accuracy coming from the model with the raw data with 1 day lag as it's feature set. The best feed-forward neural network model achieved accuracy of 58.4%, basic recurrent network achieved 53.5%, LSTM 54.1% and GRU performed the best outside of the SVM models achieving an accuracy of 58.7%.

Contents

1 Introduction **6**

2 Data **8**
- 2.1 Commodities . . . 8
- 2.2 Indices . . . 9
- 2.3 Foreign Exchange . . . 9
- 2.4 Weather Forecast . . . 9
- 2.5 Lagged data . . . 10
- 2.6 Cleaning and transforming . . . 10

3 Support Vector Machines for Classification **12**
- 3.1 Maximal Margin Classification . . . 12
- 3.2 Support Vector Classifier . . . 15
- 3.3 Support Vector Machines . . . 17
  - 3.3.1 Kernel Trick . . . 17
  - 3.3.2 Common Kernels . . . 19

4 A Deep Learning Approach **21**
- 4.1 Feed-forward Neural Network . . . 21
  - 4.1.1 Multilayer Perceptron . . . 21
  - 4.1.2 Activation Functions . . . 23
  - 4.1.3 Gradient Descent . . . 24
  - 4.1.4 Backpropagation . . . 26
- 4.2 Recurrent Neural Networks . . . 28
  - 4.2.1 Basic Recurrent Neural Network . . . 28
  - 4.2.2 Long Short-Term Memory . . . 32
  - 4.2.3 Gated Recurrent Unit . . . 34

5 Implementation and Results **37**
- 5.1 Implementation . . . 37
  - 5.1.1 K-fold Cross-validation . . . 38
  - 5.1.2 Hyperparameter Tuning . . . 39
  - 5.1.3 Choice of Features . . . 39

5.2 Evaluation 40
5.2.1 Performance Metrics 40
5.2.2 Support Vector Machines 43
5.2.3 Deep Neural Networks 46
5.2.4 Recurrent Neural Networks 48

6 Conclusion **53**

A Technical Proofs **55**
A.1 Proof of Proposition 4.1.2 55

Bibliography **59**

List of Figures

3.1 Examples of two suboptimal separating hyperplanes that perfectly bisect the two classes but with very small margins. 14
3.2 A graph of the optimal hyperplane, the *maximal margin hyperplane*, and the large margin between the decision boundary and the support vectors. 14
3.3 Four graphs showing the support vector classifier on a linearly inseparable dataset with the regularising parameter C at the different values $C = 1, 3, 10, 100$. 16
3.4 Different SVM kernels applied to non-linear data to obtain non-linear decision boundaries. A polynomial kernel of degree 3 (left) and an RBF kernel (right). Source [James et al., 2013] 18
3.5 An example of the effectiveness of the kernel trick by mapping 2-dimensional data into 3-dimensional space via a polynomial kernel, source [Jordan and Thibaux, 2004]. 19

4.1 A dense feed-forward neural network made up of a 4 neuron input layer, 2 hidden layers each with 7 neurons (plus a bias neuron) and a 3 neuron output layer. 22
4.2 Three of the most commonly used activation functions: the Sigmoid function, the tahn function and the ReLu function respectively. . . . 23
4.3 Graphs of the Sigmoid function, the tanh function and the ReLu function, respectively. 23
4.4 A 'rolled' visual of an RNN (left) and an 'unrolled' visual of an RNN showing the individual time steps within the network (right). Source [Education, 2020] 29
4.5 A many-to-one RNN visualised. Source [Education, 2020] 29
4.6 An LSTM cell 33
4.7 A GRU cell 35

5.1 An illustration of k-fold cross validation with $k = 5$. Source [Raschka, 2020] 38
5.2 A confusion matrix 40
5.3 'Raw data - 1 Day Lag' LinearSVC coef_ square weightings 45

List of Tables

5.1 Results of the naive model 43
5.2 Table of results for the 11 different SVM models 44
5.3 Table of hyperparameters for the 11 different SVM models 45
5.4 Table of results for the different feed-forward neural network models . 48
5.5 Table of results for the different recurrent neural network models . . . 51
5.6 Table of results for the different LSTM neural network models 52
5.7 Table of results for the different GRU neural network models 52

Chapter 1

Introduction

Let us start by introducing the definition of a *Futures Contract*. The CME Group state that "a futures contract is a legally binding agreement to buy or sell a standardized asset on a specific date or during a specific month"[1]. Trading futures over their respective underlying security brings with it many advantages including high liquidity, greater leverage and lower trading costs, to name but a few. As a result of the aforementioned benefits offered by trading futures, it is worth exploring the problem of forecasting the binary price change direction. Thus the aim of this research is to accurately predict whether the next day price change of the Brent Crude Oil Front Month Futures Contract will be an increase or decrease, decided on the present day. Machine learning and artificial intelligence have been incredibly popular in recent years, especially within financial institutions. Here are some sources [Waldow et al., 2021], [Hsu, 2011] that highlight the success of such techniques in futures markets in particular, justifying the choice of using such techniques in the research we propose.

Support vector machines are supervised learning models very commonly used in machine learning as a result of their robustness in classification and regression problems. In 1992, Vapnik et al. [Boser et al., 1992] proposed a method from which one is able to compute non-linear classifiers by employing the kernel trick to maximal-margin hyperplanes, an earlier piece of work by Vapnik and Chervonenkis in 1963. SVM's have been used in time series classification with great success since their inception, see [10., 2017] and [Samsudin et al., 2010], and are especially useful for classifying data with a large number of features and not especially large datasets. Considering the intersection of all the different datasets we will be using reduces the volume of data down significantly, support vector classifiers are an ideal starting point for the binary classification problem we face.

[1] https://www.cmegroup.com/education/courses/introduction-to-futures/definition-of-a-futures-contract.html

The natural progression from training a model with, in theory, one layer is to implement models where the number of layers is a tunable hyperparameter. From here we introduce the idea of Artificial Neural Networks (ANN) for classification. Artificial neural networks were first introduced in 1943, pioneered by McCulloch and Pitts [McCulloch and Pitts, 1943], so as to replicate the mechanics of neurons within the human body. The benefit of working with a model formulated with multiple layers is the ability to identify more intricate patterns in the data, for our purposes hopefully locating frequent trends that indicate an upwards or downwards price movement. However, there was a need for a model that specialises in processing sequential data, in particular time series, and thus the Recurrent Neural Network (RNN) was brought into fruition, first inspired by Rumelhart's work in 1986 [Rumelhart et al., 1986]. One problem RNN's suffer from is the vanishing/-exploding gradient problem, as discovered by Hochreiter [Hochreiter, 1991], and so variations of the RNN were born, namely the LSTM [Hochreiter and Schmidhuber, 1997] in 1997 and the GRU [Cho et al., 2014] in 2014. These adaptions allowed us to build models that are able to capture both short and long term memory, making for an attractive solution to the vanishing/exploding gradient problem.

The paper is structured as follows. Chapter 2 covers the choice in data used for the feature set in our machine learning and deep learning models, as well as the cleaning of the data and transformations that were made for the purposes of modelling. In Chapter 3 we introduce the theory behind support vector machines following the intuition behind them and the foundations that allow us to work towards constructing non-linear decision boundaries. Chapter 4 provides the same level of explanation, this time with regards to artificial neural networks in forecasting time series before moving onto recurrent neural networks and some variants of this type of network. Chapter 5 goes into detail about the implementation of the models and the evaluation metrics we will use to assess these models. Subsequently we will also state the results of these metrics and give some explanation as to why some models may have performed better than others. Finally, Chapter 6 concludes the paper, stating any further work that should be continued on the topic for future reference.

Chapter 2

Data

The primary data we will be using is the Brent Crude Oil Front Month Futures Contract close price. The *front month* refers to the contract with the nearest expiration date which, for oil, is a monthly occurrence. As the contract gets closer towards the expiration date, the underlying spot price and the price of the futures contract start to converge until the expiration date at which point they are equal for obvious reasons. Front month contracts tend to be the most actively traded for oil futures contracts and so the days leading up to the expiration date are highly volatile as traders try to make a profit on their purchases.

The feature set used in the commodity forecasting models need to be rich in information. What we ideally want is a feature set that covers as many aspects of the life cycle of crude oil as possible, from its extraction to the holding of it as a commodity, transportation of it as a commodity and it's uses both domestically and globally. In the following section we will highlight the reasons for using the features that have been chosen, those coming under the categories of: Commodities, Indices, Foreign Exchange, Weather Forecasts and Lagged data. We will then discuss the cleaning and transformation of the data in order for it to be usable in building and training our models.

2.1 Commodities

When it comes to individual commodities, in this case Brent Crude Oil, it can be very insightful to look at the behaviour and performance of other commodities in the same, or similar, sectors. Being purchased and used for related purposes suggests a demand for one commodity may also indicate a demand in others. Brent Crude Oil belonging to the energy commodities sector means for the purposes of the project we will be using other energy commodities, in particular Henry Hub Natural Gas, European Natural Gas, European Emissions, WTI Crude Oil, Rotterdam Coal and DE Power. To briefly justify the decision in using this data we calculate the pairwise

correlations of these different commodities focusing on the values obtained on the Brent Crude Oil axis. Given that the relationship is most likely non-linear, we still yield some high correlations between the commodities and Brent Crude Oil, namely 0.9907 (WTI Crude Oil), 0.9116 (European Natural Gas) and 0.7529 (Rotterdam Coal). We will also use US Crude Oil stock volume as an indication of supply and demand, and thus any price changes that will come as a result of this.

2.2 Indices

Because of the way crude oil is utilised both commercially and domestically, having a way to quantify how a country's economy is performing could also be a factor in determining what drives the price change in oil. Stock indices offer up a way of measuring how well a country is doing by keeping track of the top companies listed in their respective countries based on some metric, market capitalisation being the metric used by the most well-known indices. In the following research we will be using the FTSE 100, the CSI 300, the DAX and the S&P 500. Where raw data is used, we expect that stock indices will be a major factor in determining the direction of the change in price.

2.3 Foreign Exchange

With reasons similar to those made for the inclusion of stock index data being used, foreign exchange is also included because of it's correlation with commodities in general. A countries economic growth can be seen in how valuable it's currency is in comparison to other countries, development that can be driven by imports and exports of commodities. Two of the most closely studied and followed foreign exchange instruments to be used in this project will be EUR/USD and GBP/USD.

2.4 Weather Forecast

As a very simple example, clearly the daily temperature highs and lows are much lower in the winter months compared to the summer months. As a result, more oil is needed to warm homes, offices and buildings and so the demand for oil increases, leading to the price of oil rising. As another example, vast amounts of rainfall can be cause the activity of operational machinery either needing to stopped to prevent damage or possibly even damaged, all leading to a drop in supply and once again oil prices would then rise. The weather data we will be using includes daily temperature highs (°C), lows (°C) and amount of precipitation (centimetres) from Cologne Germany, New York USA and Beijing China.

2.5 Lagged data

While not initially included as it's own feature, in some of the models we also add the data from previous days for each of the features already mentioned so as to try and capture the causality of certain types of behaviour. Thus the number of days of lagged data included becomes it's own hyperparameter and one we experiment with. On one hand it may be beneficial to only have 1 or 2 days lag to allow for a model that generalises well and isn't overcrowded in terms of dimensionality. However there may also be trends that are captured by using 10 days worth of data that wouldn't be seen in only a small number of days. This is an easy set of features to include using well-known Python libraries such as *Pandas* (one of the most common libraries for dataframe manipulation).

2.6 Cleaning and transforming

Cleaning the data is an imperative step in ensuring the data fed into the model makes sense. For example, the Crude Oil stock volume was made up of weekly readings as opposed to the rest of the data which contained daily readings. In this case we decided the best way to fill in the missing 6 days worth of data every week was to simply set the next 6 days worth of data to a constant value, that value being the most recent weekly reading. Despite there being methods such as interpolation to fill in the missing data, we chose to use a constant value so that we weren't using values that may not yet exist between weeks.

Another transformation that had to be made was the shifting of some of the features, in particular the data for the S&P 500, CSI 300, EUR/USD and GBP/USD had to be shifted forward by a day. The reason for doing so was that the data used for these features was taken at their close, after the close of the Brent Crude Oil Futures contract. Because of the practical applications of the research we are exploring, i.e. trading the futures contract, information after the close of the security is no longer informative for the day we are trading on. However it would be fruitful for the following day. These transformations were made very easily via the *Pandas* library in Python.

Especially so in the building of the SVM models, the data was often *standardised*. That is, based on the training data, the mean μ and standard deviation σ were calculated and the transformation

$$z_i = \frac{x_i - \mu}{\sigma}$$

was made for each datum x_i in the training set, as in [Graves, 2012]. The two most popular scaling methods available to us in the scikit learn library were the StandardScaler()[1] (as described above) and the MinMaxScaler()[2], a scaling method where all of the data in transformed to be in the range $[0, 1]$. In a large majority of cases the StandardScaler() performed better than MinMaxScaler() and thus standardisation is the method we will be choosing throughout to scale our input values.

[1]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
[2]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

Chapter 3

Support Vector Machines for Classification

Support Vector Machines are both a very powerful and conceptually simple model used in Machine Learning classification problems, making them an industry favourite for classification tasks. The training complexity of SVM's is very high which makes them a suitable choice as an initial model based on the size of the datasets we intend to use, i.e. thousands of training observations.

In the following chapter we will begin by exploring the mechanics of Support Vector Machines as a Machine Learning classification tool, developing the idea of a separating hyperplane to allow some misclassifications and transforming the data through the kernel trick to work in higher dimensions.

3.1 Maximal Margin Classification

We begin by defining a hyperplane. In an m-dimensional space, a hyperplane is an $(m-1)$-dimensional affine subspace. As an example, a hyperplane in a 2-dimensional plane is a line, in 3-dimensional space a hyperplane is a plane and for higher dimensional spaces it is simply referred to as a hyperplane. Formally, in an m-dimensional space, the equation of a hyperplane is given by

$$w_1 x_1 + \ldots + w_m x_m + b = 0$$

or for brevity

$$\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = 0.$$

That is to say that any point $\boldsymbol{x} = (x_1, \ldots, x_m)^T$ lying on the hyperplane satisfies the above equation. Since a hyperplane has codimension of 1, we can think of a hyperplane as bisecting the whole ambient space leading naturally to the idea of

binary classes. These classes arise from

$$\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b > 0 \quad \text{and} \quad \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b < 0$$

which visually is seen as the point $\boldsymbol{x} = (x_1, \ldots, x_m)^T$ laying either one side of the hyperplane or the other. The side of the hyperplane the point is found is completely determined by the sign of $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$. From here, provided the data is *linearly separable*, we need only find a separating hyperplane that correctly separates the training data of one class from another. Then to classify any test data, for example $\boldsymbol{x}^* = (x_1^*, \ldots, x_m^*)^T$, we simply calculate the sign of $\langle \boldsymbol{w}, \boldsymbol{x}^* \rangle + b$ and classify the instance as either one of the classes based on this sign. Regardless of sign, a value of $|\langle \boldsymbol{w}, \boldsymbol{x}^* \rangle + b|$ far greater in magnitude than 0 indicates confidence in the classification whereas a value close to zero could mean some uncertainty due to how close to the separating hyperplane it is found.

