Forecasting the Directional Movement of a Stock's Realised Volatility Using Recurrent Neural Networks

William Kollard

Supervised by: Georgios Karagiannis

Department of Mathematical Sciences, Durham University

May 2, 2024

Abstract

This paper presents a comparative analysis of recurrent neural network models and explores their capabilities by forecasting the directional movement of a stock's realised volatility. Our study explores the limitations of the feedforward neural network for time series forecasting and proposes the recurrent neural network model as an alternative. We examine the prevalence of the vanishing gradient problem in the basic recurrent neural network model and learn how the gating mechanism of the long short-term memory neural network is designed to mitigate this issue. This paper proposes a novel approach to training a recurrent neural network for the purpose of intrahour volatility forecasting by considering the time constraints in a practical financial setting. Through empirical results, we showed that the learning efficiency of a gated recurrent unit can be increased by modifying the gating mechanism to combine the update and reset gate. Finally, we study the relationship between temporal correlations within a time series dataset and recurrent neural network performance. Such correlations were subsequently leveraged to build a custom feature set for the purpose of intraday volatility forecasting. Throughout the paper we carry out the complete pipeline of a machine-learning process, beginning with the raw price data for the AAPL stock. After performing data cleansing, scaling and reconstruction, we train our neural network models on the refined datasets.

Preface

Plagiarism Declaration

This piece of work is a result of my own work and I have complied with the Department's guidance on multiple submission and on the use of AI tools. Material from the work of others not involved in the project has been acknowledged, quotations and paraphrases suitably indicated, and all uses of AI tools have been declared.

Acknowledgements

I would like to thank my supervisor Georgios Karagiannis for his advice and guidance across the year and for providing new takes and suggestions throughout the project. I have thoroughly enjoyed exploring different areas of machine learning, neural networks and volatility, and this project has inspired me to pursue further research into these fields. I am also grateful to the staff in the Department of Mathematical Sciences at Durham University for providing excellent help and teaching throughout the entirety of the course.

Contents

Chapter 1

Introduction

Volatility Forecasting plays a vital role in the world of financial markets. In the (almost) unpredictable realm of the stock market, the ability to obtain accurate predictions of a stock's future realised volatility can be of great use to finance practitioners. Firstly, one of the most common motivations is due to the role of volatility in options pricing, as there is a direct relationship between the expectation of a stock's volatility and the price of an option. Secondly, understanding volatility movements is key for market makers, who make up a large proportion of daily traded volumes. There is a fundamental relationship between an asset's volatility and liquidity, and market makers are the driving force behind this. They will use forecasts of market volatility to adjust their bid-ask spread in order to maximise profit. Finally, realised volatility estimates are key for managing risk for trading strategies, as one may want to reduce their exposure to the market in periods of higher volatility.

1.1 Current Literature

A pioneer of such quantitative modelling is the Medallion Fund, managed by famous Mathematician, Jim Simons, which has made over $100 billion for its owners from 1988 to 2018. This was achieved by leveraging machine learning models and extensive data sets, to make similar forecasts that we will attempt in this paper. Hence, along with the fundamental role that volatility plays, there has been significant interest in this field to both academics and financial practitioners. Research into volatility modelling has especially evolved over the past few decades, with a particular focus on forecasting realised volatility over larger time periods, such as daily, weekly or monthly. For example, [31] discusses the use of traditional time series methods, such as GARCH and ARIMA to forecast daily volatility and proposes a hybrid model combining both methods. [11] introduces recurrent neural networks as a way to outperform the traditional methods and focuses on forecasting monthly realised volatility. There has been less literature on forecasting realised volatility over shorter time periods, such as the next hour or day. However, [38] attempts to forecast intraday volatility by leveraging a variety of machine learning models, including the long short-term memory neural network.

In this project, we will attempt to forecast the directional movement of a stock's realised volatility through the use of recurrent neural networks. We will focus on forecasting **intraday** and **intrahour** realised volatility. We aim to extend previous literature in two ways. Firstly, by developing a custom made dataset, including features which, to the best of our knowledge, have not previously been considered to forecast intraday realised volatility. We provide an initial feature set motivated by previous literature and our own understanding of volatility. This is narrowed down through a feature analysis, which explores temporal correlations within the dataset. Secondly, we take a novel approach in forecasting **intrahour** realised volatility. We develop a modified version of the gated recurrent unit neural network, by combining the update and reset gate, with the purpose of reducing the model's training time.

Throughout the development of the models in this project, there has been considerable care in avoiding look-ahead bias, which could lead to inflated results. The out-of-sample results produced in Chapter 7 accurately reflect how these models would perform on genuine unseen data (although it's important to note potential performance degradation may occur due to model decay). All of the code used throughout the project can be found on this Github page [25].

Alongside extending previous literature, we will develop an understanding of several neural network models, with a focus on recurrent neural networks. We will explore when these models outperform the feedforward neural network, by considering their architecture, training process and different data types.

1.2 Project Objective

Before delving into our neural network models, we first explain the objectives of our application, providing a context to use our models in. We now define a handful of financial concepts, allowing us to fully understand the application of this paper.

Definition 1.2.1. Stock - A financial asset which represents ownership of a corporation. These assets can be bought and sold at an agreed upon price, by two parties, on an exchange.

Definition 1.2.2. Option - A financial asset which derives its value based off the underlying stock price. These assets can be bought and sold at an agreed upon price, by two parties, on an exchange and they expire after a set time period, D.

Definition 1.2.3. PnL - The change in value of a trader's portfolio. It refers to how much profit or loss a trader makes from owning stocks or options in their portfolio.

Definition 1.2.4. Volatility - A statistical measure which quantifies the amount an asset's price has varied over a given time period.

Example 1.2.1. In this example, we develop an intuitive understanding of what an asset's volatility is. Figure 1.1 displays the price of two different stocks, Stock A and Stock B, over a 20 day period. Both stocks start and end at a value of £160, however, we can see that stock A exhibits a greater degree of fluctuation in its price over stock B. At the end of the 20 day period, we would describe Stock A as having exhibited a greater realised volatility than Stock B.

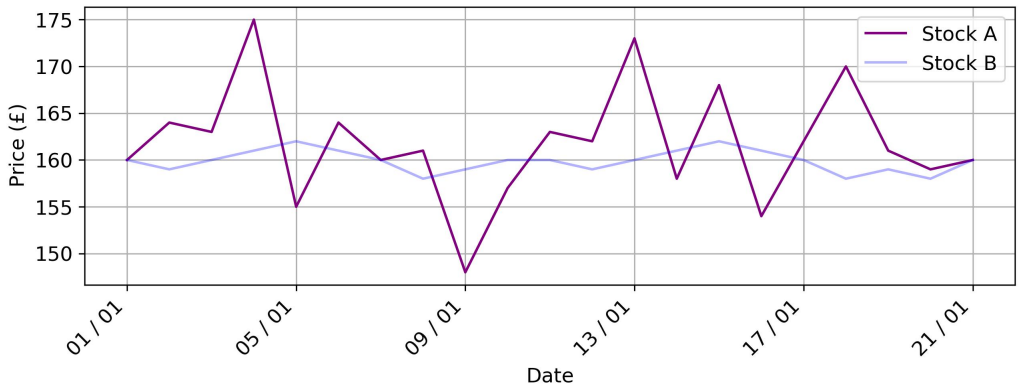


Figure 1.1: This plot shows the price of a high volatility (A) and low volatility stock (B) over a 20 day period.

In this project we will be forecasting realised volatility over two different timeframes, which we define below.

Definition 1.2.5. Average Hourly (Intraday) Realised Volatility, $\sigma(t)$, within day, t is defined as:

$$\sigma(t) = \sqrt{\frac{1}{K}\sum_{i=1}^{K} r_{t_i}^2}, \tag{1.1}$$

where,

- $r_{t_i} = \log(\frac{S_i}{S_{i-1}})$ is the log return between hour, $i-1$ and hour, i
- S_i is the price of the stock at hour, i
- K is the number of hourly price points we have on day, t for the stock price.

Definition 1.2.6. Cumulative Hourly (Intrahour) Realised Volatility, $\sigma(t)$ within hour, t is defined as:

$$\sigma(t) = \sqrt{\sum_{i=1}^{K} r_{t_i}^2}, \tag{1.2}$$

where,

- $r_{t_i} = \log(\frac{S_i}{S_{i-1}})$, is the log return between minute, $i-1$ and minute, i
- S_i is the price of the stock at minute, i
- K is the number of minutely price points we have within the hour, t for the stock price.

Equation (1.1) contains a scale factor, $\frac{1}{K}$, which is important due to the nature of market data. Over the past 20 years the market may not have been open for the same time period every day. Therefore, the number of hours of price data we have available for each day may vary. For example, on one day, we may have data from 9am to 4pm, where as on another day we have data from 9am to 5pm. This requires us to scale our volatility to the number of hours of price data we have, leading to unbiased comparisons. The equation calculates the average hourly volatility, indicating the extent of price movement within a day (although scaled down). We will refer to this as **intraday (realised) volatility**.

Equation (1.2) does not contain this scale factor as within our high frequency dataset, we have a full 60 minutes of data available for each hour. This equation indicates the extent of price movements within each hour. Hence, we will refer to this as **intrahour (realised) volatility**.

In this project as opposed to forecasting the absolute value of future realised volatility (within the next day or hour). Instead, we will be forecasting the directional movement of the day-ahead or hour-ahead realised volatility (i.e. we will be predicting whether the volatility will increase (1) or decrease (0) from today to tomorrow, or this hour to the next). To provide clarity of our application, the following two scenarios explain how we would forecast directional movement of future intrahour volatility and future intraday volatility.

Scenario 1 - Forecasting future intrahour volatility

Let's imagine the time is 2:59pm on the 2nd May 2024. We currently have all the stock data available for this hour and we have observed the intrahour realised volatility within this hour, assume it came out at 0.0032. Our aim now is to forecast whether the intrahour realised volatility within the hour 3:00pm to 4:00pm will increase or decrease from the current hour's value of 0.0032.

Scenario 2 - Forecasting future intraday volatility

Let's imagine it is currently 5pm on the 2nd May 2024 (the stock market has just closed for the day). We currently have all the stock data (such as returns and trading volume) available for this day and we have observed the intraday realised volatility within the day, assume it came out at 0.021. Our aim now is to forecast within the 3rd May 2024, whether its intraday realised volatility will increase or decrease from today's value of 0.021.

This formulates our problem as a *binary classification* problem, meaning for each data point our neural network models will be classifying each observation into one of two categories; increase (1), or decrease (0).

1.3 Why is Forecasting Realised Volatility important?

The first question we ask ourselves is why we would want to forecast the directional movement of realised volatility. We briefly mentioned previously that there are three main purposes for modelling volatility: options trading, liquidity provision and risk management. To fully understand these motivations, we describe two notions of volatility.

There are numerous ways volatility can be measured and these are based off two notions of volatility; **realised volatility** and **implied volatility**.

1. **Realised Volatility** refers to the assets past price movements, over a set period of time. It is calculated based off price data which has already been observed. Therefore, future realised volatility is determined based on price data observed during a future period. As stated, this is what we will be forecasting in our project.

2. **Implied Volatility** is future looking and it refers to the market's *expectation* of an asset's volatility over a future time period. This is not something that can be observed, but this expectation can be derived from the market price of an option through the Black-Scholes option pricing formula [28]. This formula shows how the implied volatility of a given stock, σ_{impl}, relates to the market price of the option (remebering that the option derives its value from the underlying stock). It is given by:

$$V(t) = S_t\Phi(d_1) - Ke^{-rD}\Phi(d_2), \tag{1.3}$$

where,

$$d_1 = \frac{\log(\frac{S_t}{K}) + (r + \frac{\sigma^2_{\text{impl}}}{2})D}{\sigma_{\text{impl}}\sqrt{D}} \tag{1.4}$$

and

$$d_2 = d_1 - \sigma_{\text{impl}}\sqrt{D}. \tag{1.5}$$

The remaining variables are defined as follows:

- $V(t)$ - Price of the option at time, t
- S_t - Current market price of underlying stock
- D - Time to expiry of the option
- K, r - Known constants
- Φ - Standard Normal Distribution.

Now that we understand these notions of volatility, the next subsection shows how we could theoretically profit from buying an 0DTE AAPL option, after forecasting realised intraday volatility to increase.

Definition 1.3.1. 0DTE AAPL Option - An option which derives its value based off the underlying AAPL stock. The option expires at the end of the trading day, hence D represents the remaining length of the trading day after the option was bought.

The trading volume of 0DTE options have grown exponentially in popularity over the last few years. More information on how these options are priced can be found in [5].

1.3.1 Options Trading

Volatility modelling is at the core of options trading, since the option's price is driven mostly by the stock's implied volatility. In order to beat the market, we must make a trade based on information, before it is priced into the market (i.e. reflected by the market's *expectation*). By having an accurate estimate of future intraday realised volatility of the AAPL stock we can make a profitable delta hedged trade on the 0DTE AAPL option. Intuitively, in the following trading strategy, we would make a profit if the fair price of volatility (our forecast for intraday realized volatility) was greater than the market's price of volatility (intraday implied volatility) and we buy the option.

Definition 1.3.2. Delta Hedging - A strategy used for trading options to neutralise the risk associated with price movements of the underlying stock.

If we bought a (continuously delta hedged) 0DTE AAPL option, the associated PnL at the end of the day, V_D would be given as [7]:

$$\begin{aligned} dV &= \frac{1}{2}(\sigma(t)^2 - \sigma_{\text{impl}}^2)\tilde{\Gamma}(t)dt \\ \implies V_D &= \int_0^D \frac{1}{2}(\sigma(t)^2 - \sigma_{\text{impl}}^2)\tilde{\Gamma}(t)dt, \end{aligned} \tag{1.6}$$

where the *cash gamma* reflects how the underlying stock price is *hedged out* and is defined as

$$\tilde{\Gamma}(t) = S_t^2 \frac{\partial^2 V}{\partial S_t^2}(t). \tag{1.7}$$

Equation (1.6) tells us that our PnL at the end of the day depends on the difference between the AAPL stock's realised intraday volatility throughout the day, and the AAPL stock's initial implied volatility, σ_{impl}^2, when we bought the option (which we know from the option's initial price). It is well known that implied volatility reflects the current realised volatility level [27]. Therefore, if we have forecasted intraday realised volatility to increase from the previous day and the implied volatility has not increased, this would lead to a profitable trade.

Note. Similarly, by adjusting our models and dataset to forecast the absolute value of realised intraday volatility, we could forecast the *amount* of profit we would make on this trade.

1.3.2 Liquidity Provision (Market Making)

Liquidity providers (market makers) are a crucial mechanism of ensuring that markets are efficient and have sufficient liquidity in a variety of market regimes. Liquidity providers act as a middle man, earning the bid-ask spread on trades, in return for holding inventory. That is to say, suppose AAPL stock is trading at around \$10.00 per share (mid-price), then a liquidity provider may "quote a spread" of \$9.95 to \$10.05. That means that prospective buyers can buy an AAPL share from the liquidity provider for \$10.05 (the market maker's ask price) and prospective sellers can sell an AAPL share to the liquidity provider for \$9.95 (the market maker's bid price). Therefore, the liquidity providers can make \$0.10 by buying from one market participant and selling to another, and they can make their trades more attractive to prospective buyers/sellers by tightening their bid-ask spread.