The difficulty comes in finding the optimal separating hyperplane, if one exists at all. If one separating hyperplane can be found then in theory there exists infinite possible separating hyperplanes as a result of infinitesimal rotations and translations made to the original. In this case, it makes sense to find the hyperplane that puts the most distance between the hyperplane and the closest training instances to this hyperplane. This distance is called the *margin* and thus the hyperplane we are looking to calculate is called the *maximal margin hyperplane*. Since the hyperplane is 'supported' by the training observations closest to the hyperplane, in the sense that the largest margin is dependent only on these closest observations, the observations that support the hyperplane are called *support vectors* and will play a pivotal role in Support Vector Classification.

In Figure 3.1 we can see a dataset where the two classes, Class A and Class B, are clearly linearly separable. In this example the data has two features x_1 and x_2 so that we can easily visualise the data, classes and hyperplanes. The two black lines are examples of separating hyperplanes that perfectly separate the two classes, however they have very small margins with the hyperplanes being very close to the support vectors. In Figure 3.2 we see the maximal margin hyperplane that optimally separates the two classes. We can also see the margin by the dashed lines cutting through the support vectors.

We now give the details of the hard margin optimisation problem needed to be solved in order to find the optimal separating hyperplane. If we consider n training observations $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^m$ and class labels $y_1, \ldots, y_n \in \{-1, 1\}$ then the maximal

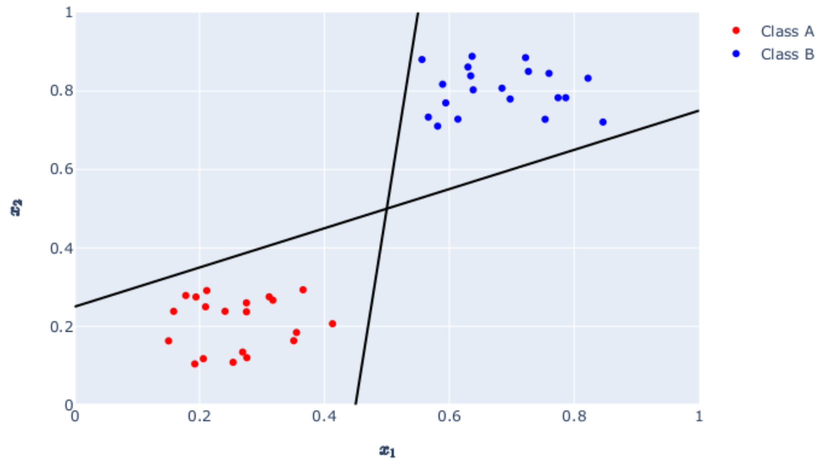

Figure 3.1: Examples of two suboptimal separating hyperplanes that perfectly bisect the two classes but with very small margins.

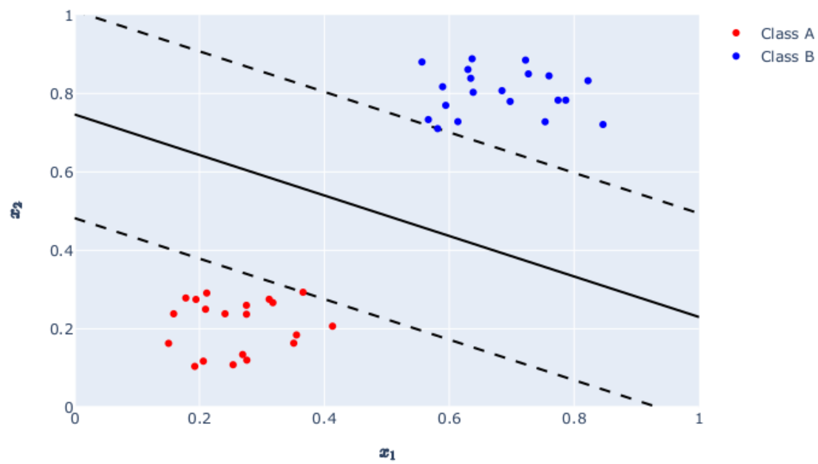

Figure 3.2: A graph of the optimal hyperplane, the *maximal margin hyperplane*, and the large margin between the decision boundary and the support vectors.

margin hyperplane can be found by solving

$$\begin{aligned} \min_{\boldsymbol{w},b} \quad & \frac{1}{2}\|\boldsymbol{w}\|_2^2 \\ \text{subject to} \quad & y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) \geq 1 \qquad \forall i = 1, \ldots, n \end{aligned}$$

Here we deconstruct the optimisation problem to make it slightly more intuitive. The norm of the feature weight vector, $\|\boldsymbol{w}\|_2$, corresponds to the slope of the decision function and so minimising $\|\boldsymbol{w}\|_2$ is equivalent to maximising the width of the margin. In the optimisation problem we aim to minimise $\frac{1}{2}\|\boldsymbol{w}\|^2$ since it has a more tractable derivative, i.e. $\nabla(\frac{1}{2}\|\boldsymbol{w}\|_2^2) = \boldsymbol{w}$, and minimising both $\|\boldsymbol{w}\|_2$ and $\|\boldsymbol{w}\|_2^2$ yield the same argmin. Since we require the hyperplane to be fitted such that there are no margin violations, the constraint that $y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) \geq 1$ ensures that all observations lie outside of the margin.

This maximum margin classification problem is a convex quadratic optimisation problem with linear constraints, also known as a Quadratic Programming problem, and as such there are plenty of pre-existing solvers for these types of problems.

3.2 Support Vector Classifier

A large assumption placed on the data when considering maximal margin classification is the ability to linearly separate the data into it's respective classes by a separating hyperplane. When the data gets noisier and starts to overlap we must develop the idea of a maximal margin classifier further so as to account for these overlaps. A tool used instead is a *soft margin classifier*, separating the data by a hyperplane as best it can while allowing for some misclassifications dependent on a tuning parameter. The added leniency to the maximal margin classifier is called a *support vector classifier* and is paramount in the development of support vector machines as a classification tool.

Even if the data is linearly separable, it may be advantageous to allow for some misclassifications in return for a model that generalises better for test observations. If, for example, a hyperplane was fit perfectly to the data but had a very small margin due to an extreme observation of one class, it might be better to allow this one extreme observation to be classified incorrectly to allow for a larger margin and thus a greater confidence in future classifications. If we were to allow such a small margin, the model might begin to overfit to the data and would be far too sensitive to individual observations to making accurate predictions.

Figure 3.3 shows the support vector classifier for 4 different values of the regu-

larising parameter C. The values were chosen so as to give the best visualisation of how varying C can increase/decrease the margin and alter the hyperplane itself completely. The regularisation parameter C works to compromise the width of the margin against the number of margin violations. In short, smaller values of C allow for a larger margin and thus more misclassifications whereas larger values of C reduce the number of misclassifications and therefore we are left with a much smaller margin.

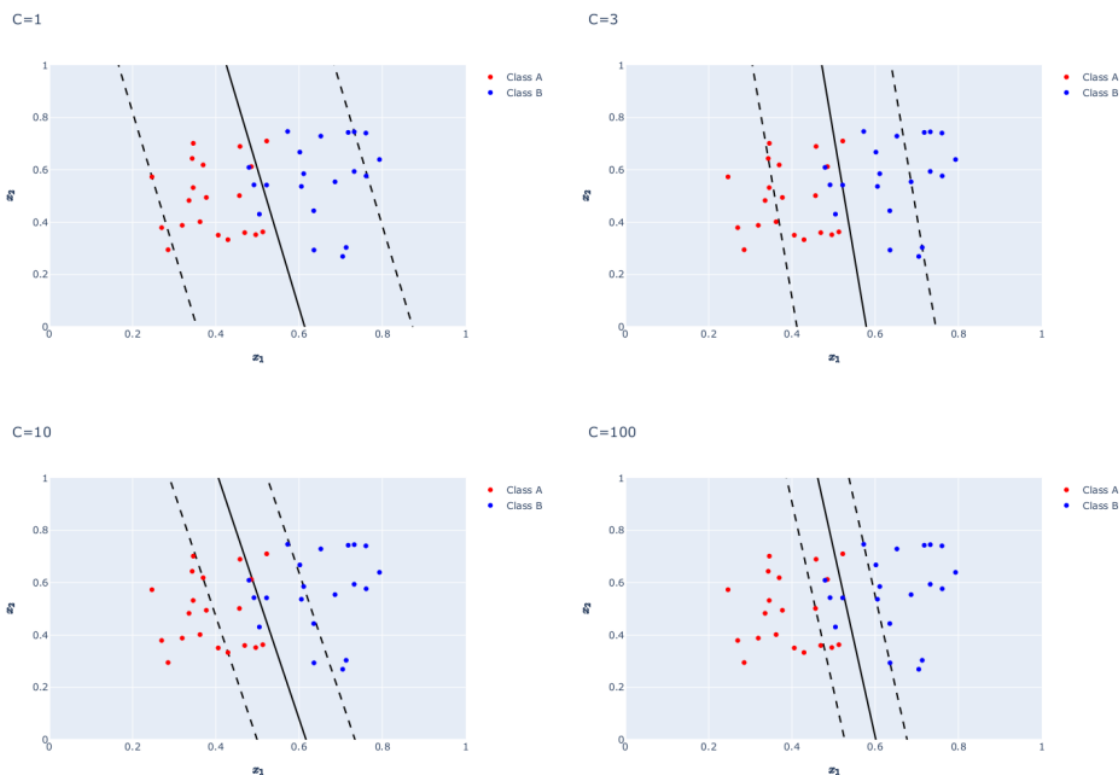

Figure 3.3: Four graphs showing the support vector classifier on a linearly inseparable dataset with the regularising parameter C at the different values $C = 1, 3, 10, 100$.

We now state the soft margin optimisation problem, which is to find the hyperplane

that solves

$$\begin{aligned} \min_{\boldsymbol{w}, b, \epsilon_1, \ldots, \epsilon_n} \quad & \frac{1}{2}\|\boldsymbol{w}\|_2^2 + C\sum_{i=1}^{n}\epsilon_i & \\ \text{subject to} \quad & y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) \geq 1 - \epsilon_i & \forall i = 1, \ldots, n \\ & \epsilon_i \geq 0 & \forall i = 1, \ldots, n \end{aligned}$$

with C a non-negative regularisation parameter and $\epsilon_1, \ldots, \epsilon_n$ our *slack variables*. Each of the slack variables ϵ_i quantifies how much the ith observation can violate the margin. Once again, let us deconstruct the optimisation problem to make it more intuitive. Similarly to the hard margin optimisation problem we aim to minimise $\|\boldsymbol{w}\|_2^2$ however this time we have the added factor of $C\sum_{i=1}^{n}\epsilon_i$. Minimising the summation of the slack variables accounts to reducing the number of margin violations while minimising $\|\boldsymbol{w}\|_2^2$, similar to the hard margin optimisation problem, aims to increase the margin. This is where C acts as a regularisation parameter, allowing us to control how much we want either factor to influence the whole objective function. We can see from the constraints that a value of $\epsilon_i = 0$ means the observation is on the correct side of the margin, $\epsilon_i > 0$ means the observation lies within the margin on the correct side of the hyperplane and $\epsilon_i > 1$ indicates a misclassification.

Much like the hard margin optimisation problem, the soft margin optimisation problem is also a quadratic programming problem and thus can be solved very easily by some pre-existing QP solvers.

3.3 Support Vector Machines

But we are still working under a rather constraining assumption; the idea that it is a linear decision boundary that separates the two classes in our binary classification problem. It is clear from Figure 3.4 that we will most likely be in need of a non-linear decision boundary which is where support vector machines come to be extremely useful. Support vector machines utilise a well known method known as the *kernel trick*, applied to the aforementioned support vector classifier, to map observations to a higher, possibly infinite, dimensional feature space. As a result, we're able to find a linear decision boundary in this higher dimensional space and translate this to a non-linear decision boundary in the original space, capturing patterns that may not have been possible using only a linear decision boundary on the initial dataset.

3.3.1 Kernel Trick

If we consider the support vector classifier dual problem, as seen in [Shashua, 2009], we note that only the inner product of the training observations $x_i, x_j \in \mathcal{X}$, $x_i^T x_j =$

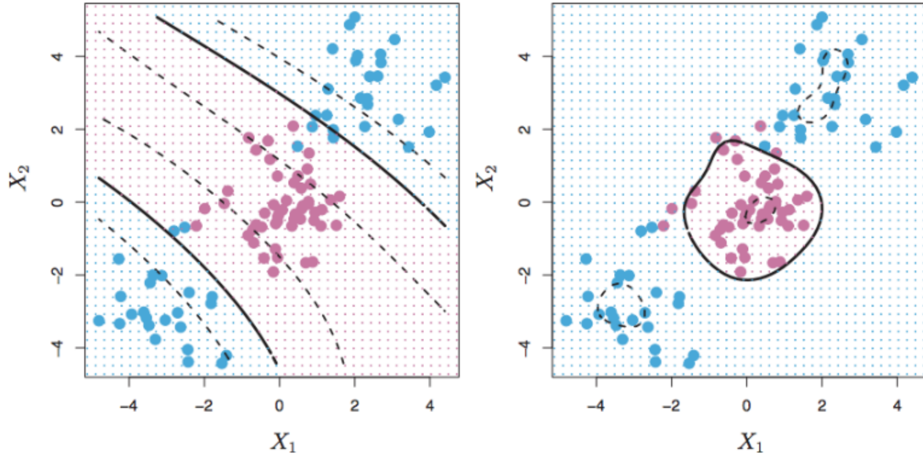

Figure 3.4: Different SVM kernels applied to non-linear data to obtain non-linear decision boundaries. A polynomial kernel of degree 3 (left) and an RBF kernel (right). Source [James et al., 2013]

$\langle x_i, x_j \rangle$, is used in the minimisation problem and thus we look for a *kernel function* $K(x_i, x_j)$ that can be expressed as an inner product in another, higher dimensional space $\mathcal{H}$. If we can find such a function $K(x, y) = \langle \phi(x), \phi(y) \rangle_H$ for our feature map $\phi : \mathcal{X} \to \mathcal{H}$ then finding a non-linear decision boundary becomes significantly less computationally complex.

Due to Mercer's theorem, we do not need an explicit representation for the feature map ϕ, provided our higher dimensional space $\mathcal{H}$ is an inner product space. Mercer's theorem states that ϕ exists whenever the input space $\mathcal{X}$ is provided with a measure ensuring the kernel function K satisfies Mercer's condition.

Definition 3.3.1 (Mercer's Condition)**.** A real-valued function $K(x, y)$ is said to fulfil Mercer's condition if for all square-integrable functions $g(x)$ one has

$$\int g(x)K(x, y)g(y)\, dx\, dy \geq 0.$$

However, in practice, kernel functions do not have to strictly satisfy Mercer's condition in order to perform well. If the Gram matrix[1] of the empirical training data is found to be positive semi-definite for some parameter values, one would still find that the training would converge successfully.

To make things clearer, the following is a well-known example to illustrate the

[1]The matrix whose (i, j)-th element is $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ where $K(\cdot, \cdot)$ is the kernel and $\boldsymbol{x}_i, \boldsymbol{x}_j$ are the i-th and j-th observations, respectively.

effectiveness of the kernel trick.

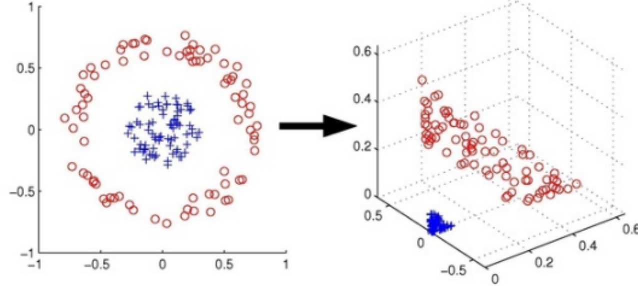

Figure 3.5: An example of the effectiveness of the kernel trick by mapping 2-dimensional data into 3-dimensional space via a polynomial kernel, source [Jordan and Thibaux, 2004].

Clearly in Figure 3.5 the 2-dimensional data on the left is not linearly separable and thus we require a non-linear decision boundary to separate the two classes. After a transformation the two classes are easily separable by a hyperplane in the higher dimensional space; in this example, 3-dimensional space.

Let $\phi : \mathbb{R}^2 \to \mathbb{R}^3$; $\phi(x_1, x_2) \coloneqq (x_1^2, \sqrt{2}x_1x_2, x_2^2)$ be our feature map, noting that the transformed vector is now 3-dimensional instead of 2-dimensional. The use of $\sqrt{2}$ here is solely to allow for a cleaner output. This gives us a non-linear decision boundary $\langle \boldsymbol{w}, \phi(\boldsymbol{x})\rangle + b = w_1x_1^2 + w_2\sqrt{2}x_1x_2 + w_3x_2^2 + b = 0$ in the new feature space. Finding the inner product of the transformed observations gives

$$\begin{aligned}
\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{y})\rangle &= \langle (x_1^2, \sqrt{2}x_1x_2, x_2^2), (y_1^2, \sqrt{2}y_1y_2, y_2^2)\rangle \\
&= x_1^2y_1^2 + 2x_1x_2y_1y_2 + x_2^2y_2^2 \\
&= \langle (x_1, x_2), (y_1, y_2)\rangle^2 \\
&= \langle \boldsymbol{x}, \boldsymbol{y}\rangle^2
\end{aligned}$$

so we have our kernel function $K(\boldsymbol{x}, \boldsymbol{y}) = \langle \boldsymbol{x}, \boldsymbol{y}\rangle^2$.

3.3.2 Common Kernels

Here we list some of the most frequently used kernel functions as defined in [Patle and Chouhan, 2013] elaborating on why they are chosen and their unique advantages.

Euclidean Kernel

$$K(\boldsymbol{x}, \boldsymbol{y}) = \langle \boldsymbol{x}, \boldsymbol{y}\rangle = \boldsymbol{x}^T\boldsymbol{y}$$

The euclidean kernel is what one would use in a regular support vector classifier; it is the standard inner product of the observations in the input space. This kernel compares how close a pair of observations are to each other via Pearson correlation.

Gaussian RBF Kernel

$$K(\boldsymbol{x}, \boldsymbol{y}) = \exp(-\gamma \cdot \|\boldsymbol{x} - \boldsymbol{y}\|^2), \quad \gamma > 0$$

The RBF kernel is one of the most commonly used kernels in non-linear support vector machine problems. It can be shown that as $\gamma \to 0$, an SVM with the RBF kernel approaches a linear SVM. Therefore the use of the RBF kernel, with an appropriate choice of γ, performs at least as well as a linear SVM. Here the feature space is infinite-dimensional thus it would be infeasible to work simply in just an enlarged input feature space. Using a kernel function is essential in order to simplify computational complexity. Clearly the Gaussian RBF kernel ranges between 0 and 1 and it's output decreases as the distance between observations increases. These properties mean, like all kernels, the function is seem as a measure of similarity between observations.

Polynomial Kernel

$$K(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}^T \boldsymbol{y} + 1)^d, \quad c \geq 0, \ d \in \mathbb{N}$$

We have already seen an example of a polynomial kernel of degree 2 being used in Figure 3.5. For $c > 0$, the kernel is called inhomogeneous whereas the kernel is known as homogeneous for $c = 0$. The free parameter c controls how much the higher order terms are weighted in the polynomial. The polynomial kernel gives a clear view into the computational complexity saved by using kernels. The feature map for a polynomial kernel of order d includes all monomials of degree up to d, giving a feature space of size $\binom{n+d}{d}$ and thus a map of complexity $\mathcal{O}(n^d)$ for n-dimensional input data. The kernel polynomial, however, is only of complexity $\mathcal{O}(n)$ and is therefore much more computationally efficient.

Sigmoid Kernel

$$K(\boldsymbol{x}, \boldsymbol{y}) = \tanh(\gamma \cdot \boldsymbol{x}^T \boldsymbol{y} + r), \quad \gamma, c \in \mathbb{R}$$

The Sigmoid kernel is unique in that the Gram matrix is not always necessarily positive semi-definite for all values of γ and c. However for the values of γ and c that do output a positive semi-definite Gram matrix, the sigmoid kernel is a function that takes inspiration from the common artificial neural networks activation function.

Chapter 4

A Deep Learning Approach

So far we have considered models with only one layer so it seems only natural for us to develop this idea further by introducing a class of models, deep neural networks, that benefit from having multiple layers within the model. As we're also dealing with time series data, capturing long term dependencies in the data is crucial in identifying signals and thus we expect recurrent neural networks, with the addition of LSTM and GRU layers, to be perfectly suited models for the problem we face.

4.1 Feed-forward Neural Network

We begin by first describing the structure of a feed-forward neural network before diving further into the more specific details of the individual neurons and the training of the overall model.

4.1.1 Multilayer Perceptron

A feed-forward neural network is a multilayer perceptron in which the signal flows in only one direction, i.e. from the inputs to the outputs. The network is built up of *neurons*, or more recently referred to as *units*, arranged into various layers. These layers include an *input layer*, an *output layer* and one or more *hidden layers*. Every layer, excluding the output layer, also includes a *bias neuron* which is fully connected to every neuron. In Figure 4.1 we can see a simple example of a *dense* feed-forward neural network, named as such since all the layers are fully connected, i.e. every neuron is connected to all neurons in the layers immediately before and after it. Here we have an input layer with 4 neurons (the input data has 4 features), 2 hidden layers both made up of 7 neurons (along with a bias neuron) and an output layer of 3 neurons. Using notation from [Pakkanen, 2020], this neural network would be the function $\boldsymbol{f_\theta} \in \mathcal{N}_3(4, 7, 7, 3; \boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2, \boldsymbol{\sigma}_3)$ with activation functions $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2, \boldsymbol{\sigma}_3$.

Now we will zoom into the model, specifically looking closer at the structure of

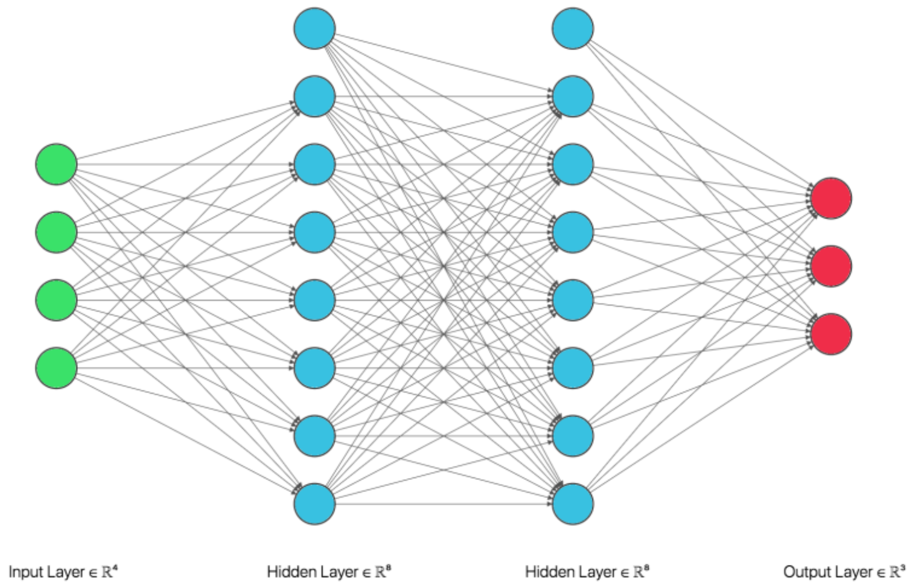

Figure 4.1: A dense feed-forward neural network made up of a 4 neuron input layer, 2 hidden layers each with 7 neurons (plus a bias neuron) and a 3 neuron output layer.

a neuron. Each neuron is equipped with an *activation function* in which a weighted sum of the inputs is fed into the neuron and an output is calculated from this activation function applied to the weighted sum. The weights of the inputs are subject to change through the training of the model via a method called *backpropagation* which will be discussed in further detail shortly. The output of a fully connected neuron with activation function ϕ can be expressed by the following equation

$$y_{\boldsymbol{w},b}(\boldsymbol{x}) = \phi(\boldsymbol{x}^T \boldsymbol{w} + b) \tag{4.1.1}$$

where $\boldsymbol{x}$ is the input vector, $\boldsymbol{w}$ is the weight vector made up of all the connection weights and b is the weight of the bias neuron. If we are working with a dense neural network and the ith layer of the network has n_i neurons in the previous layer then both $\boldsymbol{x}$ and $\boldsymbol{w}$ are n_i-dimensional vectors. These outputs are then passed to the next layer of the network, repeating until the final layer, the output layer, is reached in which an activation function specific to the type of problem being solved is used for the final output.

4.1.2 Activation Functions

There are many activation functions to choose from, with each activation function having it's own benefits and drawbacks. Here we will give the details of three of the most commonly used activation functions, two of which we will be using in our networks.

$$\phi_{\text{Sigmoid}}(x) = \frac{1}{1+\exp(-x)} \tag{4.1.2}$$

$$\phi_{\tanh}(x) = \tanh(x) \tag{4.1.3}$$

$$\phi_{\text{ReLu}}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{4.1.4}$$

Figure 4.2: Three of the most commonly used activation functions: the Sigmoid function, the tahn function and the ReLu function respectively.

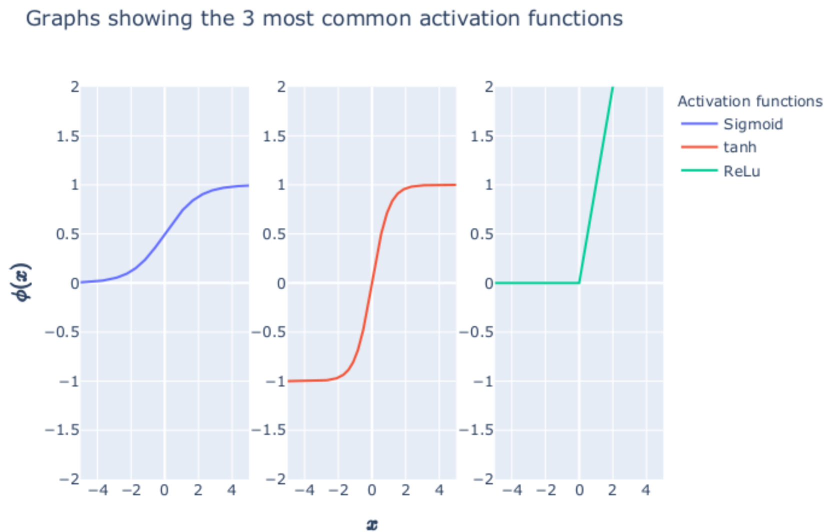

Figure 4.3: Graphs of the Sigmoid function, the tanh function and the ReLu function, respectively.

Let us start with the Sigmoid function given in (4.1.2). The Sigmoid function is extremely useful for binary classification models given the output of the function ranges between 0 and 1 and thus we will choose it to be the activation function used in the final output neuron of our models. Furthermore, the Sigmoid function

is a non-linear continuously differentiable function which allows the model to capture non-linear behaviour. The composition of multiple linear functions results in another linear function so including a non-linear function allows us to move outside the realms of linear behaviour. However, the gradient of the function tends to 0 as $x \to \pm\infty$ and subsequently the model may stop learning given that the tuning of the connection weights is based on the derivative of the outputs of neurons, i.e. we may be left with some 'dead' neurons. Further, the function is not symmetric about 0 and so the output of the function will be of the same sign as the input which is not beneficial for a gradient based learning algorithm.

The tanh function is given in (4.1.3) and has all the benefits of the Sigmoid function while also being symmetric about 0, as seen in Figure 4.3. This allows for the sign of the output to differ from the sign of the input and as a result the tanh function is better suited to gradient based learning algorithms and is often preferred over the Sigmoid function. Unfortunately the tanh function also suffers from the vanishing gradient problem where the gradient is extremely small for values of x that tend towards infinity, leaving some neurons essentially dead.

The ReLu function (Rectified Linear Unit) [Nair and Hinton, 2010], as defined in (4.1.4), is one of the most commonly used activation functions in the hidden layers and will be the activation function we choose to use in our models. The ReLu function is much more computationally efficient due to the nullification of all negative inputs, thus only activating some neurons at any given time. We do fall into the same problem as the previously mentioned activation functions in that for negative inputs the gradient is zero and so some neurons are not updated during the learning of the model. There are modifications of the ReLu function, such as the Leaky ReLu [Maas, 2013] and the ELU [Clevert et al., 2016] functions, that attempt to solve the issue of the the null gradient for negative input values.

4.1.3 Gradient Descent

The overall algorithm we will be using in order to minimise the objective function for our classification problem will be the Gradient Descent algorithm. Gradient descent is built on the idea that, for a multivariate function $F(\boldsymbol{x})$ differentiable in a neighbourhood about a point $\widetilde{\boldsymbol{x}}$, then the function decreases fastest in the direction of the negative gradient of F at the point $\widetilde{\boldsymbol{x}}$, i.e. $-\nabla F(\widetilde{\boldsymbol{x}})$.

Of course for more tractable functions the standard method would be to find the gradient of the function ∇F, solve $\nabla F(\boldsymbol{x}) = 0$ for $\boldsymbol{x} \in \text{dom}(F)$ and verify which of the solutions is the minimum point via the Hessian matrix. The problem for less tractable functions is ∇F may be difficult to compute or one may find it

even more arduous to solve $\nabla F(\boldsymbol{x}) = 0$. Instead we consider the *gradient flow* of F [Santambrogio, 2016], defined by the following equation

$$\frac{d\boldsymbol{x}(t)}{dt} = -\nabla F(\boldsymbol{x}(t)), \quad t > 0, \quad \text{initial condition } \boldsymbol{x}_0 \in \mathbb{R}^d.$$

For practical purposes we need to discretise the equation and thus, for learning rate $\eta > 0$, we obtain

$$\begin{aligned} \frac{\boldsymbol{x}(t+\eta) - \boldsymbol{x}(t)}{\eta} &\approx -\nabla F(\boldsymbol{x}(t), \quad t > 0 \\ \implies \boldsymbol{x}(t+\eta) &\approx \boldsymbol{x}(t) - \eta \nabla F(\boldsymbol{x}(t), \quad t > 0 \end{aligned}$$

from which we acquire our iterative gradient descent method

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta \nabla F(\boldsymbol{x}_n)$$

with initial condition $\boldsymbol{x}_0$.