It may be natural to then ask why do we need the middle man (liquidity provider) at all and not just pair trader 1 with trader 2? The idea is that the market maker smoothes out these transactions and takes inventory risk to hold the stock regardless of market direction. In periods of high volatility and market panic, the liquidity providers are able to support the market by buying the stock from those who are panic selling, providing an important role in market microstructure. However, in such periods of heightened volatility, the market maker would require to be compensated more for holding the inventory, and this is done by *widening* their bid-ask spread. We can show that in fact the optimal bid-ask spread is proportional to the expected variance of the underlying price. Hence, if we have a good estimate for this variance (volatility), then a liquidity provider is able to accurately adjust their bid-ask spread accordingly. In Appendix A, we provide a detailed methodology of Kyle's model [26] for trading under information and how a liquidity provider adjusts their quotes according to the asset volatility and total order flow.

1.3.3 Risk Management

For any trading strategy, it is essential that a trader is able to not only capture positive expected returns, but also manage them accordingly. *Risk* is measured as an exposure to a particular factor that is out of our (direct) control, however we are able to hedge against these risks in other ways. The main risk that traders face is that of price uncertainty (volatility), where strategies can utilise volatility scaling, risk parity or mean-variance optimisation to mitigate those risks. In all of these approaches, an accurate measure of volatility is required in order to adjust trading strategy positions and minimise risk.

1.4 Project Layout

Now that we understand the application of our project, we will discuss the project's layout and how each chapter contributes. Chapter 2 describes what machine learning is and we will develop a general framework for our methodologies. Chapter 3 covers the choice of feature variables that will be used to train our neural network models and the data engineering process that produced our datasets. In Chapter 4, we introduce the feedforward neural network model and explore different optimizer algorithms. We will build on this in Chapter 5, by introducing the recurrent neural network model and its variants. In Chapter 6, we will aim to reduce the training time of the recurrent neural network by altering the gating mechanism in one of the models. Finally, in Chapter 7 we will apply everything that we have learnt to forecast the directional movement of intraday realised volatility.

Our primary objective for this paper is to accurately forecast the directional movement of intraday realised volatility. Throughout Chapters 5 and 6, we will forecast intrahour volatility, serving primarily as a practical example to understand and compare our neural network models, rather than to achieve the best possible outcomes. In contrast, Chapter 7 will focus on optimising our models and obtaining the highest possible performance to forecast intraday volatility.

Chapter 2

Machine Learning Set-up

Within the framework of machine learning, two of the main types of learning that can occur are **supervised** and **unsupervised** (alternatively to **generative** learning). In a supervised learning setting, we have feature variables $X_1, X_2, ..., X_p$ and response variable, Y for n observations. The aim is to predict the value of a response from observed feature values in an observation. In an unsupervised learning setting, we are only interested in the feature variables. The goal here is to discover patterns within the features [21]. In our project, as our objective is to predict an increase (1) or decrease (0) in realised volatility based off features of the stock, then we are dealing with a supervised learning problem.

2.1 Supervised Learning

The general objective for a supervised learning problem is to train a model, h to learn the mapping from the observed feature variables, $\mathbf{X}$, which we call the input data, to the observed response variables, $\mathbf{y}$, which we call the output data. This model, h could then be used to predict the response values on unseen observations. The equations below represent a data set with n observations, containing 5 feature variables, $\mathbf{X}$ and a response variable, $\mathbf{y}$:

$$\mathbf{X} := \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \vdots \\ \mathbf{x_n} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & x_{n4} & x_{n5} \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \tag{2.1}$$

2.1.1 Classification vs Regression Problems

Within the supervised learning setup there are two types of problems that one may want to solve, these are **regression** and **classification** problems.

In a regression problem, our response variable is treated as continuous, implying that $y_t \in \mathbb{R}$ for all $t = 1, ..., n$. This type of problem occurs when we wish to predict a continuous quantity. If our objective was to predict the absolute value of realised volatility, then this would be a regression problem.

A classification problem differs in nature. In these problems the response variable is categorical. Therefore as opposed to predicting an absolute value, the model, predicts which category an observation, $\mathbf{x_t}$ lies in. For a given data observation, $\mathbf{x_t}$, the model, h assigns it to one of K different categories. In this project, as our models only have two classes to choose from, this formulates our problem as a *binary classification* problem. The equations below describe how a supervised model predicts which class an observation lies in:

$$\hat{y}_t = \begin{cases} 0 & \text{if } h(\mathbf{x_t}) < threshold \\ 1 & \text{if } h(\mathbf{x_t}) \geq threshold \end{cases}, \tag{2.2}$$

where,

$$h : \mathbb{R}^p \to [0, 1], \quad \mathbf{x_t} \mapsto h(\mathbf{x_t}). \tag{2.3}$$

Definition 2.1.1. Threshold Value - The cutoff point at which the probability score, $h(\mathbf{x_t})$, assigned by the model is converted into a class label $\hat{y}_t$.

$h(\mathbf{x_t})$ is the probability that the observation $\mathbf{x_t}$ lies in the increase (1) class. The actual classification of the data point is then made according to (2.2), where $\hat{y}_t$ is the predicted class for the observation. Unless stated otherwise, we will assume a *threshold* value of 0.5. All of the models we explore in this project will assume this general form.

2.2 Performance Metrics

In order to see which of our models are most effective, we must introduce performance metrics to evaluate how well they can predict our two classes. The most common performance metric for this type of problem is **Accuracy**, which is defined as:

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

When our models make a prediction, if it correctly predicts an increase or decrease in volatility this is called a True Positive, TP and True Negative, TN. However, if the model predicts the volatility to increase and it actually decreases this is called a False Positive, FP. Similarly, if our model incorrectly predicts a decrease, this is called a False Negative, FN.

Although accuracy provides a useful general indicator of a model's performance, particularly when we have a balanced number of data points in each class, it may not be very informative depending on the context of the problem we are solving. For example, we may train a model, which produces an out-of-sample accuracy of 85%. Generally speaking, this model appears to perform well. However, if we then learnt that all the incorrect classifications occurred due to false positives, this may not be suitable for a trader that will incur great losses by incorrectly predicting an increase in volatility.

We can see that we're not able to gain this insight from the accuracy metric alone, so for this project we will introduce new performance metrics that incorporate relevant information, depending on which type of finance practitioner is using the model. Table 2.1 defines 3 additional performance metrics, each being suitable for a different type of role.

Performance Metric	**Precision**	**Sensitivity (Recall)**	**Specificity**
Definition	$\frac{\text{TP}}{\text{TP+FP}}$	$\frac{\text{TP}}{\text{TP+FN}}$	$\frac{\text{TN}}{\text{TN+FP}}$

Table 2.1: Performance Metrics

Precision measures the proportion of true increases out of all increases predicted by the model. If we have a high precision then the model makes a small number of false positive predictions, therefore it is unlikely the model will predict volatility to increase and then it actually decreases. If an options

trader mistakenly thinks that the volatility of a stock will increase, they may increase their proportion of AAPL options in their portfolio. If the volatility then decreases, these options will lose value and the portfolio manager will incur a greater loss, due to the asymmetricity of the spread between realised and implied volatility. These type of investors are less concerned about missing a few investment opportunities, provided that they maintain a high quality of investment on average. Therefore precision is a suitable metric for these types of investors.

Sensitivity measures how well the model identifies true increases. The higher the sensitivity the greater the volume of actual increases in volatility are identified by the model. High frequency traders (HFTs) rely on executing as many trading opportunities as possible because their profit margins on each trade are generally very small. HFTs rely on maximizing volume and therefore identifying true increases in volatility, making sensitivity a suitable performance metric for this type of trader.

To further elaborate, precision is focused more on not being wrong, where as sensitivity is focused on maximizing how many increases in volatility are identified.

Specificity is similar to precision but instead measures how well the model identifies true decreases, i.e. it is accurate at identifying when volatility will actually decrease. This may be suitable for market makers, who are required to execute large volumes of trades at once and at the best price. If a market maker can identify that the stock's volatility will decrease, they may tighten their spreads, reduce their trade volume or delay part of the execution.

2.2.1 Receiver Operator Characteristic Curve

A Receiver Operator Characteristic (ROC) Curve provides a graphical representation of how well our model performs over the range of threshold values. It works by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) for all possible threshold values. A perfect classifier would cross through the top left corner, as it correctly classifies every data point from both classes. A classifier which categorises every data point incorrectly would pass through the bottom right corner of the curve. A classifier which has zero predictive power, would pass through the diagonal line on the curve, indicating the same predictive power as a random guess.

Throughout the project we will assume a threshold value of 0.5, which means if $h(\mathbf{x_t})$ is less than 0.50, the observation, $\mathbf{x_t}$ will be assigned to the decrease (0) category, otherwise it will be assigned to the increase (1) category. In reality, depending on which type of finance practitioner we are, we may be focused more on prioritizing one performance metric over another, which can be achieved by modifying our threshold value. The ROC curve helps us to achieve the following:

1. Choose a threshold value to prioritize a certain performance metric
2. Assess the overall performance of the model through the AUC

Example 2.2.1. An example of an ROC Curve for one of our models is shown in Figure 2.1. Assume that we were a sell-side market maker, and we would like to maximise the specificity of our model. From the curve, we can see that a specificity value of 0.85 can be achieved for a threshold of 0.59, whilst still maintaining a sensitivity of 0.50. The ROC helps us to choose a threshold value which increases the number of true negatives we identify, while still achieving a reasonable sensitivity.

The second aspect of the ROC curve, which provides meaningful insight is the area under the curve, called the **AUC**. As we learnt above, with metrics such as specificity and sensitivity, they may change significantly with only small changes in the threshold. This means if we were to compare models on this metric alone, it could provide an unfair comparison, as a *weaker* model may have a greater sensitivity than a *stronger* model for a given threshold value. The AUC provides a measure across all threshold values, allowing us to identify the best performing models. The AUC in Figure 2.1 is 0.76.

For reference, an AUC of 1 would represent a perfect model for any threshold, and an AUC of 0.5 represents a model, which is no better than a random guess.

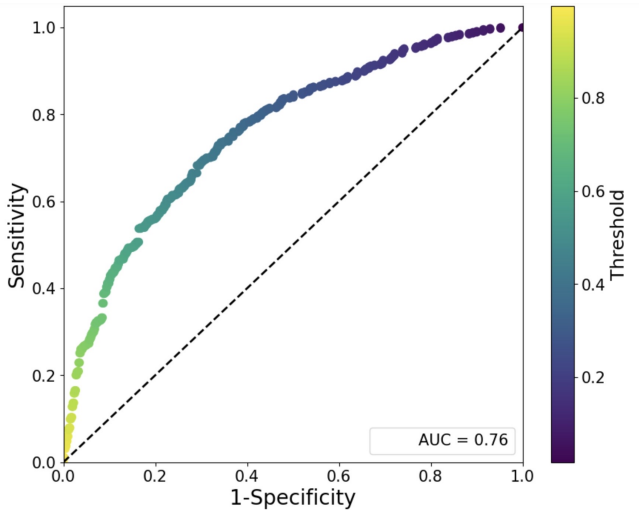


Figure 2.1: ROC Curve

2.2.2 In-Sample vs Out-of-Sample

As implied previously, the goal in a binary classification problem is to train a model using observed data (training data), which can then be used to predict the classes of unseen data (test data) based on the values of their features.

The main challenge in machine learning is to develop a model which can make accurate predictions on data it has never seen before. If a model can do this, then it is practically useful. In order to test this, we split our data, $(\mathbf{X}, \mathbf{y})$ into a training set, $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ and a testing set, $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$. The ratio in which we split our data is called the train-test split. Throughout this project we will use 80% for our training and 20% for our testing. Every model contains parameters, $\mathbf{W}$. The training set is used to enable the model to learn patterns in the data (which will be discussed in detail in subsequent chapters). It uses the training data to adjust its internal parameters, $\mathbf{W}$, and minimize errors in the training predictions. The hope is that the same model can then be used to make accurate predictions on the test set. However, what may happen is the model learns intricate details of the training data, leading to an exceptionally high performance on the training predictions, but then when it comes to making predictions on new data, it is unable to generalise - this is called **overfitting**.

Definition 2.2.1. In-Sample Performance - the performance of a machine learning model when tested on the dataset it was trained on.

Definition 2.2.2. Out-of-Sample Performance - the performance of a machine learning model when tested on new, unseen data.

The key is to find the balance between a model which can perform well on training data, but then can **also** perform well on the test data. In order to prevent overfitting, we must validate our model. In addition to parameters, a model also has hyperparameters, θ. These values are not learnt from the data. Instead they are assigned to a model, before it is trained, during the validation stage. Hyperparameters refer to features of the model, such as number of neurons in a neural network, or the value of regularisation parameters. These will determine how well the model fits to the training data. However, the hyperparameter values will be optimal for the values which achieve the highest performance on unseen data. Clearly, we cannot use our test set to obtain these values. Instead we must use a validation set. The validation set, $(\mathbf{X}_{\text{val}}, \mathbf{y}_{\text{val}})$ is formed from our training set and is used

to optimise the hyperparameter values.

Our validation set can be formed in 2 ways:

1. Split the training data into an $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ (80% of original training data) and $(\mathbf{X}_{\text{val}}, \mathbf{y}_{\text{val}})$ (20% of original training data)
2. Cross-Validation

Remark - A common practical mistake that occurs, when using the Python Keras library to train models is using the test set to validate the models, as opposed to a proper validation set. Then mistakenly taking the best validation accuracy to be the final test accuracy. The problem with doing this is there is a risk of overfitting the model to the test set. Although the model hasn't been trained on the test data, the hyperparameters have still been optimised using the test data, and we still may experience a drop in performance when testing the model on true unseen data.

2.3 Cross Validation

Cross validation allows us to develop validation sets from our training data, which the model views as *unseen* data. We can then tune the model's hyperparameters by choosing the combination of hyperparameter values which achieve the greatest validation performance. Given these values, the final model is trained on the $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$, giving the in-sample performance, it is then tested on $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$, giving the out-of-sample performance. In this section, we will explore 3 different cross-validation techniques, starting with the most basic.

2.3.1 K-Fold Cross Validation

When the amount of available data is limited we may want to use cross-validation as opposed to further splitting our training data. K-fold cross validation is the most basic technique we will explore and this method works as follows:

1. Randomly split our $\mathbf{x_{train}}$ dataset into k equally sized groups, which we called folds
2. Initialise hyperparameter values (chosen from grid search or random search optimizer)
3. For $i = 1, ..., k$:
   - Treat fold, i as *unseen* data
   - Train the model on the remaining $k - 1$ folds
   - Compute the performance metric on fold i to obtain $\text{performance}_1, ..., \text{performance}_k$
4. Calculate the Validation performance through:

$$\text{Validation Performance} = \frac{1}{k} \sum_{i=1}^{k} \text{performance}_i. \tag{2.4}$$

The validation accuracy obtained through K-fold cross validation provides us with an estimate of how our model will perform, with a given set of hyperparameter values, on the test data. Although generally this method helps us to improve our models, it may not be the best method when working with time series data.

Look-ahead Bias

In the following chapter, we will learn that the data sets we will use to train our neural network models are time series. The data is collected at consistent intervals in time and each observation is related to previous and subsequent observations. An important phenomenon we must be aware of when training our models on time series data is **Look-ahead bias**. This is a type of bias that can occur due to information that would not have been known at the time being used to make predictions. As referred to in [15] this bias can falsely inflate the returns of a portfolio, or in our case, it can lead to more accurate volatility predictions than would have been made in reality.

Imagine we had a stock price data ranging from 2005 to 2024, and we did a random train-test split to split our data, we will end up making predictions using our model from data that would not have existed at the time. i.e. if we made a prediction for 2016 it may include information from 2018 and hence we have introduced a bias. Generally, when we split a dataset into a training and test set, we randomly choose 80% of the data to become the training set and the remaining 20% becomes the test set. However, when splitting a time series data set, we must maintain chronological order. As we learn in the next chapter, the data we will use for training our neural network models will be from the year 2005 to 2020 and the data we use for testing will be from 2021 to 2024.