A big problem with regular gradient descent for training neural networks is that calculating the gradient of the objective function can take a lot of time and space. Furthermore, we may not even want to find the unique minimiser of the objective function for fear of overfitting the network to the data. This is where we introduce *Stochastic Gradient Descent*.

Suppose our training data is indexed by $\{1, \ldots, n\}$ and we split the data into k, ideally equal sized, disjoint *minibatches* $B_1, \ldots, B_k \subset \{1, \ldots, n\}$. The process of stochastic gradient descent is simply updating the network's parameter vector $\boldsymbol{\theta}$ through

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_{B_i}(\boldsymbol{\theta}_{i-1}), \quad i = 1, \ldots, k$$

with initial condition $\boldsymbol{\theta}_0$. Here, $\mathcal{L}_{B_i}(\boldsymbol{\theta})$ is the empirical risk associated with the minibatch B_i, defined to be

$$\mathcal{L}_B(\boldsymbol{\theta}) := \frac{1}{|B|} \sum_{i \in B} \ell(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)}).$$

A complete run through the training dataset is called an *epoch* where multiple epochs are used for the training of the network, updating the parameter vector through every pass.

We are now ready to apply a gradient descent algorithm for the training of our neural networks, however we need a method in order to calculate the partial derivates of the objective function. This leads us naturally onto the next section: Backpropagtion.

4.1.4 Backpropagation

Backpropagation is one of the most popular methods by which the weights of the connections between layers of neurons are fine-tuned so as to reach an optimally performing model according to a particular choice in cost/loss function.

Let us first run through a general description of the overall process before describing in more detail the mathematics of how backpropagation works. The first stage of the process is called the *forward pass* in which mini-batches are passed through the network one at a time until the final output from the output layer is computed. Here the hidden layers have been initialised with random connection weights so as to break the symmetry of the network. The network then computes the loss function specific to the problem being solved. All intermediary results throughout the network, i.e. outputs from all layers, are saved for the *backward pass*. For the binary classification problem we will be tackling, the binary cross-entropy loss function is the most appropriate to use, given by the following equation

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \frac{1}{N}\sum_{i=1}^{N}\ell(\boldsymbol{f_\theta}(\boldsymbol{x}^{(i)}), y^{(i)}) \\ &= -\frac{1}{N}\sum_{i=1}^{N} y^{(i)}\log(\hat{y}_i) + (1-y^{(i)})\log(1-\hat{y}_i).\end{aligned}$$

where $y^{(i)}$ is the actual class of the i-th observation, taking values $y^{(i)} \in \{0,1\}$, and $\hat{y}_i \in (0,1)$ is the calculated probability that the i-th observation belongs to the positive class (the positive class taking the label $y^{(i)} = 1$).

From here the chain rule is applied to calculate how much each of the saved intermediary values contributes to the loss function, passing from one layer to the one before it until the input layer is reached. The error gradients are then used to execute a gradient descent in order to fine-tune all the connection weights. Let us lay out this process more rigorously below. For the purposes of brevity we will be referencing the Imperial College London Deep Learning 2020 Lecture Notes from the MSc Mathematics and Finance Course [Pakkanen, 2020], borrowing the notation used in the notes to clearly describe the process of backpropagation.

Let $\boldsymbol{f_\theta} \in \mathcal{N}_r(I, d_1, \ldots, d_{r-1}, O, \boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_r)$ be our neural network with hyperparameters $\boldsymbol{\theta} = (W^1, \ldots, W^r; \boldsymbol{b}^1, \ldots, \boldsymbol{b}^r)$, activation functions $\boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_r$ and batch B. We assume for simplicity that $\boldsymbol{\sigma}_i$ is the component-wise application of the one-dimensional activation function $g_i \colon \mathbb{R} \to \mathbb{R}$ denoted by $\boldsymbol{g}_i$ for $i = 1, \ldots, r$ and thus $\boldsymbol{g}_i' = (g_i', \ldots, g_i') \colon \mathbb{R}^{d_i} \to \mathbb{R}^{d_i}$ is the component-wise derivative of $\boldsymbol{g}_i$.

We aim to find the minimum of our cost function via gradient descent which means we need to find

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_B(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \left(\frac{1}{|B|} \sum_{i \in B} \ell(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}) \right),$$

however, thanks to linearity, this amounts to finding

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_B(\boldsymbol{\theta}) = \frac{1}{|B|} \sum_{i \in B} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)})$$

and so we are justified in working with just the per-sample gradient $\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}), y)$. Let us introduce some new recursive notation for the remainder of the explanation:

$$\begin{aligned}
\boldsymbol{z}^i = (z_1^i, \ldots, z_{d_i}) &:= \boldsymbol{L}_i(\boldsymbol{a}^{i-1}) = W^i \boldsymbol{a}^{i-1} + \boldsymbol{b}^i, && i = 1, \ldots, r, \\
\boldsymbol{a}^i = (a_1^i, \ldots, a_{d_i}) &:= \boldsymbol{g}_i(\boldsymbol{z}^i), && i = 1, \ldots, r, \\
\boldsymbol{a}^0 &:= \boldsymbol{x} \in \mathbb{R}^I, \\
\delta_j^i &:= \frac{\partial \ell}{\partial z_j^i}, \quad j = 1, \ldots, d_i, && i = 1, \ldots, r
\end{aligned}$$

thus $\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{a}^r$, $\ell(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}), y) = \ell(\boldsymbol{a}^r, y)$ and we have introduced the **adjoint** $\boldsymbol{\delta}^i = (\delta_1^i, \ldots, \delta_{d_i}^i) \in \mathbb{R}^{d_i}$.

We will now quickly recall the *chain rule*, as it is a major player in the process of backpropagation. Suppose we have differentiable functions $G \colon \mathbb{R}^d \to \mathbb{R}$ and $\boldsymbol{F} = (F_1, \ldots, F_d) \colon \mathbb{R}^{d'} \to \mathbb{R}^d$ and define $H = G \circ \boldsymbol{F} \colon \mathbb{R}^{d'} \to \mathbb{R}$, i.e. $H(\boldsymbol{x}) = G(\boldsymbol{y})$ with $\boldsymbol{y} = (y_1, \ldots, y_d) = \boldsymbol{F}(\boldsymbol{x})$. Then the chain rule states that

$$\frac{\partial H}{\partial x_i}(\boldsymbol{x}) = \sum_{j=1}^{d} \frac{\partial G}{\partial y_j}(\boldsymbol{y}) \frac{\partial F_j}{\partial x_i}(\boldsymbol{x}), \quad \boldsymbol{x} = (x_1, \ldots, x_{d'}).$$

Definition 4.1.1. (Hadamard product) For two matrices $A, B \in \mathbb{R}^{m \times n}$ the *Hadamard product* of the two matrices, $\odot \colon \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times n}$, is defined by

$$(A \odot B)_{ij} = (A)_{ij} (B)_{ij},$$

i.e. it is the element-wise product of the two matrices.

We are now ready to state the proposition that allows us to calculate the gradient of the cost function through the process of backpropagation.

Proposition 4.1.2. *We have*

$$\boldsymbol{\delta}^r = \boldsymbol{g}_r'(\boldsymbol{z}^r) \odot \nabla_{\hat{\boldsymbol{y}}} \ell(\boldsymbol{a}^r, \boldsymbol{y}), \tag{4.1.5}$$

$$\boldsymbol{\delta}^i = \boldsymbol{g}_i'(\boldsymbol{z}^i) \odot (W^{i+1})' \boldsymbol{\delta}^{i+1}, \qquad i = 1, ..., r-1, \tag{4.1.6}$$

$$\frac{\partial \ell}{\partial b_j^i} = \delta_j^i, \qquad i = 1, ..., r, \; j = 1, ..., d_i, \tag{4.1.7}$$

$$\frac{\partial \ell}{\partial W_{j,k}^i} = \delta_j^i a_k^{i-1}, \qquad i = 1, ..., r, \; j = 1, ..., d_i, \; k = 1, ..., d_{i-1}. \tag{4.1.8}$$

4.2 Recurrent Neural Networks

One of the difficulties in using deep neural networks to forecast time series is incorporating the idea of causality into the model. We want to know if data 10 days ago or possibly even a month ago is a contributing factor in predicting what will happen tomorrow. To combat this we introduce the idea of a *Recurrent Neural Network*. Recurrent neural networks are artificial neural networks that specialise in this idea of trying to preserve a cell's 'memory' to be used in future cells.

4.2.1 Basic Recurrent Neural Network

The main difference here is that an RNN cell receives both inputs $\boldsymbol{x}_{(t)}$ and $\boldsymbol{h}_{(t-1)}$ where $\boldsymbol{h}_{(t-1)}$ is the hidden state of the cell at the previous time step $t-1$. From this we receive an output for each time step and for basic RNN architectures, the output is often equal to the hidden state. We can *unroll the network through time* along a time axis, as in Figure 4.4, to make this more coherent.

The role of the hidden state $\boldsymbol{h}_{(t)}$ is to store information about the sequence up to the time step t, incorporating this idea of preserving the networks 'memory'. This is seen in the recursive component in the definition of the hidden state. If we denote the connection weights of the input variables to the hidden state by $\boldsymbol{w}_{xh}$ and the weights between hidden states $\boldsymbol{w}_{hh}$, then the hidden state at time t can be defined by

$$\boldsymbol{h}_t = \phi\big(\boldsymbol{w}_{xh}^T \boldsymbol{x}_t + \boldsymbol{w}_{hh}^T \boldsymbol{h}_{t-1} + b_h\big) \tag{4.2.1}$$

with activation function ϕ and hidden layer bias b_h. Computing the output of the output layer is similar to the MLP output, given by

$$\boldsymbol{y}_t = \boldsymbol{w}_{hy}^T \boldsymbol{h}_t + b_y \tag{4.2.2}$$

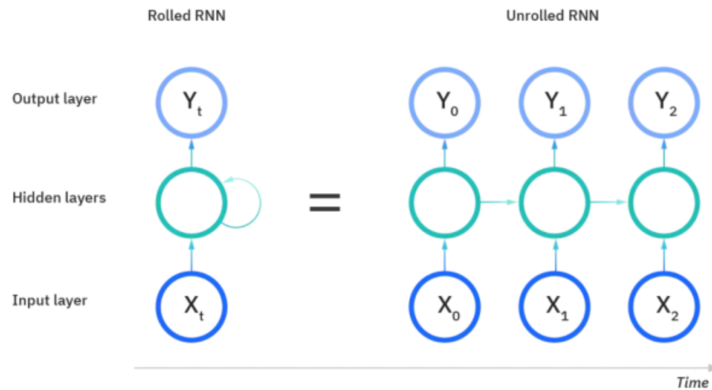

Figure 4.4: A 'rolled' visual of an RNN (left) and an 'unrolled' visual of an RNN showing the individual time steps within the network (right). Source [Education, 2020]

with $\boldsymbol{w}_{hy}$ the weight vector for connections between the hidden states and the output layer and b_y the output bias weight.

We will be working with *sequence-to-vector networks* (or *many-to-one networks*) where the network is fed in a sequence of inputs and all outputs, excluding the very last one, are ignored, as seen in Figure 4.5.

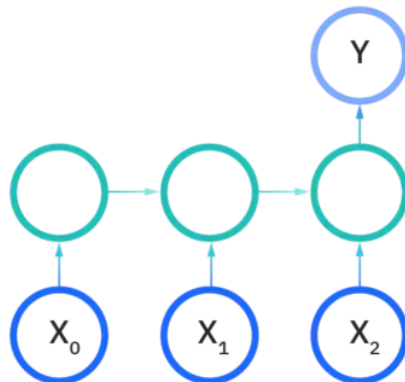

Figure 4.5: A many-to-one RNN visualised. Source [Education, 2020]

Backpropagation Through Time

As we walk through the process of Backpropagation Through Time we will assume we are working with a RNN model with 1 hidden layer. This is only for simplicity of the explanation and extending the idea to a model with more than 1 hidden layer is kindred to extending a feed-forward neural network from 1 hidden layer to more than one.

As we saw in Section 4.1.4, in order for the backpropagation algorithm to work we must be able to find derivatives of the cost function, which is exactly what we will be finding in this section. We now have more weights and parameters in the model and so we must also fine-tune these, however we will find that the recursive nature of the hidden weight connections makes this slightly more involved than the regular backpropagation algorithm. For more information on training recurrent neural networks We refer the reader to Chen's paper [Chen, 2018], for which the remainder of this explanation is accredited to.

Suppose at time t we have inputs $\boldsymbol{x}_t$, hidden states $\boldsymbol{h}_t$, output $\hat{y}_t$, model parameters $\boldsymbol{\theta} = (W_{xh}, W_{hh}, W_{h\hat{y}}, \boldsymbol{b}_{\boldsymbol{h}}, \boldsymbol{b}_{\hat{y}})$ and activation functions $\phi(x) = \tanh(x)$ in the hidden layer and $\phi(\boldsymbol{x}) = \text{Softmax}(\boldsymbol{x}) =: \sigma(\boldsymbol{x})$ in the output layer. The activation functions are defined by

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \sigma(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}.$$

Therefore we have

$$\begin{aligned} \boldsymbol{h}_t &= \tanh(W_{hh}\boldsymbol{h}_{t-1} + W_{xh}\boldsymbol{x}_t + \boldsymbol{b}_{\boldsymbol{h}}) \\ \hat{y}_t &= \sigma(W_{h\hat{y}}\boldsymbol{h}_t + \boldsymbol{b}_{\hat{y}}) \end{aligned}$$

We assume the parameters are the same across each time step to allow for a well generalised model. Suppose we use maximum likelihood to estimate the parameters, i.e. we aim to minimise the negative log likelihood so that our cost function is

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}) = -\sum_t y_t \log(\hat{y}_t).$$

For the remainder of this section we will denote the cost function by $\mathcal{L}$ and let $\mathcal{L}_t$ be the output of the model at time t so that $\mathcal{L}_t = -y_t \log(\hat{y}_t)$. Let $\boldsymbol{z}_t := W_{h\hat{y}}\boldsymbol{h}_t + \boldsymbol{b}_{\hat{y}}$ so that $\hat{y}_t = \sigma(\boldsymbol{z}_t)$. Then we get

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_t} = -(y_t - \hat{y}_t)$$

Let us find the partial derivative of the cost function with respect to W_{hh} by first considering the partial derivative of the output of the model at time $t+1$ with respect to W_{hh} by utilising the chain rule and then backpropagating through time. The third equality is justified by the weight matrix W_{hh} being used for all time steps within the hidden layer. From this we get,

$$\begin{aligned}\frac{\partial \mathcal{L}_{t+1}}{\partial W_{hh}} &= \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial W_{hh}} \\ &= \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_{t}} \frac{\partial \boldsymbol{h}_{t}}{\partial W_{hh}} \\ &= \sum_{k=1}^{t} \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_{k}} \frac{\partial \boldsymbol{h}_{k}}{\partial W_{hh}}\end{aligned}$$

If we then sum this derivative over all time steps we get the partial derivative

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t} \sum_{k=1}^{t+1} \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_{k}} \frac{\partial \boldsymbol{h}_{k}}{\partial W_{hh}}$$

Next we look to find the partial derivative of $\mathcal{L}$ with respect to W_{xh}. Just as we did before, we look first at the partial derivative of the output of the model at time step $t+1$ by once again utilising the chain rule and backpropagating through time. This yields

$$\begin{aligned}\frac{\partial \mathcal{L}_{t+1}}{\partial W_{xh}} &= \frac{\partial \mathcal{L}_{t+1}}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial W_{xh}} \\ &= \sum_{k=1}^{t+1} \frac{\partial \mathcal{L}_{t+1}}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_{k}} \frac{\partial \boldsymbol{h}_{k}}{\partial W_{xh}}.\end{aligned}$$

Summing over all previous time steps then gives

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{t} \sum_{k=1}^{t+1} \frac{\partial \mathcal{L}_{t+1}}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_{k}} \frac{\partial \boldsymbol{h}_{k}}{\partial W_{xh}}.$$

Simple application of the chain rule and summation over all time steps gives us the remaining partial derivatives

$$\frac{\partial \mathcal{L}}{\partial W_{h\hat{y}}} = \sum_{t} \frac{\partial \mathcal{L}}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial W_{h\hat{y}}}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_{\hat{y}}} = \sum_{t} \frac{\partial \mathcal{L}}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial \boldsymbol{b}_{\hat{y}}}$$

Vanishing/Exploding Gradient Problem

Recurrent neural networks, especially, are faced with the problem of vanishing and exploding gradients in which unstable gradients in deep recurrent neural network models are raised to significant powers, either growing exponentially or sending them to zero. We have seen that the backpropagation of recurrent neural networks involves composition of the same weight matrix multiple times, since it is that same weight matrix that is used across the whole of the hidden layer, applied for every time step in the model. For very deep feed-forward neural networks we potentially face the same problem, however here we can adjust individual weights per hidden layer so as to stabilise the overall model. Unfortunately we don't have that luxury in the case of a recurrent model and so handling instabilities becomes a lot more involved.

Let us illustrate this in a simple model taken from [Goodfellow et al., 2016]. Suppose we have a very simple recurrent neural network without inputs or a non-linear activation function, given by the recurrence relation

$$\boldsymbol{h}_t = \boldsymbol{W}^T \boldsymbol{h}_{t-1}. \tag{4.2.3}$$

If we work back recursively from (4.2.3) we obtain

$$\boldsymbol{h}_t = (\boldsymbol{W}^t)^T \boldsymbol{h}_0$$

and thus, if $\boldsymbol{W}$ is orthogonally diagonalisable, i.e. $\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$ for orthogonal $\boldsymbol{Q}$, we get the decomposition

$$\boldsymbol{h}_t = \boldsymbol{Q}^T \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}_0.$$

As a result of this, the eigenvalues of $\boldsymbol{W}$, and subsequently $\boldsymbol{\Lambda}$, are raised to the power of t. Eigenvalues less than 1 tend to zero, hence the term 'vanishing', and eigenvalues greater than 1 explode and grow exponentially. Consequently, long-term dependencies are hard to capture due to the instability of the eigenvalues.

There are, however, variations of recurrent neural networks that have been proposed that allow the model to handle this vanishing/exploding gradient problem much better than basic recurrent models, namely the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). We will be exploring these in the upcoming section.

4.2.2 Long Short-Term Memory

The first variant we will look at is the LSTM model, from the work of Hochreiter and Schmidhuber [Hochreiter and Schmidhuber, 1997]. The aim of the LSTM cell is to

preserve as much of the long-term memory as necessary until it is needed, working to alleviate the vanishing/exploding gradient problem. In the following section we will describe the structure of an LSTM cell and then go on to analyse the benefits of such a variation.

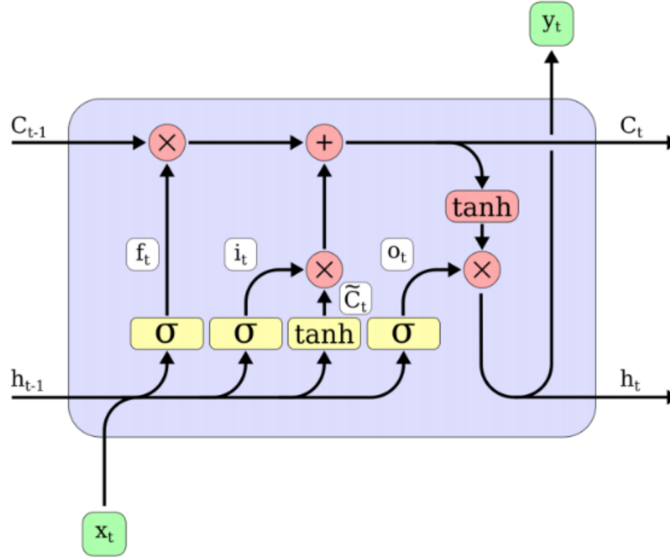

Figure 4.6: An LSTM cell

Figure 4.6 shows a standard LSTM cell, now with *gates* that govern how much of the models state is either remembered or forgotten. Here, the LSTM cell has an additional state vector, on top of the hidden state, known as the *cell state* which is denoted by $\boldsymbol{C}_t$ at time t. The cell's state can be thought of as the long-term state while the hidden state can be thought of as the short-term state. The gates are depicted by the red $\times$ circles in the diagram, where an $\times$ denotes the Hadamard product and a $+$ is the standard vector addition. The gates are the *forget gate*, *input gate* and the *output gate*. We also have 4 neural network layers in an LSTM cell as opposed to a single layer commonly found in standard RNN cells. Three of these layers utilise the Sigmoid activation function, outputting values $\boldsymbol{f}_t, \boldsymbol{i}_t, \boldsymbol{o}_t \in [0, 1]$, and the remaining layer applies the hyperbolic tangent activation function to output $\widetilde{\boldsymbol{C}}_t \in [-1, 1]$. The Sigmoid function is an important partner to the cell gates; outputting values in the range $[0, 1]$ allows for the quantities $\boldsymbol{f}_t, \boldsymbol{i}_t$ and $\boldsymbol{o}_t$ to directly impact how much of the cell state and $\boldsymbol{g}_t$ is carried through at each gate. A value of one meaning remember everything and similarly a value of 0 meaning forget everything.

Since the cell's state is the component of the LSTM model that makes it unique to that of standard RNN models, let us explore the life cycle of the cell state as it is processed through the LSTM cell. As the cell state from the previous time step, $\widetilde{\boldsymbol{C}}_{t-1}$, enters the current cell, it reaches the forget gate where the Hadamard product $\widetilde{\boldsymbol{C}}_{t-1} \odot \boldsymbol{f}_t$ is calculated. Intuitively this amounts to the forget gate controlling how much of the long-term state should be erased according to $\boldsymbol{f}_t$. Following the forget gate, memories are added to the cell state at the input gate, where the vector $\widetilde{\boldsymbol{C}}_t \odot \boldsymbol{i}_t$ is added to the output of the forget gate to get the current cell state. Alongside this, the current cell state $\widetilde{\boldsymbol{C}}_t$ is copied and fed to a tanh function and then once again filtered by the output gate, taking the Hadamard product with $\boldsymbol{o}_t$ to finally produce the cell's hidden state $\boldsymbol{h}_t$ and the output $\boldsymbol{y}_t$. This lends itself to the idea of the cell state first having "memories" added to it at the input gate, controlled by $\boldsymbol{i}_t$, and then the output gate determines how much of the long-term state should be incorporated into $\boldsymbol{h}_t$ and $\boldsymbol{y}_t$, according to $\boldsymbol{o}_t$.

More formally, the relevant equations for the LSTM forward propagation are:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma\big(\boldsymbol{W}_{xf}^T \boldsymbol{h}_{t-1} + \boldsymbol{W}_{hf}^T \boldsymbol{x}_t + \boldsymbol{b}_f\big) \\
\widetilde{\boldsymbol{C}}_t &= \tanh\big(\boldsymbol{W}_{xC}^T \boldsymbol{h}_{t-1} + \boldsymbol{W}_{hC}^T \boldsymbol{x}_t + \boldsymbol{b}_C\big) \\
\boldsymbol{i}_t &= \sigma\big(\boldsymbol{W}_{xi}^T \boldsymbol{h}_{t-1} + \boldsymbol{W}_{hi}^T \boldsymbol{x}_t + \boldsymbol{b}_i\big) \\
\boldsymbol{o}_t &= \sigma\big(\boldsymbol{W}_{xo}^T \boldsymbol{h}_{t-1} + \boldsymbol{W}_{ho}^T \boldsymbol{x}_t + \boldsymbol{b}_o\big) \\
\boldsymbol{C}_t &= \big(\boldsymbol{f}_t \odot \boldsymbol{C}_{t-1}\big) + \big(\boldsymbol{i}_t \odot \widetilde{\boldsymbol{C}}_t\big) \\
\boldsymbol{h}_t = \boldsymbol{y}_t &= \boldsymbol{o}_t \odot \tanh\big(\boldsymbol{C}_t\big)
\end{aligned}$$

where $\boldsymbol{W}_{xf}, \boldsymbol{W}_{xC}, \boldsymbol{W}_{xi}, \boldsymbol{W}_{xo}$ are the weight matrices for the connections between the 4 layers and the input $\boldsymbol{x}_t$, the weight matrices for the connections between the 4 layers and the the hidden state of the cell's previous time step $\boldsymbol{h}_{t-1}$ are denoted by $\boldsymbol{W}_{hf}, \boldsymbol{W}_{hC}, \boldsymbol{W}_{hi}, \boldsymbol{W}_{ho}$ and finally $\boldsymbol{b}_f, \boldsymbol{b}_C, \boldsymbol{b}_i, \boldsymbol{b}_o$ are the bias terms for each of the 4 layers.

However in recent years there has also been the development of a new gated memory cell which is computationally more efficient while performing similarly as well as LSTM cells: the Gated Recurrent Unit (GRU).

4.2.3 Gated Recurrent Unit

The GRU cell is a variation of the LSTM cell, first introduced by Cho et al. [Cho et al., 2014], that is much simpler in design and computationally faster while performing on par with the performance of the LSTM cell. The variation here is a reduction in the

number of neural network layers from 4 to 3 and combine the 2 state vectors in the LSTM model to just the one hidden state vector $\boldsymbol{h}_t$. The GRU cell also only has 2 gates, an *update gate* and a *reset gate*, as opposed to the 3 gates the LSTM possessed. Figure 4.7 shows a block diagram of a GRU cell with input vector $\boldsymbol{x}_t$, hidden state vector $\boldsymbol{h}_t$ and output vector $\boldsymbol{y}_t$. The '1−' block represents the function $\psi(x) = 1 - x$, with the $\times$ and $+$ blocks having the same meaning as in the LSTM cell.

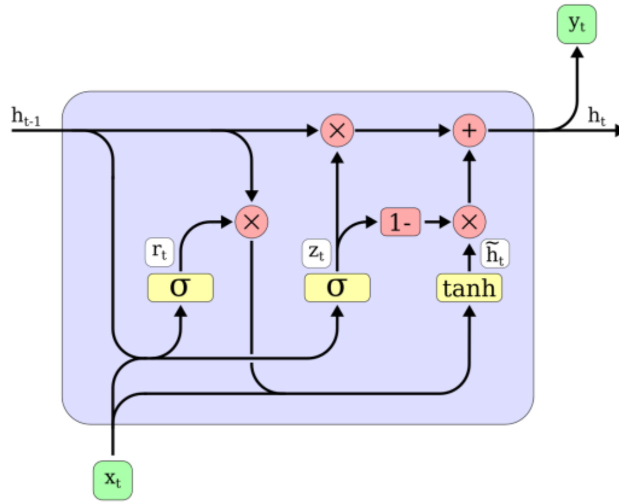

Figure 4.7: A GRU cell

In the GRU cell, the gate controller $\boldsymbol{z}_t$ controls how much of the hidden state is allowed to pass through the update gate as well as regulating, via $1 - \boldsymbol{z}_t$, the amount of $\widetilde{\boldsymbol{h}}_t$ that is added to the flow of the hidden state $\boldsymbol{z}_t \odot \boldsymbol{h}_{t-1}$. We also have a new gate controller, $\boldsymbol{r}_t$, that tunes how much of the previous state is fed to the main layer $\widetilde{\boldsymbol{h}}_t$.

More formally, the equations for the forward propagation of the GRU cell are:

$$
\begin{aligned}
\boldsymbol{r}_t &= \sigma\big(\boldsymbol{W}_{xr}^T \boldsymbol{x}_t + \boldsymbol{W}_{hr}^T \boldsymbol{h}_{t-1} + \boldsymbol{b}_r\big) \\
\boldsymbol{z}_t &= \sigma\big(\boldsymbol{W}_{xz}^T \boldsymbol{x}_t + \boldsymbol{W}_{hz}^T \boldsymbol{h}_{t-1} + \boldsymbol{b}_z\big) \\
\widetilde{\boldsymbol{h}}_t &= \tanh\big(\boldsymbol{W}_{x\tilde{h}}^T \boldsymbol{x}_t + \boldsymbol{W}_{h\tilde{h}}^T \big(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}\big) + \boldsymbol{b}_{\tilde{h}}\big) \\
\boldsymbol{h}_t = \boldsymbol{y}_t &= \boldsymbol{z}_t \odot \boldsymbol{h}_{t-1} + \big(1 - \boldsymbol{z}_t\big) \odot \widetilde{\boldsymbol{h}}_t
\end{aligned}
$$

where $\boldsymbol{W}_{xr}, \boldsymbol{W}_{xz}, \boldsymbol{W}_{x\tilde{h}}$ are the weight matrices for the connections between the 3 layers and the input $\boldsymbol{x}_t$, the weight matrices for the connections between the 3 layers and the hidden state of the cell's previous time step $\boldsymbol{h}_{t-1}$ are denoted by $\boldsymbol{W}_{hr}, \boldsymbol{W}_{hz}, \boldsymbol{W}_{h\tilde{h}}$ and finally $\boldsymbol{b}_r, \boldsymbol{b}_z, \boldsymbol{b}_{\tilde{h}}$ are the bias terms for each of the 3 layers.

Chapter 5

Implementation and Results

Here we will give a description of the implementation of the models considered in Chapters 3 & 4 before going on to detail the performance metrics and numerical results obtained from the evaluation of the models and interpret them in the context of the problem we are working on.