With this in mind, the main drawback to k-fold cross validation is that it doesn't account for the chronological ordering of the data. When k-fold cross validation separates the data into folds, it does this randomly, leading to future data being used to predict historical events in each fold. This can cause our validation accuracy to be overinflated, meaning way may incorrectly choose our hyperparameters based off false information.

2.3.2 Time-Series Cross Validation

Here, we will introduce 2 new cross-validation methods, which have been specifically developed for time series data.

Nested Cross-Validation

In nested cross-validation, as opposed to splitting our training data into k-equally sized groups, instead we use an expanding time frame as our training data, followed by a fixed time frame used for validation [30]. This can be understood through Figure 2.2. Our training data being split into 4 consecutive and expanding groups. This ensures that the validation occurs on a time-frame that has not been seen by the trained model, removing an look-ahead bias. The disadvantage of the expanding window approach is that during fold 1, our model may experience overfitting due to a lack of training data, especially in comparison to the final fold.

Walk-Forward Cross-Validation

The walk forward cross-validation method, as shown in Figure 2.3, uses a rolling time frame as opposed to an expanding time frame [35]. In this algorithm, it would train a model on the first 54 months and then validate it on the next 13 months, before then moving forward and repeating the procedure on the next 67 months. This will completed until we reach the end of the training dataset. Unlike in nested-cross validation by keeping the training data frames the same size, this helps to choose hyperparameters which neither underfit nor overfit the data. We will therefore use this cross-validation method when validating our models.

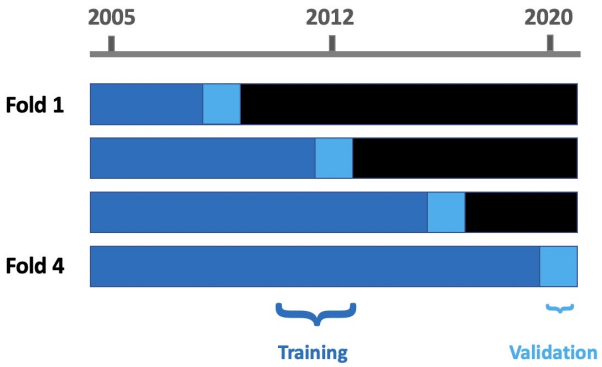


Figure 2.2: Nested cross-validation on training data from 2005 to 2020. The validation set is made up of data which comes chronologically after the training set.

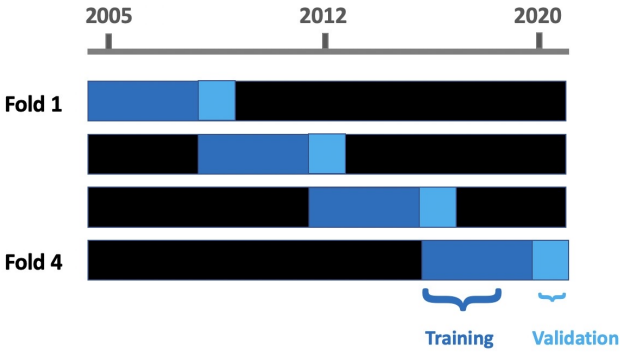


Figure 2.3: Walk-forward cross-validation on training data from 2005 to 2020. The validation set is made up of data which comes chronologically after the training set.

Hyperparameter Tuning

As we have learnt, hyperparameter tuning is an important aspect to training our model. This is particularly true with neural networks, where there are several hyperparameters to consider. If we were to manually try each combination of hyperparameters for each model it would incredibly time-consuming and computationally expensive. There are two methods which can be used to conduct this process, namely **Grid Search** and **Random Search**.

In the grid search method we pre-define a set of possible values for each hyperparameter, then the algorithm tests every combination to find the best performing model. The exhaustive nature of the search, ensures that the best performing model will be found as long as the values are within the search space. However, the downside is it becomes exponentially more time-consuming as our search space increases.

The difference with the random search method is instead of specifying individual values, we give a range of values for the search space. We then define a specified number of iterations we want the algorithm to complete and values for the hyperparameters are based on a probability distribution of our choice. For example, we would use a uniform distribution if we have no knowledge over which hyperparameter value we think is optimal, but we may use a normal distribution if we think the optimal value sits somewhere near a specific number. The main advantage of the random search is that we can the number of iterations we want the algorithm to complete, meaning we can adjust the time is takes to validate our model.

Chapter 3

Practical Implementation

For the purpose of this paper, we have engineered two different datasets, which we will use to make our intrahour and intraday volatility predictions. In this chapter, we will describe the processes that were used to create each of our dataset and our motivation behind choosing the variables in the datasets. We will explore certain correlations within our datasets, in particular, we will analyse and quantify the temporal dependencies between the feature variables and the response variable. Before we move to the data engineering process, we explore a handful of variables that we believe could be used to forecast the directional movement of intraday volatility. Clearly, as *directional movement* of intraday volatility is based off the *absolute* value of intraday volatility, we can choose variables that will predict the latter.

3.1 Choice of Feature Variables

Through reading previous literature and from our own intuition of a stock's volatility we have chosen 6 feature variables that we believe will be able to forecast our response variable. This initial choice of feature variables is shown in Table 3.1 and we explain our motivation for choosing each feature.

The choice of *Returns* and *Returns Squared* was motivated by the paper "Volatility is (Mostly) Path Dependent" by Julien Guyon and Jordan Lekeufack [16], who discover the correlation between a stock's future volatility and previous returns. They develop a model which predicts the absolute value of daily realised volatility, $\sigma(t+1)$ on day, $t+1$, for a variety of stock indexes. Their best performing model achieves an R^2 value of 0.654. The linear model which achieves this is described below:

$$\sigma(t+1) = \beta_0 + \beta_1 R_{1,t} + \beta_2 \sum_t, \tag{3.1}$$

where,

$$R_{1,t} = \sum_{t_i \leq t} K_1(t - t_i) r_{t_i} \tag{3.2}$$

and

$$\sum_t = \sqrt{\sum_{t_i \leq t} K_2(t - t_i) r_{t_i}^2}, \tag{3.3}$$

with r_{t_i} being the daily return of the asset on day t_i.

The $R_{1,t}$ is in place to capture a phenomenon called the *leverage effect*, which is when volatility tends to rise as asset prices fall. This is then weighted by a convolution kernel K_1, which puts more weight on recent returns and less weight on returns further in the past. $\sum_t$ captures recent movement in the asset price, regardless of the direction. This is based on the idea that large periods of volatility are likely to be followed by more large periods of volatility. We therefore decided to use daily log returns and daily log returns squared as feature variables.

Feature Variable
Log Daily Returns
Log Daily Returns Squared
Daily Trading Range
Daily Trading Volume
30-Day Moving Average
Intraday realised Volatility

Table 3.1: Potential feature variables to forecast intraday realized volatility.

Daily Trading Volume is a measure of market liquidity. Periods when large volumes are consistently being traded can lead to more liquid markets as more market participants will offer the product. This can lead to the asset being traded more easily without significant price movements, hence lower volatility. Additionally, when there are sudden increases in trading volumes to reflect market participants rapidly buying or selling positions, this can cause large price fluctuations and increases in volatility.

Daily Trading Range is a strong indicator of market sentiment and how participants are reacting to market news events. For example, if investors reacts positively to Apple's earnings being greater than expected, this can lead to significant buying, quickly driving up the price of the stock. This would lead to a large trading range and increase in volatility. Therefore the two are related.

30-Day Moving Average is able to identify volatility clusters. We know that periods of large volatility tend to be followed by further periods of large volatility. This average is able to smooth out daily price movements and capture the underlying volatility trend.

Intraday Volatility is our final and most obvious choice of feature variable. We learn from Guyon, Lekeufack and in [38] that the best indicator of future realised volatility values are previous realised volatility values.

3.2 Data Engineering

In order to obtain a dataset which included all of these variables, we were required to undergo a data engineering process. It was not possible to obtain the data for each variable directly, however by re-engineering raw price data, we were able to produce the full dataset. As we explained in the previous chapter, the **main** objective in the project is forecasting **intraday** volatility. Therefore the dataset we build for this task will include all 6 of the possible feature variables. On the other hand, as forecasting **intrahour** volatility is our secondary objective (which we do for the purpose of model comparison), then the dataset we build for this task will only contain 2 of the feature variables.

3.2.1 Dataset Construction

In this project we used two data sources, Yahoo Finance [1] and First Rate data [2] to acquire our raw data and it can be accessed through Dropbox [3].

Date	Return Squared	Intrahour Volatility	Response
02/02/2005 10:00	6.47E-05	0.0107	0
02/02/2005 11:00	1.87E-05	0.0057	0
...	...	...	...
19/01/2024 18:00	3.93E-07	0.00122	1

Table 3.6: **Intrahour Volatility Dataset** (Final Data)

Date	Return Squared	Return	Range	Volume	EMAF	Intraday Volatility	Response
02/02/2005	0.000714	0.0267	0.0397	1024582888	1.26	0.00737	0
03/02/2005	0.000535	-0.0231	0.0375	732986296	1.27	0.00431	0
...	...	...	...	...	...	...	...
18/01/2024	0.00103	0.0321	3.31	75770704	187	0.00573	1

Table 3.7: **Intraday Volatility Dataset** (Final Data)

From First Rate Data, we obtained the *closing price* data and *volume traded* data of the "AAPL" stock for every minute, every hour and every day from 02/02/2005 10:00 to 19/01/2024 19:00, which came in 3 separate datasets; **AAPL Minutely**, **AAPL Hourly**, **AAPL Daily**. From Yahoo Finance, we obtained the *30-day Moving Average (EMAF)* data for every day from 02/02/2005 to 19/01/2024, which came in the dataset **AAPL Daily EMAF**. These raw datasets are shown in Tables 3.2, 3.3, 3.4 and 3.5.

Date	*closing price*	Volume
02/02/2005 10:00:00	1.14	6815648
02/02/2005 10:01:00	1.14	4319840
...	...	...
19/01/2024 19:00:00	191.3	575

Table 3.2: **AAPL Minutely** (Raw Data)

Date	*closing price*	Volume
02/02/2005 10:00	1.14	18315612
02/02/2005 11:00	1.13	180934992
...	...	...
19/01/2024 19:00	191.3	15771

Table 3.3: **AAPL Hourly** (Raw Data)

Date	*closing price*	Volume
02/02/2005	1.42	1024582888
03/02/2005	1.39	732986296
...	...	...
19/01/2024	191.3	65993030

Table 3.4: **AAPL Daily** (Raw Data)

Date	EMAF
02/02/2005	1.26
03/02/2005	1.27
...	...
19/01/2024	187

Table 3.5: **AAPL Daily EMAF** (Raw Data)

This data was engineered and manipulated in Python to produce the final datasets shown in Table 3.6 and Table 3.7. The code used to produce these datasets can be found in [25]. These are the datasets we use to train all of the neural network models.

The process of constructing our **Intraday Volatility Dataset** from our raw data went as follows:

1. The **Response** variable was obtained by comparing the *intraday realised volatility* value, $\sigma(t)$ for a given row, t, and the *intraday realised volatility* value, $\sigma(t+1)$ for the following row, $t+1$. The row was assigned a 0 or 1 according to:

$$\textbf{Response} = \begin{cases} 1 & \text{if} \quad \sigma(t+1) \geq \sigma(t) \\ 0 & \text{if} \quad \sigma(t+1) < \sigma(t). \end{cases}$$

2. The *intraday realised volatility* data was obtained using the hourly *log return* data and Equation (1.1). The hourly *log return* data was obtained using the hourly *closing price*, S_i from the **AAPL Hourly** raw dataset and Equation (3.4).

$$\text{Log Daily Returns} = \log(\frac{S_i}{S_{i-1}}). \tag{3.4}$$

3. The daily **Return Squared** feature variable came from squaring the daily *log return* data. The daily *log return* data was obtained in the same manner as the hourly *log return* data, (i.e. using Equation (3.4) but with the daily *closing price* raw data, from the **AAPL Daily** raw dataset).

4. The **Volume** feature variable was obtained directly from the **AAPL Daily** raw dataset.

5. The **Range** feature variable was obtained by finding the difference between the maximum and minimum minutely *closing price* for each day.

6. The **EMAF** feature variable was obtained directly from the **AAPL Daily EMAF** raw dataset.

7. The 6 columns were merged and aligned according to the **Date**, removing any rows where data was missing.

The **Intrahour Volatility Dataset** was obtained in a similar manner to the intraday dataset, with the following differences:

- The *intrahour realised volatility* data was obtained using the minutely *log return* data and Equation (1.2). This data was then used to produce our **Response** variable.
- the hourly **Return** and **Return Squared** feature variables were obtained using the hourly *closing price* data, as opposed to the daily *closing price* data.
- the hourly **Volume** was taken from the **AAPL Hourly** raw dataset rather than the **AAPL Minutely**.

For both of our datasets, we will use the years 2005 to 2020 as our training set and the years 2021 to 2024 as our test set.

3.2.2 Removing Outliers

Before training our models, it is important that we address any outliers in our training data, which may cause the model to over fit to these data points, particularly if they deviate a significant amount from the rest of the data. These outliers may have been caused by one off economic events, which doesn't represent how the data interacts *normally*. For example, if we look at Figure 3.1, we can see that during 2008, during the time of the Great Financial Crisis, there were sharp spikes in volatility. As this is a one off event caused by factors which aren't capture by our feature variables, we would not want our models to learn these patterns.

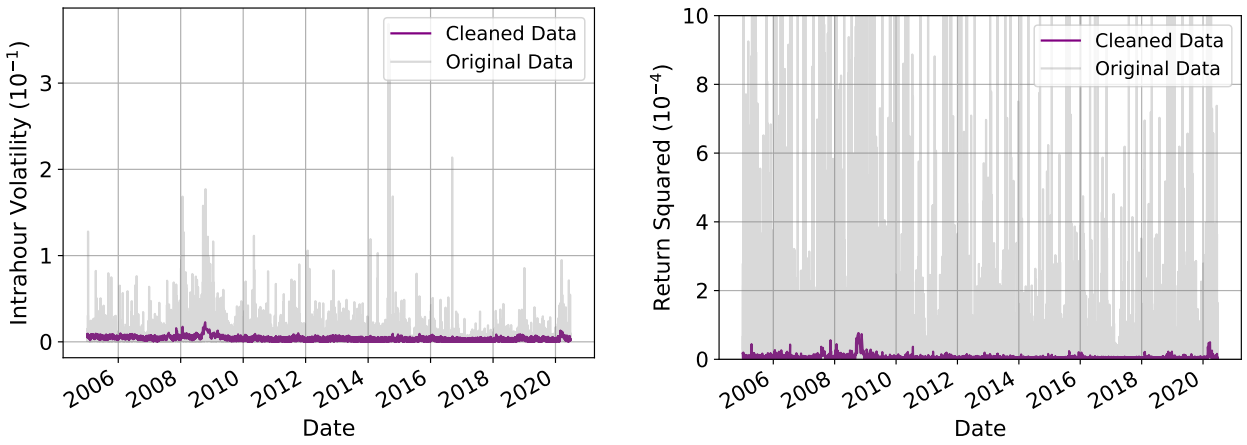
[9] explains how an outlier, $\mathbf{x_t}$ in time series data is identified if $\|\mathbf{x_t} - E(\mathbf{x_t})\| > \tau$ for some chosen, τ. In this paper, we will define an outlier as any observation containing a feature whose value is greater than 5 *mean absolute deviations, (MAD)*, from the rolling 40 hour median value. We choose the median and MAD as these values are less sensitive to outliers (i.e. if there were several outliers within the last 10 hours, this would skew the mean and standard deviation, but now the median and MAD). We define this outlier rule as:

$$\mathbf{x_t} \text{ is Outlier} = \begin{cases} \text{True} & \text{if} \quad |x_{tj} - \text{median}(x_{(t-20,..,t+20)j})| > 5 \text{ x MAD}, \quad j = 1, 2 \\ \text{False} & \text{Otherwise} \end{cases},$$

where,

$$\text{MAD} = \text{median}(|x_{tj} - \text{median}(x_{(t-20,..,t+20)j})|), \quad j = 1, 2.$$

We should be aware that removing data points will lead to gaps in time within our time series dataset. This will lead to our recurrent neural networks finding patterns between days/hours which don't consecutively follow each other. However, this approach is more favourable than leaving outliers in the dataset, which could lead to over-fitting. Figure 3.1 shows the first 80% of the **Intrahour Volatility Dataset** before and after removing outliers.

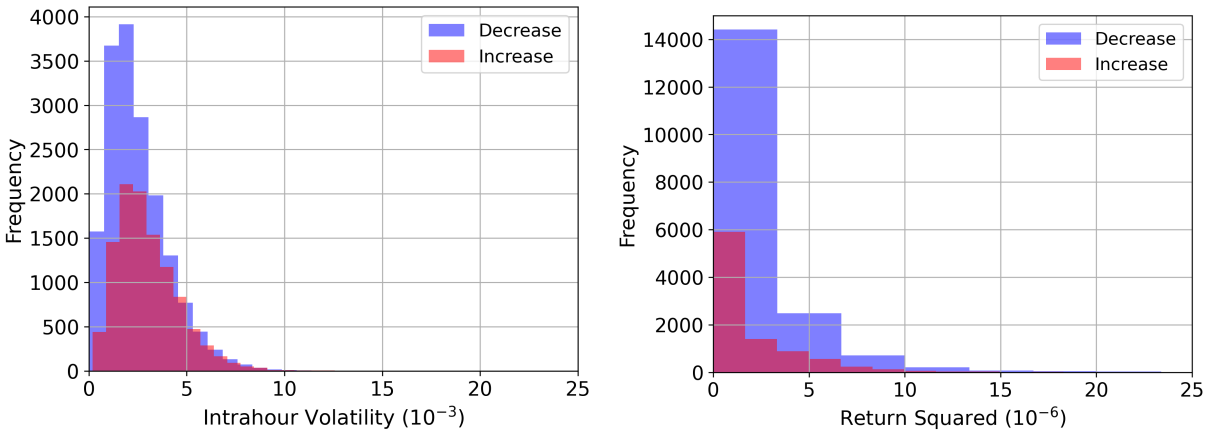


(a) Intrahour Volatility Feature

(b) Return Squared Feature

Figure 3.1: Comparison of the Original Feature set and the Cleaned Feature set in the **Intrahour Volatility Datatset**. The cleaned data will be used to train our models.

The histograms in Figure 3.2 help us to visualise the distribution of data in the cleaned **Intrahour Volatility Datatset**. Both classes follow similar distributions with respect to the feature variables, causing the classes to overlap significantly. Therefore, to train a model to separate this data, it will be required to learn non-linear patterns within the data. Neural network models do not make any assumptions about the distributions of the underlying data.



(a) Intrahour Volatility Feature

(b) Return Squared Feature

Figure 3.2: Distribution of increases (1) and decreases (0) for each feature variable.

Min-Max Scaling

As opposed to training our models on the absolute values of our feature variables, we train our models on the standardised features, meaning each feature is transformed to have a range between 0 and 1. This step ensures that each feature contributes to the training of the model equally, regardless of the absolute value of the data. We standardise the data by finding the minimum value, $x_j^{\min}$ and maximum value, $x_j^{\max}$ of each feature $j = 1, ..., p$ in the training set. We then scale each data point's features, $x_{t,j}$ as follows:

$$x_{t,j}^{\text{scaled}} = \frac{x_{t,j} - x_j^{\min}}{x_j^{\max} - x_j^{\min}}, \qquad \forall t = 1, ..., n.$$

Clearly, our test data should also be scaled in the same manner. However, it is important that we do not scale the test set with the maximum and minimum feature values from the test set. This is because we must assume that all of this data is unseen, and therefore we would not know its maximum or minimum values. Instead, we scale the test data using the maximum and minimum values from the training data.

3.3 Autocorrelation and Partial Autocorrelation

The two datasets we have engineered can be described as a multivariate time series. A time series is a set of observations $\mathbf{X} = (\mathbf{x_1}, ..., \mathbf{x_n})^T \in \mathbb{R}^{n \times p}$, where each observation is taken at time, t [10]. For example, in our **Intrahour Volatility Dataset**, this contains consecutive observations of the stock's *intrahour volatility* and *return squared* at every hour. It would therefore be defined as a multivariate time series.

There are many features of time series data, and for the purpose of our project the feature we are interested in is the correlations between observations at different times in our dataset. The reason for this is because if these correlations exist, we may be able to train models which exploit these correlations to make predictions on future values of the data. For example, if we learnt that there are correlations between a stock's current volatility and the volatility over the past 10 hours, then we would want our models to take advantage of this. Within a multivariate time series, there are two types of correlation that we are interested in. We may be interested in the correlation between one variable's current value and the same variable's previous values, which is called autocorrelation. Equally, we may be interested in one variable's current value, and a different variable's previous values, this is called cross correlation.

Autocorrelation

Within the data series $X = (x_1, x_2, ..., x_n)^T \in \mathbb{R}^n$, the values x_{t-k} and x_t , for $t = k+1, ..., n$ are separated by k units of time, which we call the time lag. The autocorrelation determines the correlation between the observations at a time lag, k. The autocorrelation, r_k of a data series at time lag, k is defined as:

$$r_k = \frac{\sum_{t=1}^{n-k}(x_t - \bar{x})(x_{t+k} - \bar{x})}{\sum_{t=1}^{n}(x_t - \bar{x})^2}. \tag{3.6}$$

When we have obtained our autocorrelation values, we would also like to know the relative significance of these correlations, to decide whether it is worth training a time series model. The significance of the autocorrelation coefficient, r_k at lag k, can be determined through the Ljung-Box Q statistic [**?**] in the following hypothesis test:

H_0 : Autocorrelation does not exist within the time series data at time lag, k

H_1 : Autocorrelation exists within the time series data at time lag, k

Under H_0, the Ljung-Box Q Statistic assumes the following distribution:

$$\mathrm{Q} = n(n+2)\sum_{i=1}^{k}\frac{r_i^2}{n-i} \sim \chi_k^2 \tag{3.7}$$

We reject H_0 at the 5% significance level if $P(\chi_k^2 > Q) < 0.05$.

Cross Correlation

We may be interested in how future values of one variable, Y are related to previous values of another variable, X. This is where we introduce cross-correlation. Cross-correlation is similar to autocorrelation, however instead it refers to the correlation between two different, data series, Y and X at some time lag k. For the two data series $Y = (y_1, y_2, ..., y_n)^T \in \mathbb{R}^n$ and $X = (x_1, x_2, ..., x_n)^T \in \mathbb{R}^n$, we define the cross correlation of Y over X at time lag, $k > 0$ to be:

$$c_{yx,k} = \frac{\sum_{t=1}^{n-k}(x_t - \bar{x})(y_{t+k} - \bar{y})}{\sqrt{\sum_{t=1}^{n}(x_t - \bar{x})^2}\sqrt{\sum_{t=1}^{n}(y_t - \bar{y})^2}}. \tag{3.8}$$

With cross correlation, we need to be careful which variable is leading and which is lagging. In the definition above we say that X *lags* Y and this is the correlation between past values of X and future values of Y. The standard correlation of X and Y is equivalent to the cross correlation at time lag, $k = 0$. We can also test the statistical significance of our cross-correlation values, $c_{yx,k}$ at time lag, k through the following hypothesis test [14]:

H_0 : Cross-correlation does not exist within the time series data at time lag, k

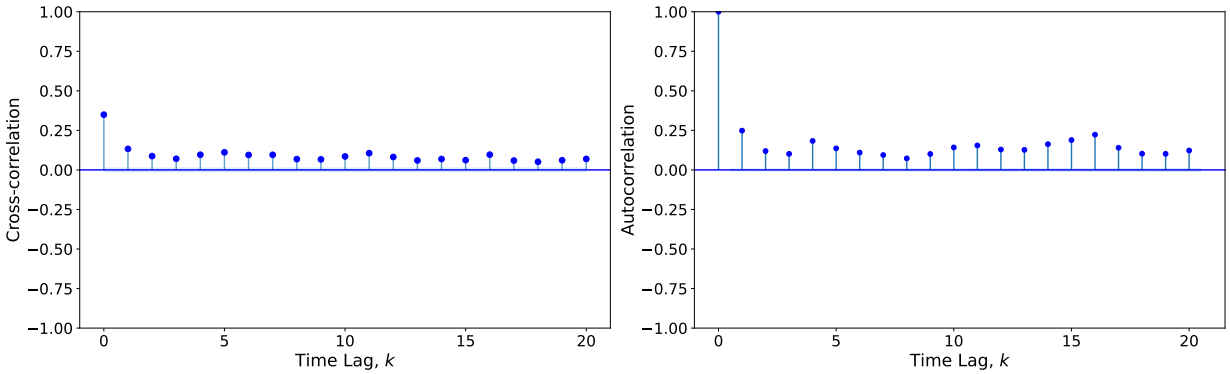
H_1 : Cross-correlation exists within the time series data at time lag, k

Under H_0, the $T_{yx,k}$ Statistic assumes the following distribution:

$$T_{yx,k} = \sqrt{n}c_{yx,k} \sim t_k \tag{3.9}$$

We reject H_0 at the 5% significance level if $P(t_k > T_{yx,k}) < 0.05$.

Example 3.3.1. Figure 3.3a is a cross correlation function (CCF) plot, which shows the cross correlation between *Intrahour Volatility* at time, $t + k$ and *Return Squared* at time, t, for a time lag $k = 1, ..., 20$. Figure 3.3b is an autocorrelation function plot (ACF) which shows the autocorrelation within the *Intrahour Volatility* data for a time lag $k = 1, ..., 20$. On both plots, the shaded blue region around the x-axis represents the confidence interval for cross correlation, c_k or autocorrelation, r_k at each time lag , k based off the hypothesis tests described above.



(a) **Intrahour Volatility over Return Squared**

(b) **Intrahour Volatility**

Figure 3.3: These plots help us to visualise the time series correlations within the **Intrahour Volatility Dataset**. At each time lag, k, the data points are plotted in blue if the correlation is statistically significant and in red if the correlation is not statistically significant.

We can see that at every time lag there exists significant cross-correlation, which suggests previous values of the return squared are a good indicator of future intrahour volatility. Additionally, there exists significant autocorrelation at each time lag, which suggests future intrahour volatility values are dependent on previous intrahour volatility values. We say that temporal dependencies exist within this dataset. As we will learn later on in the project, this property will make our data suitable for training recurrent neural networks.

Chapter 4

Feedforward Neural Networks and Optimizers

Throughout this chapter we will introduce the concept of an artificial neural network and explain how they learn from training data. We will introduce the feedforward neural network, the simplest type of neural network, in order to lay a foundation to build on in later chapters. Following this we will compare different optimizer algorithms, which are used to update the parameters in a model. We will explore the Adam optimizer in depth.

4.1 Feedforward Neural Networks

A feedforward neural network (FFNN) is one of the simplest forms of an artificial neural network and it is characterized by containing layers of neurons where information flows through the model in one direction. Each network has an *input layer* to receive the input $\mathbf{x_t}$, an *output layer* to present the model's prediction and at least one *hidden layer* to compute the prediction. The far left layer is the input layer, which contains p units, representing the number of feature variables in our dataset. Moving across the next layers are the hidden layers. These transform the data through a weighted linear summation before being acted on by an activation function, introducing non-linearity into the model. Throughout this project, we will only consider neural networks with a single hidden layer. The far right layer is the output layer. This receives the transformed values, which is used to produce the final prediction. For all of our models, the output layer will contain 1 unit, representing the probability that an observation lies in the increase class (1).

In this project, we will not explore different choices of activation functions. Instead, we will state which activation functions are used in each model, with a short description explaining its use. Further reading on choices of activation functions can be found in [33], which describes 10 of the most commonly used activation functions in neural networks. We will define a FFNN model, h with 5 neurons in the input layer, a single hidden layer with 80 neurons and an output layer with a single neuron [22]:

$$h(\mathbf{x_t}) = \sigma_2\left(\left(\sum_{j=1}^{80} w_{(2),1,j}\right) \cdot \sigma_1\left((\sum_{i=1}^{5} w_{(1),j,i}x_{ti}) + b_{1,j}\right) + b_2\right) \tag{4.1}$$

$$= \sigma_2\left(\mathbf{w}_{(2,1)} \cdot \sigma_1\begin{pmatrix}\mathbf{w}_{(1,1)}\mathbf{x_t} + b_{1,1}\\ \dots \\ \mathbf{w}_{(1,80)}\mathbf{x_t} + b_{1,80}\end{pmatrix} + b_2\right), \tag{4.2}$$

where,

$$\hat{y_t} = \begin{cases} 0 & \text{if } h(\mathbf{x_t}) < threshold \\ 1 & \text{if } h(\mathbf{x_t}) \geq threshold \end{cases}. \tag{4.3}$$

The function, σ_2 is a softmax activation function, which maps the final summation to a value, $h(\mathbf{x_t})$ between 0 and 1, representing the probability that observation, $(\mathbf{x_t})$ lies in the increase (1) category. The function σ_1 is a ReLU activation function, used to introduce non linearity into the model.

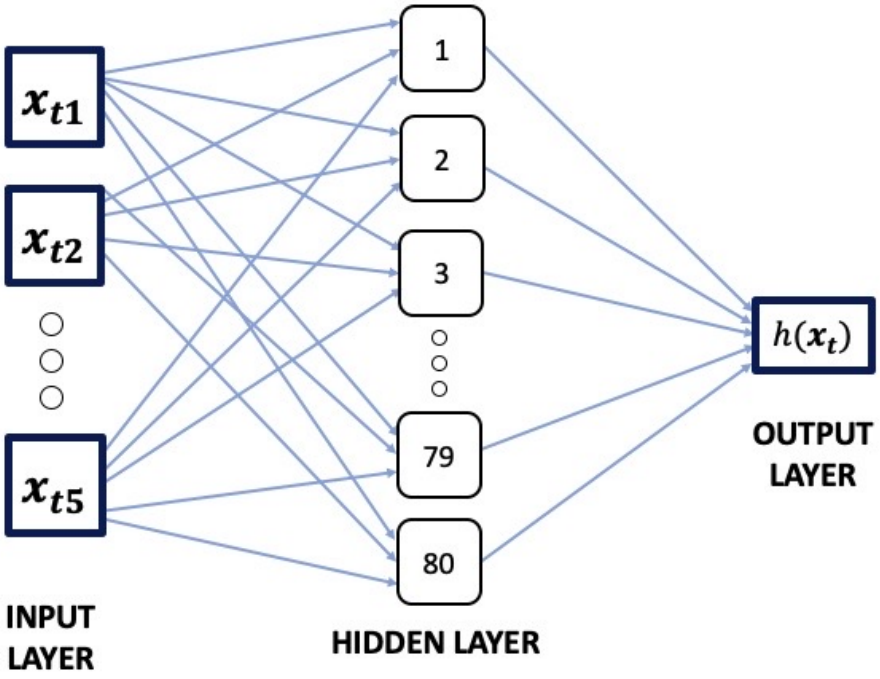


Figure 4.1: Architecture of the single layered feedforward neural network defined by (4.1) and (4.3).