5.1 Implementation

All the processing of data and models implemented in this project were done so using Python, through the appropriate Scikit-learn and Tensorflow libraries. Due to the stochastic nature of the data, there was no significant imbalance in the target data, i.e. a roughly similar number of daily increases and decreases in the price of the futures contract, and thus no alternative methods were needed in order to account for this. As a result we were free to use a full range of performance metrics where, otherwise, an imbalance in the target data would've made some metrics, 'accuracy' for example, an unreliable measure.

A crucial element of building a model is making sure that the model is not overfit to the training data, hoping for a model that generalises well when applied in a practical setting. Although it is possible to build a model that performs exceptionally well on the data it has been trained on, often when presented with new incoming data the model will perform significantly worse. However if one makes the decision of what model to choose and refine based on the performance on this new data, it's possible that we begin to overfit on this new data as well. To overcome the danger of overfitting we utilise Scikit-learn's *train_test_split* class in order to split and shuffle the data in a 4:1 ratio, first splitting into a *training set* and a *test set* and then performing the same procedure on the training set to obtain a training set and a *validation set*. The benefit of going through this process is we now have a large set to train the models on, a validation set to evaluate the models on for further refining and a test set for our final evaluations and results.

5.1.1 K-fold Cross-validation

As previously mentioned, overfitting a model to the training data is not ideal for the future use of a model. We don't want a model that performs well on the training data only for it to be suboptimal when introduced to new data. One method we use in the training of the models in order to avoid the problem of overfitting is *K-fold Cross-validation*. In k-fold cross-validation we split the training data into k distinct subsets of equal, or as close to equal, size. The training is run through k times, each time leaving out one of the k subsets which will be used for validation after the training of the model. A performance metric is used to evaluate each of these models and then an arithmetic mean is calculated from these individual k scores to evaluate the model as a whole.

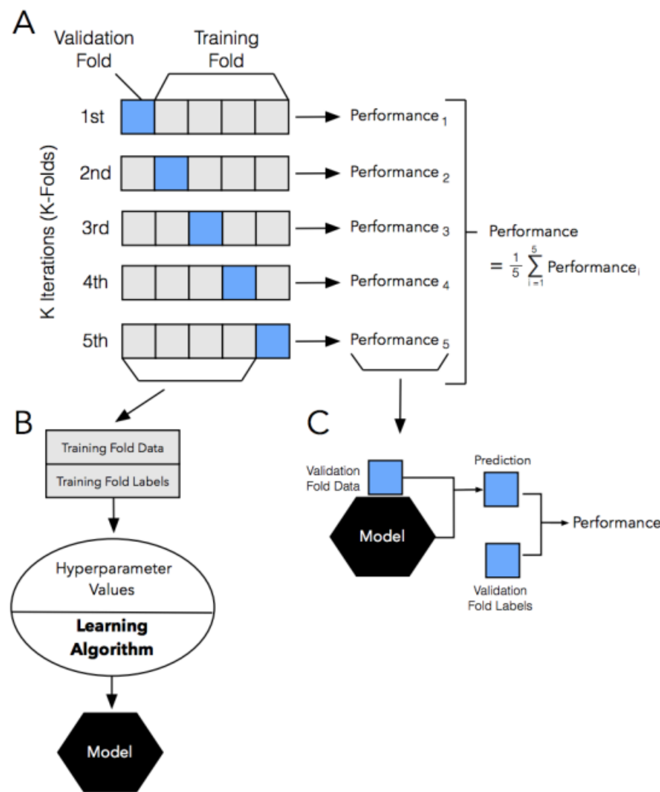

Figure 5.1: An illustration of k-fold cross validation with $k = 5$. Source [Raschka, 2020]

For the training of our models we will be using k-fold cross-validation with $k = 5$. Before any justification for the choice of $k = 5$, the author first notes that the

number of folds used is itself a hyperparameter that can affect the performance of the model and thus a way of improving our models in future pieces of work would be to experiment with different values of k. The choice of $k = 5$ is made for computational ease. Any larger values of k would take the models a lot longer to train whereas any values of k less than 5 would give us less data to train on and thus the *pessimistic bias* of the performance metric would increase as well as an increase in the *variance* due to an oversensitivity to how the data is split.

5.1.2 Hyperparameter Tuning

Within the models there are tunable parameters, known as *hyperparameters*, that effect the performance of the models. These parameters are fixed at the beginning of training and thus cannot be learned through the training of the models. To find the best performing model we must tune these hyperparameters to their optimum values. A standard method to find the optimum hyperparameters is utilising the GridSearchCV class[1] provided by the scikit-learn library. This entails systematically working through a defined parameter space for each hyperparameter specified. A cross-validation is performed for each permutation of the hyperparameter space and the best performing model is chosen based on which model performed best according to some metric. For our models this metric will be the accuracy of the model.

5.1.3 Choice of Features

In the building of these models we will be changing the features that are used in order to capture different types of behaviour. For the support vector machines and artificial neural networks this will mean including certain parts of the data, transforming it and even lagging the data in order to capture long term dependencies.

Raw Data

We begin at the very simplest implementation in which daily data for each of the different feature types discussed in Chapter 3 are used. For this kind of implementation it is expected that features whose absolute value on the day is important, such as weather forecasts, foreign exchange and stock indices, will have more influence over these models when compared to the remaining features whose daily changes are more telling of future events.

Lagged Data

For the majority of our models we will be lagging the initial features as mentioned in Chapter 3, that is to say we will be including a varying amount of historical data

[1] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

for each feature as it's own feature. We do this in order to capture any long or short term dependencies, depending on the amount of historical data we choose to include. This is a standard method used in forecasting time series in machine learning and is adopted easily using the Pandas library.

Differences

A transformation we will be applying to the data is looking at the difference between the data of the current day and the data from the previous day, i.e. 'differenced' data. The idea in doing so is to see if both the sign and the magnitude of the change in the last day, and previous days if we are combining differenced data with lagging the data, make a significant impact on the direction of the price in the future. For example, a significant drop in temperature may increase demand for certain crude oil products which in turn would drive the underlying price of the oil up, finally impacting the price of the futures contract.

5.2 Evaluation

5.2.1 Performance Metrics

Confusion Matrix

A confusion matrix is an intuitive way of visualising the performance of a supervised learning algorithm. For our binary classification problem the confusion matrix is a 2×2 matrix whose rows are the 'actual' classes of each instance and the columns are the 'predicted' classes of the instances having been run through the trained model.

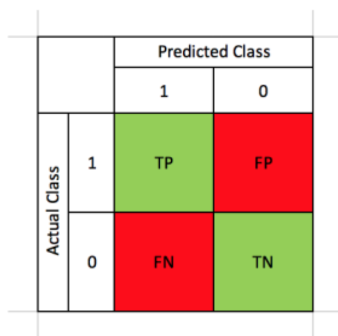

Figure 5.2: A confusion matrix

If we consider our two classes to be a 'positive' class labelled by 1 and a 'negative' class labelled by 0, then the elements in the confusion matrix can be described as follows:

- True Positive (TP) - Corrected predicted instances from the positive class,
- True Negative (TN) - Correctly predicted instances from the negative class,
- False Positive (FP) - Incorrectly predicted instances from the positive class (equivalent to Type I error),
- False Negative (FN) - Incorrectly predicted instances from the negative class (equivalent to Type II error).

The confusion matrix allows us to see exactly where our predictive models are classing instances and whether there may be bias towards one class over the other. While *accuracy*, which will be discussed in more detail next, is usually the immediate performance metric to be considered, for general problems the performance of a model purely based on accuracy can lead to problems when there is *imbalanced data*, i.e. a significantly higher proportion of one class of data over the other. Let us consider a well-known example of where accuracy can be misleading. This isn't to discourage the use of accuracy as a performance metric but instead more of an example of where carefully examining the confusion matrix can be beneficial in understanding how a model is behaving.

> Consider the binary classification problem in which we have an imbalanced dataset of 100 samples made up of 95 positive instances and 5 negative instances. Suppose we have a model that classifies every instance as positive. Then we would yield a performance of 95% accuracy and an F1 score of 97%. Both of these scores are very high for a classification problem, however if we were to consider the performance metric that calculates the proportion of correctly classified negative instances, *selectivity*, then we would yield a result of 0%, which is significantly worse than all the impressive results we had previously achieved.

Fortunately, for the data we are using, there is roughly the same proportion of both target classes and so accuracy is still a viable metric to consider.

Accuracy

Accuracy is one of the most natural performance metrics to consider and can be formally defined in the now familiar terminology by the following equation,

$$Accuracy = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

i.e. it is the total proportion of instances correctly classified. For a balanced dataset it gives one of the best indications of how well a model performs and will be one of the leading metrics we will be basing the performance of our models on.

Precision & Recall

Precision can be thought of as the proportion of the data that is correctly predicted in the positive class. The formal definition is given by,

$$Precision = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Essentially, it is the ability of the predictive model to accurately predict the positive instances that are classified as positive. In the context of the model we are trying to build, a high precision would mean the model has a high success rate in predicting an increase in the price of the futures contract.

A metric that often goes hand in hand with precision is the *recall* of a model. Formally we define recall to be the following equation,

$$Recall = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

In essence, it is the ability of the model to accurately classify all actually positive instances as positive. For the problem we face, a high recall would mean the model performs well as identifying all the positive price changes as positive.

Achieving either a high precision or a high recall on their own is relatively easy with some very trivial models; for example if every observation was classified to be the positive class then the model would achieve a 100% recall. However clearly this model would never correctly identify any of the negative instances and thus recall, or precision, on it's own is not enough to judge the performance of a classifier. Instead they are often considered alongside each other in a precision-recall curve or combined in the F_β score, which will be looked at further in the following section.

F_1 Score

One of the measures that we will be considering is the F_β score, in particular the F_1 score. Taken from the scikit-learn metrics documentation[2], the F_β score is 'the weighted harmonic mean of precision and recall, reaching its optimal value at 1 and

[2]https://scikit-learn.org/stable/modules/generated/sklearn.metrics.fbeta_score.html

its worst value at 0'. The formal definition is given below,

$$F_\beta \ Score \quad = \quad (1+\beta^2) \times \frac{Precision \times Recall}{(\beta^2 \times Precision) + Recall} \tag{5.2.1}$$

Different weightings of either precision or recall may be necessary depending on what kind of problem is being solved. As an example, if one was training a model to identify a cancer in the body, it is far more beneficial for a classification model to be overcautious and thus have a much higher weight on recall so that all cancers are identified. In the F_β score, β is defined to be the weight attributed to precision, i.e. it is the scale by which we consider recall β times more important than precision.

Given our balanced dataset, and the fact that neither precision nor recall is more important than the other, we will be using the F_1 score which weights the two metrics equally. That is, we will set $\beta = 1$ in (5.2.1) so that our measure is calculated by the following equation,

$$F_1 \ Score \quad = \quad 2 \times \frac{Precision \times Recall}{Precision + Recall}.$$

There are criticisms of the F_1 measure in using it for unbalanced data which is justified given it's focus purely on the positive class. However, given the dataset we are using, the F_1 score is fairly justified and the choice of $\beta = 1$ allows for a perfectly weighted harmonic mean accounting for both precision and recall equally.

5.2.2 Support Vector Machines

We now present the results from the best performing SVM models for each of the different feature sets. After an extensive search, there was no well-respected global benchmark for the time series forecasting performance metrics so in order for us to make some comparisons we have introduced a "naive model" in which we predict the price of the futures contract to increase one day, decrease the next, increase the day after that and so on, i.e. our prediction vector is $[1, 0, 1, 0, \ldots, 1, 0]$. Comparing our models to this, and each other, allows for us to quantify how well our actively tuned models are performing. Table 5.1 shows how well this naive model performs while Table 5.3 gives the performance scores from the SVM models, with the best scores for each metric highlighted in bold. To visualise the results further, Figure **??** graphs all of the performance metrics for each of the models.

	Accuracy	F_1 Score	Precision	Recall
Naive Model	0.458491	0.451243	0.445283	0.457364

Table 5.1: Results of the naive model

Model	Accuracy	F_1 Score	Precision	Recall
Raw data - Unlagged	0.515094	0.566610	0.501493	0.651163
Raw data - 1 Day Lag	**0.620755**	**0.644248**	0.610738	**0.681648**
Raw data - 2 Day Lag	0.578450	0.598198	**0.628788**	0.570447
Raw data - 3 Day Lag	0.604915	0.604915	0.573477	0.640000
Raw data - 5 Day Lag	0.565217	0.554264	0.567460	0.541667
Raw data - 10 Day Lag	0.571970	0.590580	0.548822	0.639216
Differenced data - 1 Day Lag	0.568998	0.589928	0.618868	0.563574
Differenced data - 2 Day Lag	0.578450	0.586271	0.546713	0.632000
Differenced data - 3 Day Lag	0.593573	0.616756	0.570957	0.670543
Differenced data - 5 Day Lag	0.589792	0.589792	0.573529	0.607004
Differenced data - 10 Day Lag	0.541667	0.566308	0.548611	0.585185

Table 5.2: Table of results for the 11 different SVM models

The first result we can conclude is that all of the SVM models perform better than our benchmark model which justifies the use of the support vector machine as a predictive model. Not only that, all models also have an accuracy of over 50% which means our models are better than making a decision based on the flip of a coin. Clearly the 'Raw data - 1 Day Lag' model performs best across all but one metric with an accuracy of 62.1%, F_1 score of 64.4% and a recall of 68.2%. Even the precision of this model ranks as the second highest with 61.1% only bested by 'Raw data - 2 Day Lag' with a precision of 62.9%. Thus we can conclude that, of the SVM models, the model we would use for the forecasting of price changes is the 'Raw data - 1 Day Lag' model. From now on 'n Day Lag' with be abbreviated to nDL for brevity.

A closer look at 5.3 reveals that in most cases, the top scoring models belong to the 'Raw data' class of SVM models. The top two accuracy scores (1DL and 3DL), two of the top three F_1 scores (1DL and 3DL), two of the top three precision scores (2DL and 3DL) and three of the top five recall scores (1DL, 3DL and 10DL). With this in mind, we conclude that, for the SVM models, the more socioeconomic features in the feature set play a much bigger role in correctly predicting our classes. That is to say that factors such as foreign exchange, stock indices and weather forecasts are expected to play a more significant role in the calculation of the hyperplane than possibly other features. We can explore this hypothesis by plotting the square of the *coef_* attribute, as in [Guyon et al., 2002], from the LinearSVC class we have implemented for the 'Raw data - 1 Day Lag' model.