We can see from (4.1), our model contains a set of parameters $\mathbf{W} = (\mathbf{w}_{(2,1)}, b_2, \mathbf{w}_{(1,j)}, b_{1,j})$ for $j = 1, ..., 80$, which must be learnt. This is done during a process called backpropagation, which we will describe shortly. Before doing so, we must explore the idea of a loss function.

4.1.1 Loss Functions

A loss function is a function that measures the difference between the value, $\hat{y}_t$ predicted by the model for an observation $\mathbf{x_t}$, and the true response value, y_t, [29]. Therefore, a loss function can measure the performance of our model, h. Clearly, depending on the nature of the problem, we must choose a loss function that can capture this information, depending on the nature of our response variable. As we are using our neural network models for binary classification, then we must use a loss function that has a low value when the model correctly classifies our observations and a high value when the model incorrectly classifies. One type of loss function we could use is called a **zero-one** loss function, which takes the form:

$$\mathcal{L}(\hat{y}_t, y_t) = \begin{cases} 0 & \text{if} \quad \hat{y}_t = y_t \\ 1 & \text{if} \quad \hat{y}_t \neq y_t \end{cases}. \tag{4.4}$$

This is a good starting point, however the downside to this loss function is that it doesn't incorporate how close the model was to making the correct prediction, it is only concerned with whether the model was right or wrong. This is where we introduce the idea of a **binary cross-entropy (BCE)** loss function, which is described as follows:

$$\mathcal{L}(h(\mathbf{x_t}), y_t) = -(y_t \log(h(\mathbf{x_t})) + (1 - y_t) \log(1 - h(\mathbf{x_t}))). \tag{4.5}$$

The BCE loss function takes into account how right or wrong a prediction was, leading to a better understanding of our model performance. For example, assume we have an observation, $\mathbf{x_t}$ with a true response value of $y_t = 1$. If the model predicted this observation to have $\hat{y}_t = 0$, then depending on the value of $h(\mathbf{x_t})$, the model may have been close to making the correct prediction, or it may have been significantly wrong. If the model predicted $h(\mathbf{x_t}) = 0.1$, which is a very incorrect prediction, then the loss function would have a value of 2.30. If the model predicted $h(\mathbf{x_t}) = 0.49$, which is a much better prediction, then the loss function would have a value of 0.71. The BCE loss is a convex function in $h(\mathbf{x_t})$, with a minimum at $h(\mathbf{x_t}) = y$, which is necessary for ensuring we converge to a true optimal solution. Throughout this paper, we will use the BCE loss function when we are training our neural network models. Clearly, we will not just be interested in the loss function for a single observation, but instead the average loss over a sample of data, or even the whole dataset. Therefore we introduce the idea of an empirical risk function.

Definition 4.1.1. The **Empirical Risk Function (ERF)** is the average of the loss functions of k observations in our dataset, it is defined as:

$$\hat{R}_k = \frac{1}{k} \sum_{j=1}^{k} \mathcal{L}(h(\mathbf{x_j}), y_j), \tag{4.6}$$

where the observations are randomly chosen with or without replacement from the whole dataset.

When $k = n$, our empirical risk function is the average loss over the whole dataset. For $k = 1$, we have the individual loss function for the observation $\mathbf{x_t}$. As we will learn shortly, when we're training our models we will be required to find the derivative of the empirical risk function with respect to the model's parameters. The number of observations, k chosen is called our batch size.

Definition 4.1.2. Batch - A subset of our training dataset containing k observations, where the number of observations is called the *batch size*.

Definition 4.1.3. Epoch - One epoch is a complete passage of our whole training dataset through our neural network model during backpropagation.

4.1.2 Training Neural Networks

As stated earlier, in order to train an FFNN, it involves an algorithm called backpropagation. This algorithm iteratively adjusts the parameters in the neural network in a way that reduces the value of the loss functions for our training data predictions. By doing this, it trains the model to make accurate predictions on our training data. (With the correct hyperparameter tuning, the hope is that the model will then generalise to make accurate predictions on the test data). The general process of backpropagation goes as follows, which is repeated iteratively until a desired number of epochs it reached:

1. The neural network's weights and biases, $\mathbf{W}$ are randomly initialized
2. Data is passed through the model to make initial predictions, $\hat{\mathbf{y}}_{\mathbf{train}}$ (this stage is called forward pass)
3. The empirical risk function is calculated from our loss functions, $\mathcal{L}$
4. The gradient of the empirical risk function is computed with respect to the parameters (this stage is called backward pass)
5. The parameters are updated in the opposite direction of the gradient using an optimizer.

For the remainder of this chapter, we will not explore all the details of this algorithm, we will focus mainly on stage 5. In the following chapter, we will explore how the gradients are calculated, in the context of a recurrent neural network as opposed to a FFNN. Therefore to avoid repetition in the project, specific details of the backward pass process for an FFNN can be found in [20]. Depending on which optimizer we use determines the way in which data is passed through the model and therefore how much data is included in the empirical risk function. Now that we understand the general idea of how a neural network is trained we will explore how the parameters are updated with the use of an optimizer.

4.2 Optimizers

There are several types of optimizer algorithms which one could use, all which have a variety of advantages and disadvantages. An optimizer is used to update the parameters in the neural network in such a way that it reduces our ERF. Optimizers help the parameters to converge to values which minimise the loss function (ERF). Once the optimizer has minimized the empirical risk function for our data, we could describe the neural network as optimal for the training data. In this paper, we will explore 3 types of optimizer algorithms, Mini-Batch Gradient Descent, Adagrad and Adam. For the first 2, we will describe how the algorithms work to gain an intuitive understanding of an optimizer algorithm. We will use these 2 optimizes for a comparison with the Adam optimizer. We will explore the Adam optimizer in more depth and formally describe the iterative process.

Mini-Batch Gradient Descent

The mini-batch gradient descent algorithm randomly creates batches of size k, with data taken with or without replacement from the whole dataset [23]. For each batch, stages 1 to 4 of the backpropagation process The algorithm then updates the parameters after each batch is processed. This process then repeats until the desired number of epochs has been reached.

Definition 4.2.1. Iteration, t - An update of the model's parameters, which occurs after each batch is processed.

The mini-batch gradient descent algorithm, with learning rate, η and batch size, k can be described by the following equations:

$$\mathbf{W_{t+1}} = \mathbf{W_t} - \eta \cdot g_t, \tag{4.7}$$

where g_t is computed after each iteration, t according to:

$$g_t = \nabla_{\mathbf{W_t}} \hat{R}_k^{(t)} = \frac{1}{k} \sum_{j=1}^{k} \nabla_{\mathbf{W_t}} \mathcal{L}(h(\mathbf{x_j}), y_j). \tag{4.8}$$

This process is repeated until we reach a desired number of iterations, N, yielding the final parameter values, $\mathbf{W_N}$.

Note. We may be wondering how we can determine the optimal number of iterations and batch size when training our models. As discussed previously, these values are both hyperparameters which we will attempt to optimise during the validation stage. The aim is to find values which allow the model to fit well to the training data, but will also perform well on unseen data.

This optimizer finds the balance between standard gradient descent (GD) and stochastic gradient descent (SGD). In GD, an iteration occurs after the whole dataset has been processed, therefore the gradient of the ERF is calculated using the whole dataset. This can be problem for large datasets and leads to slow convergence of the parameters. In SGD, an iteration occurs after each data point has been processed, and the updates are based on the gradient of a single loss function. This can lead to faster convergence, but can also cause a lot of variation in the parameter updates. Mini-batch gradient

descent finds the balance between both of these algorithms, and the batch size is a hyperparameter which can be tuned to fit our dataset.

The main limitation to this algorithm is the fixed learning rate, η. After every iteration, the size of the step the parameters take is the same each time. For a too larger learning rate as we get closer to the minimum, this could lead to the parameter values overshooting the minimum, meaning the parameters won't converge. For a too smaller learning rate, our parameter values may converge too slowly meaning they won't reach the optimal values in a reasonable number of epochs. With this in mind, we will introduce our next optimizer.

Adaptive Gradient Algorithm (Adagrad)

The Adagrad optimizer is a more sophisticated algorithm. It still uses batches in the same way as mini-batch gradient descent, but it is designed to have a variable learning rate [32]. The Adagrad algorithm, with learning rate, η can be described by the following equations:

$$\mathbf{W_{t+1}} = \mathbf{W_t} - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t \tag{4.9}$$

where,

$$G_t = G_{t-1} + (g_t)^2. \tag{4.10}$$

G_t is a diagonal matrix, where each diagonal element is the sum of squares of the gradients up to iteration, t. G_t is initialized as a matrix of zeros, and after each iteration, t, the values of the matrix increases. The ϵ term is in place to avoid the division by zero in the first iteration. We can see that this algorithm facilitates a variable learning rate. As the algorithm completes more iterations and we get closer to the minimum of the ERF, the step size the parameters take will decrease. This will result in a more stable convergence to the optimal parameters.

The limitation to this algorithm is due to the cumulative growing nature of the G_t matrix, this can cause the variable learning rate shrink too quickly, therefore preventing the parameters from converging. This shows that ideally we would like a learning rate that can adaptively change. This is where we introduce our final optimizer, namely, the Adam optimizer.

Adaptive Moment Estimation (Adam)

The Adam optimizer was first introduced by [24]. This paper describes the Adam algorithm and how it can be used to minimize some general cost function, $f(\theta)$. Corollary 4.2 in the paper proves that if the algorithm completes enough iterations then the parameters will converge to their optimal values. We will formally describe the Adam algorithm, assuming that the cost function we are minimising is the ERF with batch size, k, where the data is randomly chosen from our full data set with or without replacement. We will also assume that we have a set number of iterations, N that the algorithm will complete.

Algorithm 1. Adam Algorithm with learning rate, η

1. **Set** t=0
2. **Initialize** hyperparameters, $\beta_1, \beta_2 \in [0, 1)$
3. **Initialize** moment vectors $m_0 = 0$, $v_0 = 0$ and initialize parameter vector, W_0
4. *While* $t \leq N$:
   - **Obtain** $\hat{R}_k^{(t)}$
   - $t \leftarrow t + 1$
   - $g_t \leftarrow \nabla_{\mathbf{W_t}} \hat{R}_k^{(t)}$ (Compute gradient of ERF)
   - $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ (Update biased decaying average of past gradients)
   - $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ (Update biased decaying average of past gradients squared)
   - $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ (Compute biased corrected average of past gradients)
   - $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ (Compute biased corrected average of past gradients squared)
   - $\mathbf{W_t} \leftarrow \mathbf{W_{t-1}} - \eta \frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}$ (Update parameters)
5. **Return** $\mathbf{W_N}$ (resulting parameters)

As we've learnt, the Adagrad algorithm reduces its learning rate after each iteration in a way that can cause convergence to slow down too quickly. The Adam optimizer overcomes this by scaling the direction of the average moving gradient, v_t, with its magnitude m_t, through the term $\frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}$. This means that Adam maintains a learning rate which is adjusted based on the previous value of the gradient, preventing the problem. We describe Adam as having an *adaptive learning rate*.

In addition to this, rather than the parameter updates being made purely on the gradient, which is the case in mini-batch gradient descent, Adam incorporates the idea of *momentum*. Through the terms m_t and v_t, the algorithm stores the moving average of the gradients and gradients squared, which helps to stabilise the convergence across each iteration. This occurs because, if we consider a batch containing a lot of noisy data, this would result in a fluctuation in the gradient. Therefore, by updating the parameters based off both this gradient and the average of previous gradients, it prevents the parameters updating in the wrong direction and smooths out the update process.

The final aspect of the Adam optimizer is that it includes bias correction terms. As the terms m_t and v_t are initialized with values of zero at $t = 0$, this leads to the estimate of these terms being slightly biased towards zero. [24] shows this for the v_t term, We will show this for the m_t term. We know that the decaying average of past gradients, m_t at iteration, t is formulated as:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ &= (1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} \cdot g_i^2, \end{aligned}$$

where the second equality comes from writing the equation as a function of gradients from the previous iterations. The expected value of the decaying average of past gradients is given by:

$$E[m_t] = E\left[(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\cdot g_i^2\right]$$
$$= E[g_t]\cdot(1-\beta_1^t)+\xi.$$

Assuming that the average gradient does not significantly change over time, then $\xi = 0$. Therefore, we can see how the expected value of the gradient is biased by the $(1-\beta_1^t)$ term. The Adam optimizer however corrects for this bias by introducing the $\hat{m}_t$ and $\hat{v}_t$ terms. To conclude, due to the Adam's adaptive learning rate, ability to incorporate momentum and bias correction, we would expect it to perform better than both the Adagrad and Mini-batch gradient descent algorithms, in terms of quicker convergence to an optimal solution.

Example 4.2.1. We evaluated the mini-batch gradient descent, Adagrad and Adam optimizers by training a single-layered feedforward neural network, with the architecture shown in Figure 4.1, on our **Intraday Volatility Dataset**. Clearly, the results in Figure 4.2 show that the Adam optimizer reduces the loss function (ERF) most effectively, leading to a quicker convergence of optimal parameter values. Therefore, moving forward with the project, we will use the Adam optimizer to train all of our neural network models.

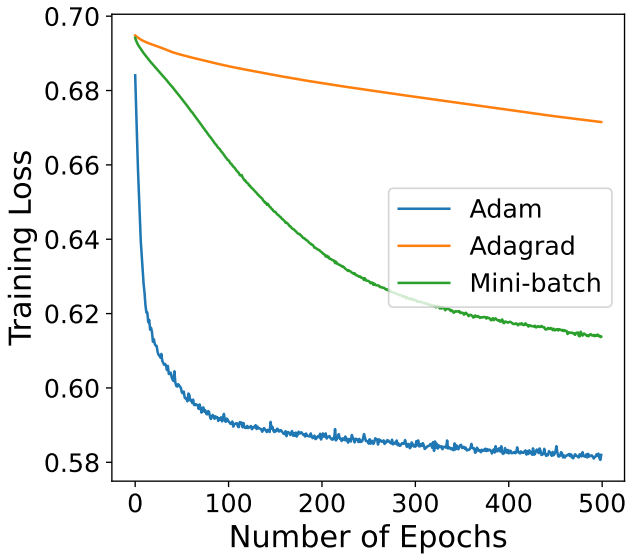


Figure 4.2: This figure compares the reduction in training loss over 500 epochs for the three different optimizer algorithms.

Chapter 5

Recurrent Neural Networks

The recurrent neural network (RNN) takes a different approach to making a prediction when compared with FFNNs. Throughout this chapter, we will explore different variations of the recurrent neural network (RNN) model, including the basic RNN and long short-term memory networks (LSTMs). Throughout this chapter and the next, we will train these models on the **Intrahour Volatility Dataset** to forecast the directional movement of the stock's intrahour realised volatility. Additionally, we will explore temporal dependencies within our dataset and understand how this impacts an RNN's predictive performance.

5.1 Basic Recurrent Neural Network Model

In the previous chapter, we learnt that a trained FFNN model can be thought of as an injective mapping function from $\mathbf{X}$ to $\mathbf{y}$. The model uses values of the feature variables from a single observation, $\mathbf{x_t}$ and predicts the response variable, $\hat{y}_t$ for the observation. In the FFNN, each data input is processed independently by the network during forward and backward pass, meaning the model is limited to *spatial* learning. This is learning the relationship across space (i.e. learning the relationship between $\mathbf{x_t}$ and y_t for individual observations in time).

The many-to-one recurrent neural network (RNN) takes a different approach. Instead, a sequence of T values from successive observations of the feature variables, $\mathbf{x^{(i)}} = (\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_T})^T$ are used to predict the value of a singular response variable, $\hat{y}_i$. As we will learn from the architecture of an RNN, it is designed to facilitate *spatial learning* and *temporal learning*. This means the network is able to learn patterns between T successive observations in the feature set (*temporal learning*). It can also learn patterns between the feature variables, $\mathbf{x^{(i)}}$ and response variable, y_i at fixed moments in time (*spatial learning*).

As the **Intrahour Volatility Dataset** is a multivariate time series, which can be separated into sequences of observations, we would hope that the RNN can find patterns in these sequence to predict future response values. The main objective in time series prediction is to forecast future values of a response variable using historical values of the variable, as well as historical values of additional exogenous variables, which are correlated to the response variable [18]. For example, as explained, our objective in this chapter is to predict the directional movement of a stock's intrahour realised volatility, which is our response variable. Therefore in order to forecast this, we will use previous values of realised intrahour volatility in addition to previous values of another related variable. We choose this variable to be the stock's return squared. The role of the RNN is to learn the relationships between the sequences of observations and the response variable. The advantage of using an RNN over traditional approaches, such as the ARCH model [17], is that the RNN is able to learn non-linear relationships within the time series data.

5.1.1 Structure of a Recurrent Neural Network

The reason an RNN is able to learn temporal relationships is due to architecture of the network, which forms feedback loops, allowing for a sequence of observations to be inputted into the model. Figure 5.1 displays the looped architecture of the RNN. It shows the equivalent 'rolled' and 'unrolled' architecture to demonstrate how consecutive data points are inputted into the model individually and then processed sequentially.

The reason why this architecture allows for an RNN to recognise temporal patterns within the feature set, is due to the hidden state. When the RNN makes a prediction for $\mathbf{x^{(i)}}$, as the T observations are sequentially entered into the model, all of this information is stored by the hidden state. This is why an RNN is described as having *memory*.

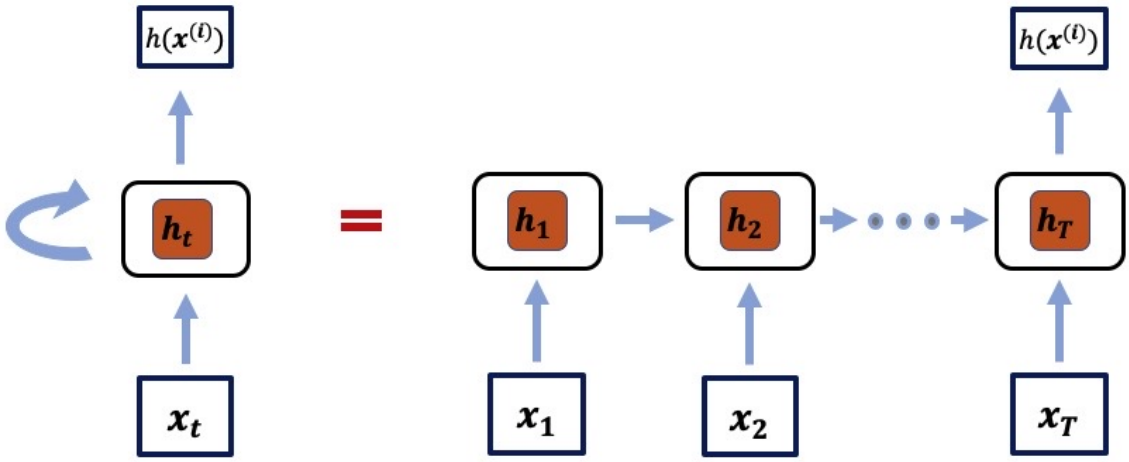


Figure 5.1: Architecture of a "Many to One" RNN with a single hidden layer. This figure shows the rolled (left) and unrolled (right) versions.

Data Reconstruction

Currently, our **Intrahour Volatility Dataset** has been constructed as a two dimensional array. In its current form, the first 80% of the dataset (which we will use as the training data) contains 57665 observations, containing 2 feature variables and 1 response variable. $\mathbf{X_{train}}$ is a (57556 x 2) array and $\mathbf{y_{train}}$ is a (57556 x 1) array. In order to fit our RNN models, we must reconstruct this data to have a three-dimensional form so that the model can process sequences of data, as opposed to individual observations. The new input will be three-dimensional with shape (x,y,z), where x is the number of observations, y is the sequence length and z is the number of features. In this chapter we will choose the sequence length of our neural networks to be 15. As we reconstruct our data, $\mathbf{X_{train}}$ will become a (57542 x 15 x 2) array and $\mathbf{y_{train}}$ will become a (57542 x 1) array.

Before proceeding, we will introduce new notation and definitions to describe our new data structure. For our RNNs we require each observation to be a sequence of successive data points, where $\mathbf{x^{(i)}}$ are the values of the feature variables and y_i is the value of the response variable for this observation. This data will take the following form:

$$\mathbf{x^{(i)}} := \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \vdots \\ \mathbf{x_T} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{T1} & x_{T2} \end{bmatrix}, \qquad y_i \in \mathbb{R}. \tag{5.1}$$

So far, we have used $t \in 1, ..., n$ to index an observation, $\mathbf{x_t}$ in the dataset $\mathbf{X} = (\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n})^T$. Moving forward, we use $i \in 1, ..., n - T$ to index an observation (sequence), $\mathbf{x^{(i)}}$ in our reconstructed dataset. We use $t \in 1, ..., T$ to index each data point, $\mathbf{x_t}$ in a single observation sequence $\mathbf{x^{(i)}}$.

Definition 5.1.1. Data Point, $\mathbf{x_t}$ - a single row of data in the observation sequence.

Definition 5.1.2. Sequence Length, **T** - the length of the sequence of successive data points for a single observation, $\mathbf{x^{(i)}}$ (i.e. number of data points, $\mathbf{x_t}$ in the observation, $\mathbf{x^{(i)}}$)

Tables 5.1 to 5.3 show the first 3 observations in our reconstructed dataset, which we will now refer to as the **Reconstructed Intrahour Volatility Dataset**. For example, in the first observation, the response of 1, represents an increase in intrahour realised volatility from the hour 03/02/2005 14:00, to the hour 03/02/2005 15:00. The reason the number of observations in our dataset has reduced from 57556 to 57542 is because for the first 14 data points the sequence length would not be long enough to make a prediction.

Date	Return Squared	Intrahour Volatility	Response
02/02/2005 10:00	6.47E-05	0.0107	
02/02/2005 11:00	1.87E-05	0.0057	
...	...	...	1
03/02/2005 14:00	2.33E-06	0.0037	

Table 5.1: **Observation 1**: $\mathbf{x^{(1)}}$

Date	Return Squared	Intrahour Volatility	Response
02/02/2005 11:00	1.87E-05	0.0057	
02/02/2005 12:00	7.41E-05	0.00031	
...	...	...	1
03/02/2005 15:00	11E-04	0.00000031	

Table 5.2: **Observation 2**: $\mathbf{x^{(2)}}$

Date	Return Squared	Intrahour Volatility	Response
02/02/2005 12:00	7.41E-05	0.00031	
02/02/2005 13:00	1.23E-04	0.0082	
...	...	...	0
03/02/2005 16:00	1.58E-04	0.0019	

Table 5.3: **Observation 3**: $\mathbf{x^{(3)}}$

5.1.2 Forward Pass

Forward pass for an RNN is the where the network generates a prediction $\hat{y}_i$ from the features of an observation $\mathbf{x^{(i)}}$. In an RNN, as each successive data point, $\mathbf{x_t}$ in $\mathbf{x^{(i)}}$ enters the model, the model *remembers* the data points and finds relationships between consecutive values in the sequence. It achieves this through the hidden state.

We define the hidden state, $\mathbf{h_t}$ at time step, $t \in 1, ..., T$ by:

$$\mathbf{h}_t = \phi(W_{xh}\mathbf{x}_t + W_{hh}\mathbf{h}_{t-1} + b_h), \tag{5.2}$$

where $\phi(x)$ is the tanh activation function, W_{xh} and W_{hh} are weight matrices and the hidden layer bias is given by b_h. The hidden state, $\mathbf{h_0}$, is initialized as a vector of zeros.

The probability that the observation $\mathbf{x^{(i)}}$ lies in the increase (1) class is given by:

$$h(\mathbf{x^{(i)}}) = \sigma(W_{hy}\mathbf{h}_T + b_y). \tag{5.3}$$

We have that $\sigma(\cdot)$ is the sigmoid activation function, W_{hy} a weight matrix, b_y is the output layer bias, and $\mathbf{h_T}$ is the hidden state at the final time step in the sequence, T. The predicted class is given by:

$$\hat{y_i} = \begin{cases} 0 & \text{if } h(\mathbf{x^{(i)}}) < threshold \\ 1 & \text{if } h(\mathbf{x^{(i)}}) \geq threshold \end{cases}. \tag{5.4}$$

When we reach the end of the network, the model has analysed the relationships between all T data points in the sequence. It is possible for an RNN to generate an output at each time step, and this would be done using a many-to-many RNN architecture. We would use this architecture if we were using the model for a task such as sequence labelling, where we want to generate a response for each data point in the sequence. This process is described by [19]. However, as we are using our RNN for binary classification, we are only interested in the prediction the model makes at $t = T$, because at this point the prediction has taken into account all T data points in the sequence and uses it to make a single classification. We will use a many-to-one RNN architecture to achieve this.

As we observe in Equation (5.2) and (5.3), the RNN is parameterized by 5 parameters, $\mathbf{W} = (W_{xh}, W_{hh}, W_{hy}, b_h, b_y)$. In addition to the biases, which we have defined, we also have 3 weight matrices. Namely, we have the **input to hidden weight matrix**, W_{xh}, **hidden to hidden weight matrix**, W_{hh} and **hidden to output weight matrix**, W_{hy}. It is important to note that the parameter values, $\mathbf{W}$ do not change over different time steps, but the weight matrices and biases are the same at each time step in a trained network.

Throughout the next step we will explain the backward pass process for an RNN. To fully understand this and how it can lead to the vanishing gradient problem, we must understand both the size of the weight matrices, and the data it contains. The size of the weight matrices are determined by the number of time steps used to train the model and the number of features in the dataset. For example, if we trained an RNN on our intrahour volatility dataset, W_{xh} would be a (15 x 2) matrix, due to a time step of 15 and 2 feature variables. Before the model is trained, each entry of the matrix is initialized with a random value (usually a small value chosen from a normal distribution with zero mean and unit variance [32]). These weights are then updated as the model is trained, depending on how much each feature and data point in the sequence affects the response.

5.1.3 Backpropagation through time

Now we understand the forward pass process, we would like to explore how the gradient of our loss functions are calculated for an RNN. In the previous chapter, we explored backpropagation, which is the process used to train a FFNN. However, we left out the backward pass process of this algorithm. To train a recurrent neural network, the same general idea is used, however the algorithm is called backpropagation through time (BPTT). The reason "through time" is added is due to the way the gradient of the loss function is calculated.

This process is explored in [12] by Chen, however the backward pass process explained in this paper is specific to a model designed for sequence labelling. That is, given a sequence of data points $\mathbf{x^{(i)}} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_T}\}$, the model maps each data point to a corresponding response variable $\hat{y} = \{\hat{y_1}, \hat{y_2}, ..., \hat{y_T}\}$. As our **Reconstructed Intrahour Volatility Dataset** only contains a

single response value, y_i for each sequence $\mathbf{x^{(i)}} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_{15}}\}$, then we must re-adapt the process explained by Chen, to fit our application. We have explored Chen's algorithm, which can be found in the Appendix B, and throughout the following derivation we will make comparisons to this.

Our aim here is to find the derivatives of the loss function with respect to each of our parameters, $\mathbf{W}$. Throughout the next derivation, we assume that we are training an RNN with a single hidden layer, which analyses a sequence $\mathbf{x^{(i)}} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_{15}}\}$ to make a prediction $\hat{y}_i$ for the response variable y_i. The loss function will be a binary cross-entropy loss function.

We proceed by taking the partial derivative of $\mathcal{L}$ with respect to W_{hy}. As $\mathcal{L}$ is a function of $h(\mathbf{x^{(i)}})$, and $h(\mathbf{x^{(i)}})$ is a function of W_{hy}, then via the chain rule we can write:

$$\frac{\partial \mathcal{L}}{\partial W_{hy}} = \frac{\partial \mathcal{L}}{\partial h(\mathbf{x^{(i)}})} \frac{\partial h(\mathbf{x^{(i)}})}{\partial W_{hy}}. \tag{5.5}$$

Similarly, we yield the following derivative for the output bias:

$$\frac{\partial \mathcal{L}}{\partial b_y} = \frac{\partial \mathcal{L}}{\partial h(\mathbf{x^{(i)}})} \frac{\partial h(\mathbf{x^{(i)}})}{\partial b_y}. \tag{5.6}$$

In Chen's algorithm, when the above derivatives are calculated they summed over every time step in the sequence to incorporate the loss at each time step. Our derivative is only taken at the final time step. This is because we are only concerned with the how W_{hy} and b_y affect the prediction at the end of the sequence.

To compute the next three partial derivatives we must consider how the information from each hidden state affects the loss function. The parameters W_{xh}, W_{hh} and b_h are shared across each hidden state and all of these hidden states have an effect on the final prediction. Therefore, to capture all of this information, we backpropagate the derivatives of the hidden states over the whole time sequence to produce the following derivatives:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{k=1}^{15} \frac{\partial \mathcal{L}}{\partial h(\mathbf{x^{(i)}})} \frac{\partial h(\mathbf{x^{(i)}})}{\partial \mathbf{h}_{15}} \frac{\partial \mathbf{h}_{15}}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W_{hh}} \tag{5.7}$$

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{k=1}^{15} \frac{\partial \mathcal{L}}{\partial h(\mathbf{x^{(i)}})} \frac{\partial h(\mathbf{x^{(i)}})}{\partial \mathbf{h}_{15}} \frac{\partial \mathbf{h}_{15}}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W_{xh}} \tag{5.8}$$

$$\frac{\partial \mathcal{L}}{\partial b_h} = \sum_{k=1}^{15} \frac{\partial \mathcal{L}}{\partial h(\mathbf{x^{(i)}})} \frac{\partial h(\mathbf{x^{(i)}})}{\partial \mathbf{h}_{15}} \frac{\partial \mathbf{h}_{15}}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial b_h}. \tag{5.9}$$

In theory, these gradients would be used by our Adam optimizer to help the parameter values converge to their optimal values. We will now learn why this may not always be the case.

5.1.4 Vanishing (Exploding) Gradient Problem

The key limitation of the RNN model lies within a phenomenon called the vanishing gradient problem, which occurs during training. As we learnt in the Optimizers section, during training the model's parameters are updated in the opposite direction of the gradient in order to reduce the loss function. The problem is, we can experience difficulties in obtaining a suitable gradient for this process. What can happen is the gradient of our loss function can either become far too small or exponentially large, causing the vanishing (or exploding) gradient problem.

The consequence of the vanishing gradient problem is it will take the model's parameters far too long to converge to their optimal values. The exploding gradient problem will lead to an unstable training process as the parameters will experience large fluctuations as they're updated. This again makes it

difficult for the parameters to converge.

The specific reason the vanishing gradient problem occurs is due to the fact that RNNs need to back-propagate gradients over arbitrarily long sequences, which leads to a series of multiplications of our weight matrices. As we have learnt, the longer our sequence length is, the greater the siez of the weight matrices. However, this also means the absolute value of the entries in weight matrices will decrease as our sequence length increases. This is because as the number of data points in the sequence increases, the relative amount that each data point affects the response will decrease.