The square of the hyperplane's coefficients indicates how important each coefficient is compared to the others. Clearly a coefficient of 0 shows that the direction of that component played no part in calculating the separating hyperplane. As hypothesised, the two major contributors to the SVM hyperplane are the present day

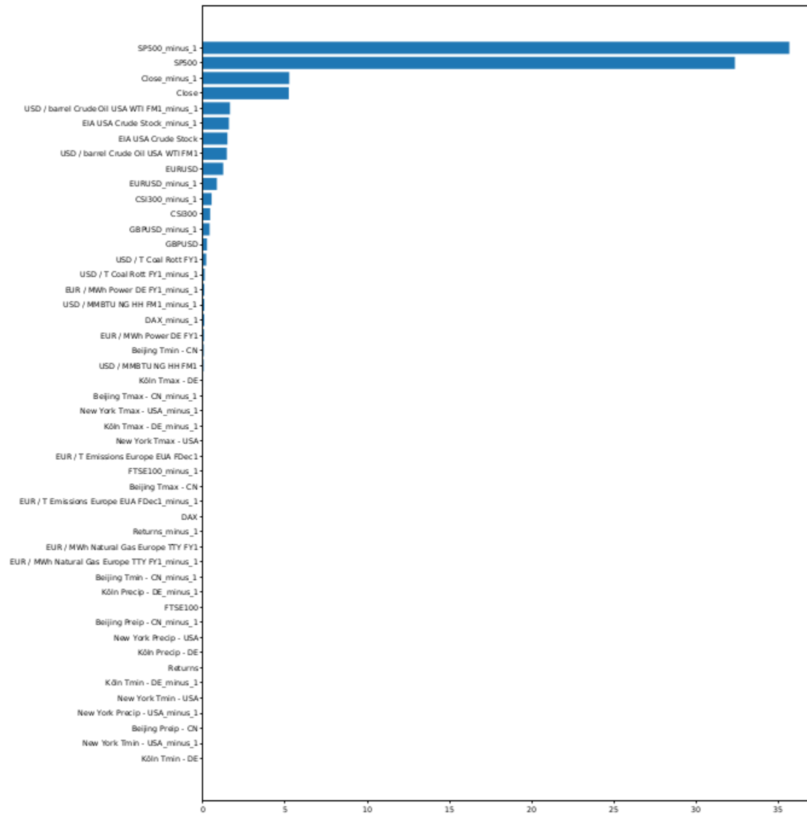

Figure 5.3: 'Raw data - 1 Day Lag' LinearSVC coef_ square weightings

and previous day values of the S&P 500 stock index; an index tracking the top 500 companies, by market capitalisation, listed in the US.

Model	Kernel	Hyperparameters
Raw data - Unlagged	Linear	$C = 0.04$
Raw data - 1 Day Lag	Linear	$C = 19.0$
Raw data - 2 Day Lag	Linear	$C = 1.0$
Raw data - 3 Day Lag	Linear	$C = 1000$
Raw data - 5 Day Lag	Linear	$C = 8.0$
Raw data - 10 Day Lag	Linear	$C = 100.0$
Differenced data - 1 Day Lag	Sigmoid	$C = 1.0$ $\gamma = 0.026$
Differenced data - 2 Day Lag	Linear	$C = 0.02$
Differenced data - 3 Day Lag	Linear	$C = 1.4$
Differenced data - 5 Day Lag	Linear	$C = 0.02$
Differenced data - 10 Day Lag	Linear	$C = 0.01$

Table 5.3: Table of hyperparameters for the 11 different SVM models

5.2.3 Deep Neural Networks

Following the SVM results, we proceed onto the results obtained from employing deep feed-forward neural networks in our classification problem. First we note that no model outperformed the best SVM model 'Raw data - 1 Day Lag' in the 'accuracy' metric. Since this is the metric we will be prioritising in choosing a superior model, we can conclusively state that no feed-forward neural network we explored is better than the SVM model's already looked at. While the feed-forward neural networks do offer some high scores in terms of F_1 Score and Recall, we will see that this is as a result of the model's inability to train or find a pattern. In most cases, these models should then be ignored. The results can be found in Table 5.4.

Let us first dissect the 'Raw Oil Data' models. In general the 'accuracy' scores for these models fluctuate around 50%, which is only just slighter better/worse than the flip of a fair coin. The lowest 'accuracy' score among the lagged SVM models was 0.541667, a score greater than all of the 'Raw Oil Data' scores and so we can conclude that this category is not worth exploring further. Moreover, we can see that this category boasts some incredibly high 'recall' values, with one of the models even achieving a recall of 100%. Upon further inspection of the models' confusion matrices, this is as a result of the models inability to train on the data provided and thus dumping all observations in the positive class. Because the target variables are marginally unbalanced, in particular there are slightly more positive class instances than negative class instances, classifying every single observation as the positive class allows the model to achieve an accuracy of just over 50% in most cases and so the model sees this as a win. The F_1 Score is a weighted average of the precision and recall and thus very high recall scores, despite note necessarily being advantageous, scale up the F_1 Score, making them appear better than they actually are in spite of fairly average precision scores.

The 'Differenced Oil Data' doesn't offer any particularly attractive results. Similarly to the 'Raw Oil Data' category we have accuracy scores fluctuating around 50% which, coupled with very high recall scores and inspection of the confusion matrices, can be attributed to the models once again not training successfully on the data and classifying every training instance as the positive class. As seen before, we have some quite large F_1 Scores due to extremities in the recall of the models and so they are nothing significant. Without large recall values, as seen in the models with more than 15 days of lagged data, we can observe very insignificant values across all performance metrics. From the previous two paragraphs it is enough to conclude that in order to build effective classification models we will need to use more than just historical Brent Crude Oil data.

The introduction of our complete feature set immediately shows improvements. To begin with, the 'Full Raw Feature Set - 15 Day Lag' achieves the second highest accuracy score across all feed-forward neural network models with 57.1%. This is clearly a model that has trained to some degree and can successfully predict binary movements more often than not without unrealistic recall and precision sores, i.e. this score is not down a bulk classification of all positive or negative instances. If we exclude all FNN models that have been unsuccessful in training, made evident by a very large recall or precision score, them the 'Full Raw Feature Set - 15 Day Lag' model also has the highest F_1 score across all accepted models. This is clearly a move in the right direction in terms of improving our models and we will see that the final category performs arguably the best out of all the categories.

Here we highlight the performance of the 'Full Differenced Feature Set' category of feed-forward neural networks. The first point to note is the 'Full Differenced Feature Set - 2 Day Lag' model achieving the highest accuracy score of 58.4%, which also places it as the fifth most accurate model amongst the SVM models. This category of models also hosts 4 of the top 5 best performing feed-forward neural networks in terms of accuracy which justifies it's use as a category. All of these top 5 results have an accuracy of over 54% which is typically how the target variables are split and thus clearly these models have trained successfully and recognised various signals in the data that give them a competitive edge. This category is also home to the model with the highest precision ('Full Differenced Feature Set - 1 Day Lag' with 61.9%) which, given the results we've seen so far, is a significant result. Typically we've seen fairly average precision scores amongst our results due to the models classifying all results in one lass for models that are unable to train effectively. A higher precision suggests a lower number of False Positives which is promising; less False Positives means more classifications in the negative class which is a sign that our model is training successfully. Overall we can conclude that the 'Full Differenced Feature Set' was the most successful set of features with the best performing model amongst the feed-forward neural networks being the model associated with the 'Full Differenced Feature Set - 2 Day Lag' feature set.

While disappointing, it doesn't come as a surprise that the majority of our SVM models perform better than the feed-forward neural networks. Research by Ince and Trafalis [Ince and Trafalis, 2008] produced similar results whereby SVM's were shown to perform better in short-term stock price prediction over Multi-layer Perceptrons.

Model	Accuracy	F_1 Score	Precision	Recall
Raw Oil Data - 1 Day Lag	0.524218	0.676127	0.524611	0.950704
Raw Oil Data - 2 Day Lag	0.513182	0.649603	0.520878	0.862837
Raw Oil Data - 3 Day Lag	0.498467	0.664754	0.497851	**1.000000**
Raw Oil Data - 5 Day Lag	0.492638	0.425295	0.522184	0.358734
Raw Oil Data - 10 Day Lag	0.488029	0.655941	0.488323	0.998744
Raw Oil Data - 15 Day Lag	0.514742	0.617248	0.517045	0.765625
Raw Oil Data - 20 Day Lag	0.502766	0.644396	0.506916	0.884198
Raw Oil Data - 30 Day Lag	0.510154	0.640145	0.523669	0.823256
Raw Oil Data - 40 Day Lag	0.533580	**0.686802**	0.533419	0.963995
Differenced Oil Data - 1 Day Lag	0.521153	0.682907	0.522360	0.985932
Differenced Oil Data - 2 Day Lag	0.494788	0.659222	0.495955	0.982737
Differenced Oil Data - 3 Day Lag	0.522992	0.656664	0.522105	0.884661
Differenced Oil Data - 5 Day Lag	0.500613	0.583419	0.505768	0.689238
Differenced Oil Data - 10 Day Lag	0.531614	0.621715	0.534527	0.742891
Differenced Oil Data - 15 Day Lag	0.512899	0.557724	0.535906	0.581395
Differenced Oil Data - 20 Day Lag	0.507683	0.534031	0.519231	0.549701
Differenced Oil Data - 30 Day Lag	0.485538	0.501193	0.505415	0.497041
Differenced Oil Data - 40 Day Lag	0.512015	0.530806	0.527059	0.534606
Full Raw Feature Set - 1 Day Lag	0.523629	0.524528	0.538760	0.511029
Full Raw Feature Set - 2 Day Lag	0.540643	0.522593	0.607306	0.458621
Full Raw Feature Set - 3 Day Lag	0.487713	0.549085	0.492537	0.620301
Full Raw Feature Set - 5 Day Lag	0.514178	0.541889	0.515254	0.571429
Full Raw Feature Set - 10 Day Lag	0.503788	0.540351	0.496774	0.592308
Full Raw Feature Set - 15 Day Lag	0.571157	0.594982	0.549669	0.648438
Full Raw Feature Set - 20 Day Lag	0.515209	0.495050	0.510204	0.480769
Full Raw Feature Set - 30 Day Lag	0.505725	0.483034	0.542601	0.435252
Full Raw Feature Set - 40 Day Lag	0.500000	0.561345	0.470423	0.695833
Full Differenced Feature Set - 1 Day Lag	0.565217	0.575646	**0.619048**	0.537931
Full Differenced Feature Set - 2 Day Lag	**0.584121**	0.584906	0.587121	0.582707
Full Differenced Feature Set - 3 Day Lag	0.555766	0.575045	0.578182	0.571942
Full Differenced Feature Set - 5 Day Lag	0.556818	0.561798	0.576923	0.547445
Full Differenced Feature Set - 10 Day Lag	0.518027	0.511538	0.542857	0.4836364
Full Differenced Feature Set - 15 Day Lag	0.56464	0.568738	0.583012	0.555147
Full Differenced Feature Set - 20 Day Lag	0.500952	0.513011	0.530769	0.4964029
Full Differenced Feature Set - 30 Day Lag	0.500956	0.479042	0.493827	0.465116
Full Differenced Feature Set - 40 Day Lag	0.533589	0.542373	0.558140	0.527473

Table 5.4: Table of results for the different feed-forward neural network models

5.2.4 Recurrent Neural Networks

Finally we examine the results of our recurrent neural network models and their variations, i.e. LSTM and GRU models. Due to time constraints, only the full feature set was explored as opposed to isolating the oil data aswell, given the performance of both feed-forward neural network and simple recurrent neural network models were best for these feature sets.

Immediately we can conclude that the basic recurrent neural networks, across all choices of feature sets, are the worst performing of all the models we have implemented. The best performing model across this choice of architecture is 'Full Differenced Feature Set - 40 Day Lag' with an accuracy of 53.3%. Compared to the SVM models previously mentioned, the best performing model for the recurrent neural network underperforms compared to every SVM model except the 'Raw data - Unlagged' model. This is disappointing as one would hope that by the very nature of recurrent neural networks, they would be able to hold some kind of memory throughout the model and perform better than they have done here. A onsiderable number of models are even trained to achieve accuracy scores of less than 50%, performing worse than basing ones decision on the flip of a coin.

As we saw with the feed-forward models, for the oil data alone there are some extremely high recall values, with 2 models achieving 100% recall. But once again, after a closer inspection of the confusion matrices, this is entirely down to the models being unable to train and thus classifying a large majority (if not all instances) as the positive class. This in turn will produce some large F_1 values, as can be seen in both Oil Feature set categories where a large F_1 score is not accompanied by a high level of accuracy. These suggest poorly fit models, if fit at all.

After implementing and evaluating the simple recurrent neural network architectures we try to capture any long term dependencies and preserve both long and short term memories within the data, thus choosing to implement recurrent neural network architectures with LSTM and GRU cells. We begin by giving the results for the LSTM cell networks in Table 5.6 before then giving the GRU results in Table 5.7.

The LSTM models are more promising than the simple recurrent neural network models, evident by our ability to immediately identify more stable recall values. This suggest the models are making 'conscious' decisions about the observations rather than dumping all instances in one class or the other. Our most accurate LSTM model, 'Full Differenced Feature Set - 40 Day Lag', is also more accurate than all simple recurrent neural network models and greater than 54% which once again suggests some 'conscious' decisions by the model rather than a possible split in the day. Interestingly we may note that the most accurate model is in the Full Differenced Feature Set, as is the best performing model from all our artificial neural network architectures. This suggests that for models that learn in the way these artificial neural networks do, i.e. backpropagation, the most information is extracted from the changes in the previous days data across the whole dataset which is in contrast to the best performing model of the SVM models. The best performing

model for the SVM models was trained on a raw dataset rather than the changes. Our guess it that, unlike the SVM models, the artificial neural network models learn the most from features where their daily changes are more telling of future events rather than their raw absolute value.

Introducing GRU cells into our recurrent neural networks gives us our best performing model across all recurrent neural network models in 'Full Differenced Feature Set - 20 Day Lag' with an accuracy of 58.7%. This model also has the highest precision amongst all the GRU models, boasting a precision of 61.7%. A large precision is now significant given the identification of some of the models' tendencies to dump all observations in the positive class if unable to train successfully. The highest F_1 score and recall value go hand-in-hand, unsurprisingly, with the 'Full Differenced Feature Set - 30 Day Lag' model. From this we can possible claim that GRU cells train on less data better than the LSTM cells and are thus more suited to this problem than the other recurrent unit cells.