Chen briefly explains how the vanishing gradient problem is caused by the sequence of matrix multiplications. We will explore the cause of this problem in more depth by understanding how both the choice of activation function and the matrix multiplication of weights has an effect. To do this we will investigate the derivative in Equation (5.7). In particular, we will see that this problem occurs due to the $\frac{\partial \mathbf{h}_{15}}{\partial \mathbf{h}_k}$ terms. The following equations uses theorems from [6] applied to our dataset.

$$\frac{\partial \mathbf{h}_{15}}{\partial \mathbf{h}_k} = \prod_{i=k+1}^{15} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \tag{5.10}$$

$$= \prod_{i=k+1}^{15} \frac{\partial \mathbf{h}_i}{\partial \mathbf{c}_i} \frac{\partial \mathbf{c}_i}{\partial \mathbf{h}_{i-1}} \tag{5.11}$$

$$= \prod_{i=k+1}^{15} \text{diag}(\phi'(\mathbf{c_i}))W_{hh}. \tag{5.12}$$

If we consider the norm of each matrix, $\text{diag}(\phi'(\mathbf{c_i}))$ and W_{hh}, which represents its magnitude, we can see how each term is bounded. We will assume that our activation function, ϕ is a tanh activation function, whose gradient ranges between 0 and 1.

$$||\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}|| = ||\text{diag}(\phi'(\mathbf{c_i}))W_{hh}|| \tag{5.13}$$

$$\leq ||\text{diag}(\phi'(\mathbf{c_i}))|| \quad ||W_{hh}|| \tag{5.14}$$

$$\leq ||W_{hh}|| := \lambda. \tag{5.15}$$

This yields the following result:

$$||\frac{\partial \mathbf{h}_{15}}{\partial \mathbf{h}_k}|| \leq \lambda^{15-k} \tag{5.16}$$

Therefore for a weight matrix where the eigenvalues are less than 1, we can see that $\lambda^{15-k} \to 0$, which causes our gradient to vanish. An explanation of the exploding gradient problem can be shown in the same manner.

This limitation makes the basic RNN model impractical for our use case. Instead we are more interested in looking at variations of the RNN, with architectures designed to avoid this problem. In the next section we will explore the long short-term memory neural network.

5.2 Long Short-term Memory (LSTM) Model

The first RNN variation we will look at is the long short-term memory (LSTM) neurla network. As we will learn, this type of network is capable of learning long-term dependencies within time-series

data without experiencing the vanishing gradient problem. Unlike the basic RNN, which sequentially updates its hidden (memory) state, based solely on new data points and the previous hidden state, the LSTM has a more sophisticated way of remembering information. The LSTM contains two memory states, one which stores long term information, the other which stores short term information. The LSTM has a gating mechanism which controls how much new information (data) is inputted, remembered and then forgotten. As we will learn, this design allows the network to retain information from long sequences of data, without experiencing the vanishing gradient problem. Before proceeding, we define the Hadamard product operator which is used during forward pass for an LSTM:

Definition 5.2.1. Let P and Q be m x n matrices. The **Hadamard product** of P and Q is defined as:

$$[P \circ Q]_{i_j} = [P]_{i_j}[Q]_{i_j}, \quad \forall \quad 1 \leq i \leq m, \quad 1 \leq j \leq n. \tag{5.17}$$

Within the network two types of memory are stored, the long term memory, which we call the **cell state**, C_t and the short term memory, which we call the **hidden state**, $\mathbf{h_t}$. The cell state carries information from the whole sequence of data and is updated at each time step. The cell state simultaneously incorporates information from newly processed data points, in addition to retaining information from previous data points. This is done through two main sources of information, the input gate, $\mathbf{i_t}$ and forget gate, $\mathbf{f_t}$. On the other hand, the hidden state takes information from the immediate past and is used to calculate the output of the LSTM unit. Figure 5.2 shows the structure of a single LSTM unit. It's important to realise that a single LSTM unit is analogous to a sinlge unit in the basic RNN.

The key difference with an LSTM is the gating mechanism which controls how much of the model's states are remembered or forgotten within each time step. The three gates are described as follows:

- **Forget Gate,** f_t - controls how much of the cell state should be forgotten
- **Input Gate,** i_t - controls how much information should be added to the new cell state.
- **Output Gate,** o_t - This is used in parallel to the new cell state to update the hidden state, before generating the output.

Earlier, we explored the forward pass process for a basic RNN, including the equations required to obtain $h(\mathbf{x^{(i)}})$, $\hat{y_i}$ and explaining how the hidden state is updated. Equations (5.3) and (5.4) used to obtain $h(\mathbf{x^{(i)}})$ and $\hat{y_i}$ remain the same for an LSTM. However, we will now state the gate equations and explain how the hidden states, $\mathbf{h_t}$ and cell states, C_t are updated in an LSTM. This understanding is taken from [39]. We consider a single LSTM unit at time step, t:

1. The model receives a vector of input data, $\mathbf{x_t}$ and the hidden state, $\mathbf{h_{t-1}}$ from the previous step is fed into the cell.

2. The forget gate decides what percentage of the previous cell's memory should be retained. This percentage is calculated through the following equation:

$$f_t = \sigma(W_{xf}\mathbf{x_t} + W_{hf}\mathbf{h_{t-1}} + b_f), \tag{5.18}$$

   where, σ is the sigmoid activation function.

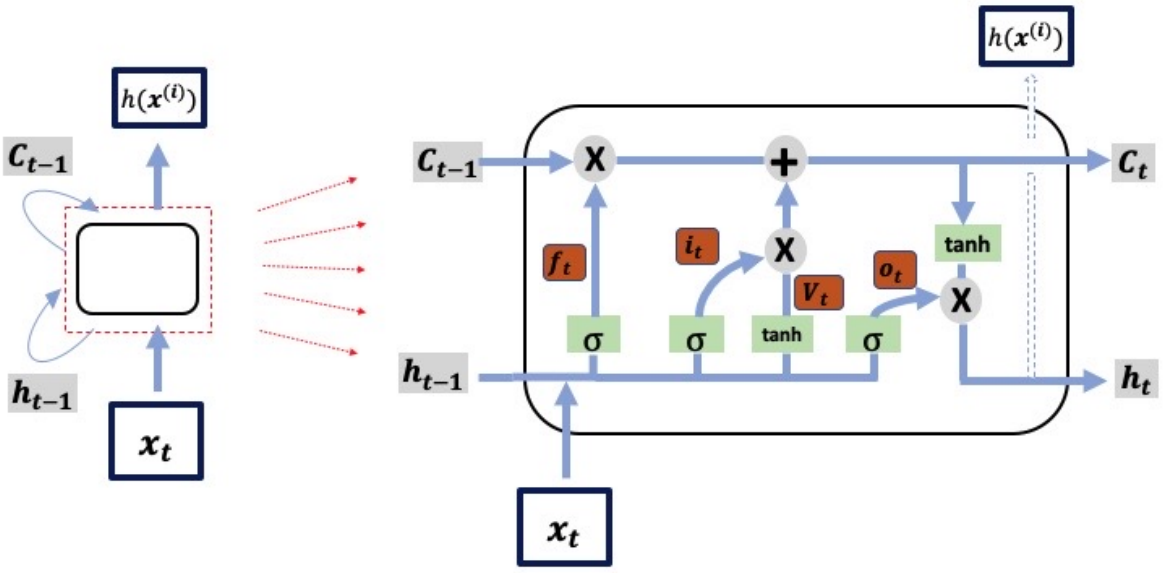


Figure 5.2: LSTM Unit

3. Simultaneously, the input gate decides which new information should be added to the cell's memory as per the following two equations:

$$i_t = \sigma(W_{xi}\mathbf{x_t} + W_{hi}\mathbf{h_{t-1}} + b_i) \tag{5.19}$$

and

$$V_t = \tanh(W_{xv}\mathbf{x_t} + W_{hv}\mathbf{h_{t-1}} + b_v), \tag{5.20}$$

where (5.19) can be thought of as the percentage of potential information to be added and (5.20) can be thought of the value of additional information to be added to the cell state. The tanh activation functions, gives this a value between -1 and 1.

4. The cell state, C_t is updated according to:

$$C_t = C_{t-1} \circ f_t + V_t \circ i_t, \tag{5.21}$$

which allows the long term memory to dynamically update as new information becomes available, and previous information becomes redundant. The input gate and forget gate operate simultaneously within the unit to update the cell state.

5. The output gate is updated through the equation:

$$o_t = \sigma(W_{xo}\mathbf{x_t} + W_{ho}\mathbf{h_{t-1}} + b_o). \tag{5.22}$$

This is then used to update the cell's hidden state:

$$\mathbf{h_t} = \tanh(C_t) \circ o_t \tag{5.23}$$

We observe that the four components f_t, i_t, V_t and o_t individually act as 4 basic recurrent neural networks. Their equations share the same format as Equation (5.2). This suggests these gates may individually experience the vanishing gradient problem (e.g. $\frac{\partial f_t}{\partial \mathbf{W}} \to 0$). We will learn that even though

this is the case, provided that our $\frac{\partial C_t}{\partial \mathbf{W}}$ terms do not vanish, then the gradients of the loss $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$ will also not vanish.

5.2.1 Backpropagation through time for an LSTM

As we have learnt in the above equations, our LSTM model contains significantly more parameters than the basic RNN. The LSTM is parameterized by the following parameters, $\mathbf{W} = (W_{xf}, W_{hf}, W_{xi}, W_{hi}, W_{xv}, W_{hv}, W_{xo}, W_{ho}, b_f, b_i, b_v, b_o)$, which must be learnt. In order to train this model, we are required to compute the value of the loss function with respect to each parameter. To avoid repetition with the previous section we will not go into the details of this process. However, we will explore one of the derivatives to understand how the gating mechanism of the LSTM helps to avoid the vanishing gradient problem.

Let's assume we are using a binary cross entropy loss function and we are required to compute $\frac{\partial \mathcal{L}}{\partial W_{xf}}$. We must consider the gradient of the loss function across each time step in the sequence and then aggregate the gradients from each. Specifically, we would look to compute:

$$\frac{\partial \mathcal{L}}{\partial W_{xf}} = \sum_t \frac{\partial \mathcal{L}}{\partial W_{xf,t}}. \tag{5.24}$$

We consider the following, $\forall t \in T$:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{xf,t}} &= \frac{\partial \mathcal{L}}{\partial C_t}\frac{\partial C_t}{\partial W_{xf}} && (5.25)\\
&= \prod_{k=t}^{T-1} \frac{\partial C_{k+1}}{\partial C_k}\frac{\partial \mathcal{L}}{\partial C_T}\frac{\partial C_t}{\partial W_{xf}} && (5.26)\\
&= \prod_{k=t}^{T-1} (f_{k+1})\frac{\partial \mathcal{L}}{\partial C_T}\frac{\partial C_t}{\partial W_{xf}} && (5.27)\\
&= (f_{t+1} \circ f_{t+1} \circ .. \circ f_T)\frac{\partial \mathcal{L}}{\partial C_T}\frac{\partial C_t}{\partial W_{xf}}. && (5.28)
\end{aligned}$$

We know as $f_i \in (0, 1)$ the further we go back through the sequence the smaller this component of the gradient becomes due to depletion from the forget gate. However, as we aggregate across all time steps, this shows how the above architecture addresses the vanishing gradient problem. Now that we understand how an LSTM is designed to successfully capture long-term dependencies with our data, we will demonstrate why the LSTM is suitable for our **Intrahour Volatility Dataset**.

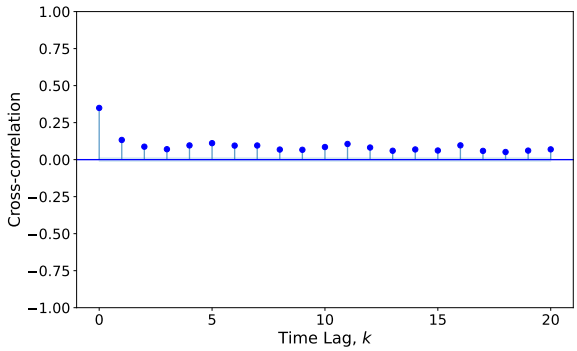
5.3 Modelling Temporal Dependencies

We have hypothesised that an LSTM is a good choice of model when working with time series data, as the model is able to learn temporal patterns in the data. However, we have not explicitly stated which patterns an LSTM is able to identify. As alluded to in **Example 2.3.1** from Chapter 2, it is the temporal dependencies that exist within a sequence of data points that the LSTM is able to learn. Due to the network's *memory* it is able to remember correlations within a sequence of data. We will now demonstrate that these correlations must exist for the LSTM to be useful. Consider 2 versions of the **Intrahour Volatility Dataset**:

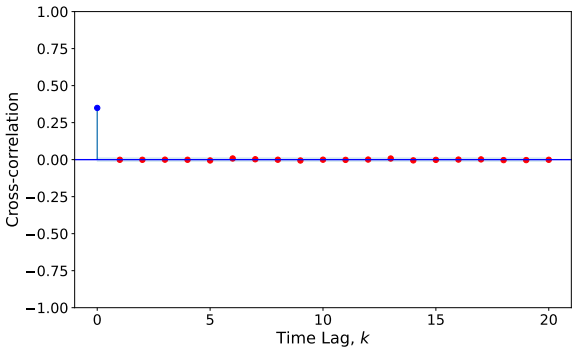
1. **Ordered Intrahour Volatility Dataset** - This is our original dataset, it is organised chronologically, therefore maintaining the chronological ordering of events.

2. **Shuffled Intrahour Volatility Dataset** - In this dataset we have randomly re-ordered all of the observations. It has lost its chronological ordering.

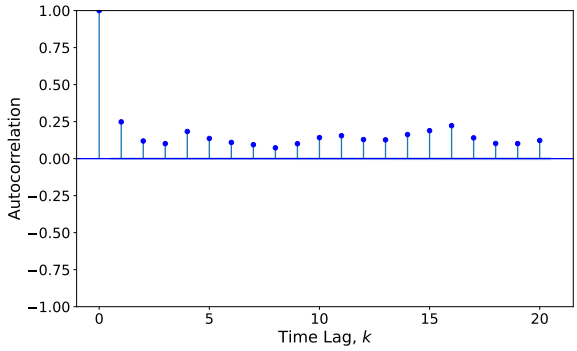
Within both datasets, the data values and distribution of the classes are exactly the same. The only difference is the ordering of the data. Figure 5.3 demonstrates how shuffling the time series data removes time-series correlations within the dataset. In the shuffled datasets, the autocorrelation within the intrahour volatility and the cross correlation between the intrahour volatility and return squared is no longer statistically significant.



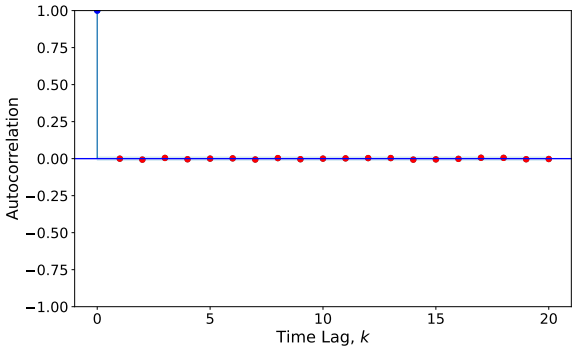
(a) **CCF (Intrahour Volatility over Return Squared) for Ordered Data**



(b) **CCF (Intrahour Volatility over Return Squared) for Shuffled Data**



(c) **ACF (Intrahour Volatility) for Ordered Data**



(d) **ACF (Intrahour Volatility) for Shuffled Data**

Figure 5.3: The figures on the left demonstrate that statistically significant time series correlation exists within the ordered dataset. The figures on the right demonstrate how the shuffled dataset no longer exhibits statistically significant correlation.

We trained identical LSTMs on the shuffled and ordered datasets (after reconstructing) and a FFNN on the ordered dataset. Figure 5.4 demonstrates how LSTM's performance declines after removing temporal dependencies within the dataset.