Model	Accuracy	F_1 Score	Precision	Recall
Raw Oil Data - 1 Day Lag	0.505212	0.590563	0.520107	0.683099
Raw Oil Data - 2 Day Lag	0.511956	0.662426	0.518936	0.915592
Raw Oil Data - 3 Day Lag	0.496628	0.660883	0.496894	0.986436
Raw Oil Data - 5 Day Lag	0.521472	0.677952	0.523263	0.962485
Raw Oil Data - 10 Day Lag	0.488643	0.656495	0.488643	**1.000000**
Raw Oil Data - 15 Day Lag	0.507985	0.642251	0.511016	0.864183
Raw Oil Data - 20 Day Lag	0.507068	0.668869	0.508475	0.977081
Raw Oil Data - 30 Day Lag	0.488615	0.403446	0.527205	0.326744
Raw Oil Data - 40 Day Lag	0.516944	0.636364	0.529730	0.796748
Differenced Oil Data - 1 Day Lag	0.522379	**0.685507**	0.522783	0.995311
Differenced Oil Data - 2 Day Lag	0.497241	0.664210	0.497241	**1.000000**
Differenced Oil Data - 3 Day Lag	0.503985	0.632773	0.511747	0.828775
Differenced Oil Data - 5 Day Lag	0.507362	0.533952	0.513393	0.556227
Differenced Oil Data - 10 Day Lag	0.522406	0.564877	0.534958	0.598341
Differenced Oil Data - 15 Day Lag	0.528870	0.654349	0.534216	0.844186
Differenced Oil Data - 20 Day Lag	0.503380	0.626962	0.510143	0.813174
Differenced Oil Data - 30 Day Lag	0.515077	0.663535	0.519038	0.919527
Differenced Oil Data - 40 Day Lag	0.494763	0.572917	0.508318	0.656325
Full Raw Feature Set - 1 Day Lag	0.529301	0.572899	0.536977	0.613971
Full Raw Feature Set - 2 Day Lag	0.502836	0.493256	0.558952	0.441379
Full Raw Feature Set - 3 Day Lag	0.497164	0.519855	0.500000	0.541353
Full Raw Feature Set - 5 Day Lag	0.506616	0.561345	0.507599	0.627820
Full Raw Feature Set - 10 Day Lag	0.515152	0.555556	0.506329	0.615385
Full Raw Feature Set - 15 Day Lag	0.515152	0.593651	0.505405	0.719231
Full Raw Feature Set - 20 Day Lag	0.511407	0.541889	0.504983	0.584615
Full Raw Feature Set - 30 Day Lag	0.522901	0.533582	0.554264	0.514388
Full Raw Feature Set - 40 Day Lag	0.471264	0.584337	0.457547	0.808333
Full Differenced Feature Set - 1 Day Lag	0.468809	0.270130	0.547368	0.179310
Full Differenced Feature Set - 2 Day Lag	0.510397	0.483034	0.514894	0.454887
Full Differenced Feature Set - 3 Day Lag	0.523629	0.556338	0.544828	0.568345
Full Differenced Feature Set - 5 Day Lag	0.496212	0.539792	0.513158	0.569343
Full Differenced Feature Set - 10 Day Lag	0.497154	0.515539	0.518382	0.512727
Full Differenced Feature Set - 15 Day Lag	0.521822	0.533333	0.543396	0.523636
Full Differenced Feature Set - 20 Day Lag	0.506667	0.407323	**0.559748**	0.320144
Full Differenced Feature Set - 30 Day Lag	0.506692	0.568562	0.500000	0.658915
Full Differenced Feature Set - 40 Day Lag	**0.533589**	0.571429	0.551020	0.593407

Table 5.5: Table of results for the different recurrent neural network models

Model	Accuracy	F_1 Score	Precision	Recall
Full Raw Feature Set - 1 Day Lag	0.491493	0.562602	0.504373	0.636029
Full Raw Feature Set - 2 Day Lag	0.461248	0.219178	0.533333	0.137931
Full Raw Feature Set - 3 Day Lag	0.489603	0.474708	0.491935	0.458647
Full Raw Feature Set - 5 Day Lag	0.512287	0.575658	0.511696	0.657895
Full Raw Feature Set - 10 Day Lag	0.490530	0.629986	0.490364	0.880769
Full Raw Feature Set - 15 Day Lag	0.492424	**0.658163**	0.492366	**0.992308**
Full Raw Feature Set - 20 Day Lag	0.526616	0.636496	0.512941	0.838462
Full Raw Feature Set - 30 Day Lag	0.507634	0.516854	0.539063	0.496403
Full Raw Feature Set - 40 Day Lag	0.486590	0.554817	0.461326	0.695833
Full Differenced Feature Set - 1 Day Lag	0.529301	0.506931	**0.595349**	0.441379
Full Differenced Feature Set - 2 Day Lag	0.517958	0.597156	0.514986	0.710526
Full Differenced Feature Set - 3 Day Lag	0.523629	0.505882	0.556034	0.464029
Full Differenced Feature Set - 5 Day Lag	0.520833	0.540835	0.537906	0.543796
Full Differenced Feature Set - 10 Day Lag	0.491461	0.480620	0.514523	0.450909
Full Differenced Feature Set - 15 Day Lag	0.531309	0.480000	0.570000	0.414545
Full Differenced Feature Set - 20 Day Lag	0.540952	0.554529	0.570342	0.539568
Full Differenced Feature Set - 30 Day Lag	0.531549	0.617785	0.516971	0.767442
Full Differenced Feature Set - 40 Day Lag	**0.541267**	0.621236	0.547486	0.717949

Table 5.6: Table of results for the different LSTM neural network models

Model	Accuracy	F_1 Score	Precision	Recall
Full Raw Feature Set - 1 Day Lag	0.519849	0.552817	0.530405	0.577206
Full Raw Feature Set - 2 Day Lag	0.506616	0.472727	0.570732	0.403448
Full Raw Feature Set - 3 Day Lag	0.483932	0.534923	0.489097	0.590226
Full Raw Feature Set - 5 Day Lag	0.525520	0.593193	0.521368	0.687970
Full Raw Feature Set - 10 Day Lag	0.518939	0.513410	0.511450	0.515385
Full Raw Feature Set - 15 Day Lag	0.498106	0.567700	0.492918	0.669231
Full Raw Feature Set - 20 Day Lag	0.505703	0.566667	0.500000	0.653846
Full Raw Feature Set - 30 Day Lag	0.484733	0.490566	0.515873	0.467626
Full Raw Feature Set - 40 Day Lag	0.498084	0.546713	0.467456	0.658333
Full Differenced Feature Set - 1 Day Lag	0.489603	0.427966	0.554945	0.348276
Full Differenced Feature Set - 2 Day Lag	0.527410	0.410377	0.550633	0.327068
Full Differenced Feature Set - 3 Day Lag	0.504726	0.545139	0.526846	0.564748
Full Differenced Feature Set - 5 Day Lag	0.515152	0.580328	0.526786	0.645985
Full Differenced Feature Set - 10 Day Lag	0.497154	0.535902	0.516892	0.556364
Full Differenced Feature Set - 15 Day Lag	0.523719	0.582363	0.536810	0.636364
Full Differenced Feature Set - 20 Day Lag	**0.586667**	0.597403	**0.616858**	0.579137
Full Differenced Feature Set - 30 Day Lag	0.523901	**0.647808**	0.510022	**0.887597**
Full Differenced Feature Set - 40 Day Lag	0.506718	0.536937	0.528369	0.545788

Table 5.7: Table of results for the different GRU neural network models

Chapter 6

Conclusion

In this thesis we have tackled the problem of predicting the direction of the day-ahead price change of Brent Crude Oil Front Month Futures Contracts by employing support vector machines, feed-forward neural networks and recurrent neural networks and their variations. It is clear to see that support vector machines performed consistently better than than the other two types of models, achieving the highest accuracy of 62.1% with the 'Raw data - 1 Day Lag' feature set which was significantly better than the results of both the feed-forward networks and the recurrent networks. Of the artificial neural networks, the recurrent neural network architecture made up of GRU cells with 'Full Differenced Feature Set - 20 Day Lag' performed the best, with an accuracy of 58.7% marginally beating the feed-forward network with the 'Full Differenced Feature Set - 2 Day Lag' feature set. Overall the artificial neural networks performed disappointingly which can be attributed to two main factors: not enough data for the models to train on and too shallow a search in the hyperparameter space. On the other hand, the support vector machines performed surprisingly well with most SVM models outperforming the majority of the ANN models implemented. In particular we saw that the S&P500 dataset provided great insight into the movements of Brent Crude Oil Futures, contributing significantly to the coefficients of the separating hyperplane. Unfortunately, the 'blackbox' nature of artificial neural networks allows us to draw little information about the factors that influence the weightings of the models, which is something that could be explored in further work.

Upon reflection, there are three directions I would like to take the research further. Firstly, a more intelligent, efficient method to find the optimum hyperparameters should be used in order to blah blah blah. Bayesian optimisation is a popular and effective method used in a vast number of machine learning problems and thus it makes for an ideal starting point for a more intelligent global optimisation method. Secondly, given more time, it would be beneficial to tune as many hyperparameters as possible. This includes using different values of K in our K-fold cross-validation, trying different permutations of Basic RNN, LSTM and GRU cells, the learning

rate, etc. Any parameter that was initially fixed can become a hyperparameter with enough time and resources. Finally, experimenting with more advanced kernel functions for the support vector machines, tailored to sequential data would be an interesting idea to take further. Salvi et al. [Salvi et al., 2021] suggested using the *Signature Kernel*, finding success in applying it to forecasting time series in machine learning tasks. This would be the next step in improving the current research proposed.

Appendix A

Technical Proofs

A.1 Proof of Proposition 4.1.2

Once again recalling from [Pakkanen, 2020], here we prove Proposition 4.1.2.

Proof. First note that $\ell = \ell(\boldsymbol{a}^r, \boldsymbol{y}) = \ell(\boldsymbol{g}_r(\boldsymbol{z}^r), \boldsymbol{y})$ so that, along with

$$\frac{\partial a_s^r}{\partial z_j^r} = \begin{cases} g_r'(z_j^r), & s = j, \\ 0, & s \neq j \end{cases}$$

and the chain rule gives

$$\delta_j^r = \frac{\partial \ell}{\partial z_j^r} = \sum_{s=1}^{O} \frac{\partial \ell}{\partial \hat{y}_s}(\boldsymbol{a}^r, \boldsymbol{y}) \frac{\partial a_s^r}{\partial z_j^r} = g_r'(z_j^r) \frac{\partial \ell}{\partial \hat{y}_j}(\boldsymbol{a}^r, \boldsymbol{y}), \quad j = 1, \ldots, O,$$

and (4.1.5) is the above in vector form.

If we view ℓ as a function of $\boldsymbol{z}^{i+1}$ and $\boldsymbol{z}^{i+1}$ as a function of $\boldsymbol{z}^i$ then the chain rule gives

$$\begin{aligned} \delta_j^i = \frac{\partial \ell}{\partial z_j^i} &= \sum_{s=1}^{d_{i+1}} \frac{\partial \ell}{\partial z_s^{i+1}}(\boldsymbol{z}^{i+1}) \frac{\partial z_s^{i+1}}{\partial z_j^i} \\ &= \sum_{s=1}^{d_{i+1}} \delta_s^{i+1} \frac{\partial z_s^{i+1}}{\partial z_j^i}, \quad j = 1, \ldots, d_i. \end{aligned}$$

Since

$$z_s^{i+1} = \sum_{u=1}^{d_i} W_{s,u}^{i+1} g_i(z_u^i) + b_s^{i+1}$$
$$= \sum_{u=1}^{d_i} W_{s,u}^{i+1} a_u^i + b_s^{i+1} \tag{A.1.1}$$

and thus

$$\frac{\partial z_s^{i+1}}{\partial z_j^i} = W_{s,j}^{i+1} g_i'(z_j^i)$$

through which

$$\delta_j^i = g_i'(z_j^i) \sum_{s=1}^{d_{i+1}} \delta_s^{i+1} W_{s,j}^{i+1} = g_i'(z_j^i) \sum_{s=1}^{d_{i+1}} (W^{i+1})'_{j,s} \delta_s^{i+1}, \quad j = 1, \ldots, d_i,$$

and (4.1.6) is the above in vector/matrix form.

Using the relationship shown in (A.1.1), we can see that

$$\frac{\partial z_s^i}{\partial b_j^i} = \begin{cases} 1, & s = j, \\ 0, & s \neq j, \end{cases} \qquad \frac{\partial z_s^i}{\partial W_{j,k}^i} = \begin{cases} a_k^{i-1}, & s = j, \\ 0, & s \neq j, \end{cases}$$

and so the chain rule gives

$$\frac{\partial \ell}{\partial b_j^i} = \sum_{s=1}^{d_i} \frac{\partial \ell}{\partial z_s^i} \frac{\partial z_s^i}{\partial b_j^i} = \delta_j^i, \quad \frac{\partial \ell}{\partial W_{j,k}^i} = \sum_{s=1}^{d_i} \frac{\partial \ell}{\partial z_s^i} \frac{\partial z_s^i}{\partial W_{j,k}^i} = \delta_j^i a_k^{i-1},$$

which completes the proof of Proposition 4.1.2. □

Bibliography

[10., 2017] (2017). Solving time series classification problems using support vector machine and neural network. *Int. J. Data Anal. Tech. Strateg.*, 9(3):237–247.

[Boser et al., 1992] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press.

[Chen, 2018] Chen, G. (2018). A gentle tutorial of recurrent neural network with error backpropagation.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation.

[Clevert et al., 2016] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2016). Fast and accurate deep network learning by exponential linear units (elus).

[Education, 2020] Education, I. C. (2020). Recurrent Neural Networks. `https://www.ibm.com/cloud/learn/recurrent-neural-networks`.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Graves, 2012] Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385. Springer-Verlag Berlin Heidelberg.

[Guyon et al., 2002] Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. (2002). Gene Selection for Cancer Classification using Support Vector Machines. *Machine Learning*, 46:389–422.

[Hochreiter, 1991] Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen*. Diploma thesis, Technical University Munich, Institute of Computer Science.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Hsu, 2011] Hsu, C.-M. (2011). Forecasting stock/futures prices by using neural networks with feature selection.

[Ince and Trafalis, 2008] Ince, H. and Trafalis, T. B. (2008). Short term forecasting with support vector machines and application to stock price prediction. *International Journal of General Systems*, 37(6):677–687.

[James et al., 2013] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). An Introduction to Statistical Learning: with Applications in R. *Springer Texts in Statistics*.

[Jordan and Thibaux, 2004] Jordan, M. and Thibaux, R. (2004). The Kernel Trick. *CS281B/Stat241B: Advanced Topics in Learning & Decision Making*.

[Maas, 2013] Maas, A. L. (2013). Rectifier nonlinearities improve neural network acoustic models.

[McCulloch and Pitts, 1943] McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Miophysics*, 5:115–133.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *ICML*.

[Pakkanen, 2020] Pakkanen, M. (2020). *Deep Learning Lecture Notes*. Department of Mathematics, Imperial College London.

[Patle and Chouhan, 2013] Patle, A. and Chouhan, D. S. (2013). Svm kernel functions for classification. In *2013 International Conference on Advances in Technology and Engineering (ICATE)*, pages 1–9.

[Raschka, 2020] Raschka, S. (2020). Model evaluation, model selection, and algorithm selection in machine learning.

[Rumelhart et al., 1986] Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.

[Salvi et al., 2021] Salvi, C., Cass, T., Foster, J., Lyons, T., and Yang, W. (2021). The signature kernel is the solution of a goursat pde.

[Samsudin et al., 2010] Samsudin, R., Shabri, A., and Saad, P. (2010). A Comparison of Time Series Forecasting using Support Vector Machine and Artificial Neural Network Model. *Journal of Applied Sciences*, 10:950–958.

[Santambrogio, 2016] Santambrogio, F. (2016). Euclidean, Metric, and Wasserstein gradient flows: an overview.

[Shashua, 2009] Shashua, A. (2009). Introduction to machine learning: Class notes 67577.

[Waldow et al., 2021] Waldow, F., Schnaubelt, M., Krauss, C., and Fischer, T. G. (2021). Machine learning in futures markets. *Journal of Risk and Financial Management*, 14(3).

HUMPHREYS_JONAH_01971479

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62