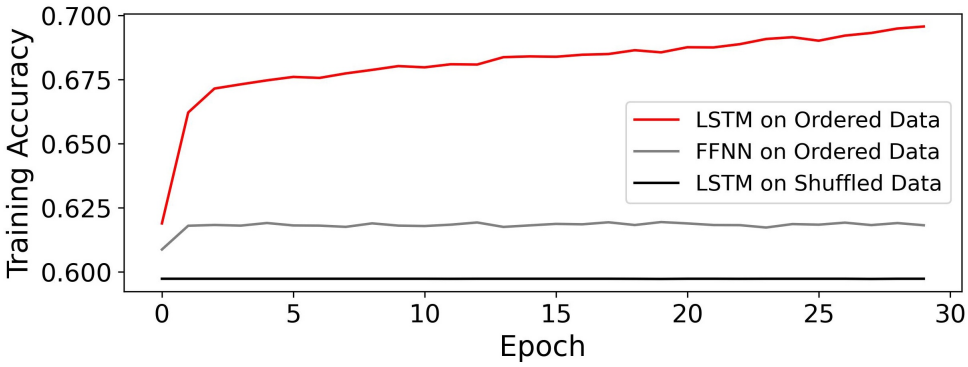


Figure 5.4: These plots show the performance of a single layered FFNN and LSTM trained on the ordered data and an LSTM on the shuffled data. The code producing this plots can be found in [25].

The LSTM model trained on the ordered dataset reached a training accuracy of 69% after 30 epochs. In comparison, the LSTM model trained on the shuffled dataset stagnated at an accuracy of 60%, underperforming even compared to the FFNN.

This result concludes that statistically significant autocorrelation within the response variable, or, statisticaly significant cross correlation between an additional exogenous variable and the response variable, can positively impact the performance of an LSTM. This conclusion will help us to conduct a feature analysis in our final chapter when training our final models to forecast intraday realised volatility.

Chapter 6

Optimized Models

It is important to understand that although accuracy is an important indicator of how useful our model is, we must consider the context in which we would use one of these models. If we were on a trading floor within an investment bank and were tasked with predicting which direction the volatility would move next hour, we would be required to make this prediction both accurately and quickly. Therefore the time taken to train a model is also an important metric to consider.

Throughout this chapter we will explore how many parameters our different neural network models have. We will introduce the final RNN models, which have a reduced number of parameters. Clearly, a model with fewer parameters requires calculating fewer gradients of the loss function, therefore speeding up the training process.

Number of Parameters

All of the neural network models we have trained so far have been using the Keras Python library [13]. Before we explore the number of parameters in our LSTM model, we will focus our attention back to the feed forward neural network. In chapter 4, we introduced the structure of an FFNN dense layer. We will now calculate the number of parameters in an FNN with 2 units in the input layer, 80 units in the hidden layer and a single unit in the output layer.

$$\begin{aligned} \text{Number of Parameters} &= \text{input to hidden weights} + \text{hidden to output weights} \\ &\quad + \text{hidden layer biases} + \text{output layer biases} \\ &= 2 \cdot 80 + 80 \cdot 1 + 80 + 1 \\ &= 321 \end{aligned}$$

In an LSTM, we are aware that the architecture contains a sequence of T units, where each unit contains 4 gates; input, forget, output (and candidate gate, V_t), each containing weight matrices and biases. However, we haven't yet discussed the idea of parallel units. In our theory so far, we have assumed that our model only contains a single sequence of units, which means our hidden state would be a vector of length, T. In practice, when our LSTM models are trained using the Keras library, we train multiple parallel sequences, leading to multiple hidden states.

Note. In Keras, the *number* of parallel sequences of LSTM units is defined as "units". We will define the *number* of parallel sequences as the *hidden state dimension* to avoid confusion with an LSTM unit.

Example 6.0.1. In this example, we will calculate the number of parameters in a single LSTM layer with hidden state dimension, 80 and 2 feature variables. A single unit with the increased dimensions can be visualised in Figure 6.1. We will start by exploring the number of parameters in one of the gates and then multiplying by 4 (as each gate contains the same number of parameters). We recall the equation for the forget gate:

$$f_t = \sigma(W_{xf}\mathbf{x_t} + W_{hf}\mathbf{h_{t-1}} + b_f). \tag{6.1}$$

This model will contain the following number of parameters:

- W_{xf} parameters: $2 \cdot 80 = 160$
- W_{hf} parameters: $80 \cdot 80 = 6400$
- b_f parameters: $1 \cdot 80 = 80$
- total number of parameters in forget gate = 6640
- total number of parameters in LSTM layer = $6640 \cdot 4 = 26560$.

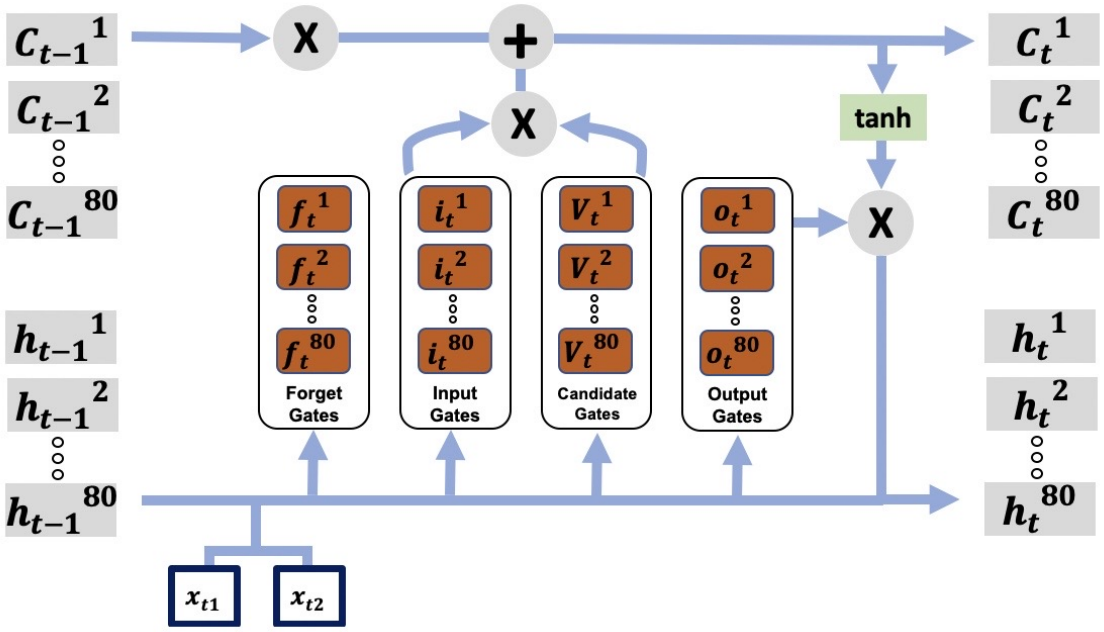


Figure 6.1: This figure illustrates a single LSTM unit with a hidden state dimension of 80.

As we can see there are considerably more parameters in the LSTM model than there are in the FFNN model. We will now introduce the gated recurrent unit (GRU) neural network model, which has a similar architecture to the LSTM, but fewer parameters to learn for a given hidden state dimension and number of features. [37] compares the performance of the LSTM and GRU by considering data sets and sequences of different size. The paper shows that due to the simpler architecture of the GRU model, with fewer parameters to learn, it is faster to train the GRU model than the LSTM model, yet still providing a similar accuracy. As we will learn in the next section, the GRU model contains 1 less gate than the LSTM. Therefore, the number of parameters in the GRU cell is reduced by approximately 25%. We trained LSTM and GRU models using the **Reconstructed Intrahour Volatility Dataset**, yielding the results in Table 6.1.

Model	No. of Parameters	Training Time (seconds)	In-sample Accuracy
LSTM	26560	111	0.687
GRU	20160	95	0.689

Table 6.1: This table compares the training time and in-sample accuracy from training identical LSTM and GRU models on the **Reconstructed Intrahour Volatility Dataset**. Each model had a hidden state dimension of 80. Both models were trained over 20 epochs. The Python code producing these results can be found in [25].

6.1 Gated Recurrent Unit Model

As we have stated, the architecture of a gated recurrent unit (GRU) is simpler than the architecture of an LSTM model. Unlike the LSTM which has 4 gates (including candidate gate), in the GRU, there are only 3 gates (including temporary memory) namely the update gate, reset gate and temporary memory gate. The functionality of the update gate combines the role of the input and forget gate in the LSTM. Additionally, where the LSTM has both a hidden state and cell state, the GRU only has a hidden state, $\mathbf{h_t}$.

Earlier, we explored the forward pass process for a basic RNN, including the equations required to obtain $h(\mathbf{x^{(i)}})$, $\hat{y}_i$ and explaining how the hidden state is updated. Equations (5.3) and (5.4) to obtain $h(\mathbf{x^{(i)}})$ and $\hat{y}_i$ remain the same for a GRU. However, we will now state the gate equations and explain how the hidden state, $\mathbf{h_t}$ is updated in a GRU. This understanding is taken from [34]. We consider a single GRU cell at time step, t:

1. The model receives the input data $\mathbf{x_t}$ at time step, t.

2. the reset gate, r_t decides what proportion of the previous hidden state, $\mathbf{h_{t-1}}$ is used in temporary memory. It takes a value between 0 and 1. If r_t takes a value of zero then the temporary memory will be made up of information from the current input $\mathbf{x_t}$ only. It is computed according to:

$$r_t = \sigma(W_{xr}\mathbf{x_t} + W_{hr}\mathbf{h_{t-1}} + b_r), \tag{6.2}$$

   where, σ is the sigmoid activation function.

3. The temporary memory, $\tilde{\mathbf{h}}_\mathbf{t}$ is calculated according to:

$$\tilde{\mathbf{h}}_\mathbf{t} = \tanh(W_{x\tilde{h}}\mathbf{x_t} + W_{h\tilde{h}}(r_t \circ \mathbf{h_{t-1}}) + b_{\tilde{\mathbf{h}}}) \tag{6.3}$$

4. the update gate determines the extent to which the updated hidden state is composed of the current hidden state vs the current temporary memory. It can filter out information from either the previous hidden state or the current temporary memory. It is computed according to:

$$z_t = \sigma(W_{xz}\mathbf{x_t} + W_{hz}\mathbf{h_{t-1}} + b_z) \tag{6.4}$$

5. The hidden state, $\mathbf{h_t}$ is updated through the following process:

$$\mathbf{h_t} = (1 - z_t) \circ \mathbf{h_{t-1}} + z_t \circ \tilde{\mathbf{h}}_\mathbf{t} \tag{6.5}$$

We now understand, due to a reduced number of parameters in the GRU, and shown through our empirical results in Table 6.1 that this model has a shorter training time over a set number of epochs than the LSTM. The GRU model also achieved greater in-sample accuracy than the LSTM on our data. In this test we chose each model to be trained over 30 epochs. The purpose of this was to compare training times over a fixed number of epochs.

If the same model were to be trained in industry, we would optimise the hyperparameters (using Walk-Forward or Nested cross validation) to find the optimal number of epochs, striking the balance between over-fitting and under-fitting. This idea motivates our final model, the optimized GRU, which, if trained using a cross-validation technique, we would expect to have a shorter training time than the standard GRU. The optimized GRU is an adaptation of the standard GRU, designed to increase its learning efficiency (we will explore why shortly). We would expect the optimized GRU to have a shorter training time than the standard GRU based on the following logic:

$$\begin{aligned} \text{learning efficiency increases} &\Rightarrow \text{model fits to training data after reduced number of epochs} \\ &\Rightarrow \text{cross validation will choose reduced number of epochs as optimal} \\ &\Rightarrow \text{model is trained in a shorter time.} \end{aligned}$$

6.2 Optimized Gated Recurrent Unit Model

The optimised gated recurrent unit (OGRU) was introduced by [36] and it was designed to improve the learning efficiency of the standard GRU. The paper is focused specifically on improving the learning rate of **deep** GRU neural networks. We will compare the GRU and OGRU cells with a single-layered neural network model. We learn that the GRU is not particularly effective at filtering out redundant information from new data points, particularly with time-series data of long sequences.

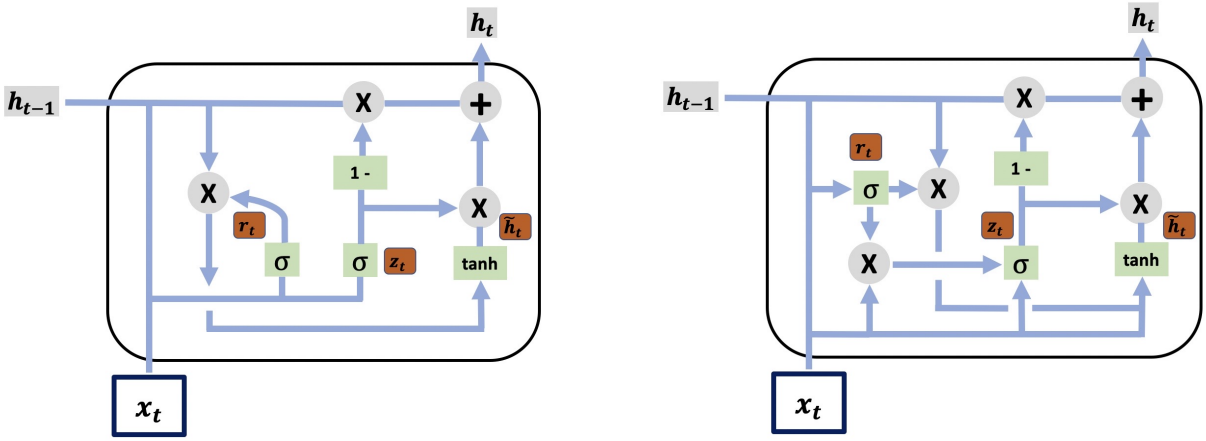
The reset gate is used to control the composition of the temporary memory between the current hidden state and input data. However, if the temporary memory is dominated by input data, the update gate cannot filter out redundant information from the previous hidden state and temporary memory (dominated by input data) at the same time. This can lead to information from $\mathbf{x_t}$, already present in the hidden state, being redundantly included in the updated hidden state. The OGRU gating structure has the following adaptation to prevent this from happening:

the $\mathbf{x_t}$ term in the update gate, Equation (6.4), is replaced with $(r_t \circ \mathbf{x_t})$, leading to the following revised equation for the update gate:

$$z_t = \sigma(W_{xz}(r_t \circ \mathbf{x_t}) + W_{hz}\mathbf{h_{t-1}} + b_z) \tag{6.6}$$

This will reduce the amount of redundant information from $\mathbf{x_t}$ being added to the updated hidden state due to the following reasoning. If the reset gate has a value close to zero, then the temporary memory, $\tilde{\mathbf{h}}_\mathbf{t}$ will be dominated by input data. However, the reset gate now introduced in the update gate will decrease the contribution from $\mathbf{x_t}$. This allows the update gate to filter out redundant information from the input data and previous hidden state simultaneously. The modified gate can be visualised in Figure 6.2.

To compute $(r_t \circ \mathbf{x_t})$, we need to align the dimensions of r_t with the number of input features. This required reducing the dimensionality of r_t, which we achieved by averaging over the 80 hidden states. Although this adjustment slightly altered the function of the reset gate, it was necessary to integrate this feature.



(a) GRU Cell

(b) OGRU Cell

Figure 6.2: Comparison between the Standard GRU and Optimised GRU cell structure.

Example 6.2.1. We compared the learning efficiency of the GRU and OGRU by training identical single-layered GRU and OGRU neural networks on the **Reconstructed Intrahour Volatility Dataset**. Figure 6.3 shows that after 50 epochs, the OGRU model had reached a training loss of 0.373, where as the GRU model had only reached a training loss of 0.503. This result demonstrates the OGRU's ability to learn from the data at a quicker rate than the standard GRU.

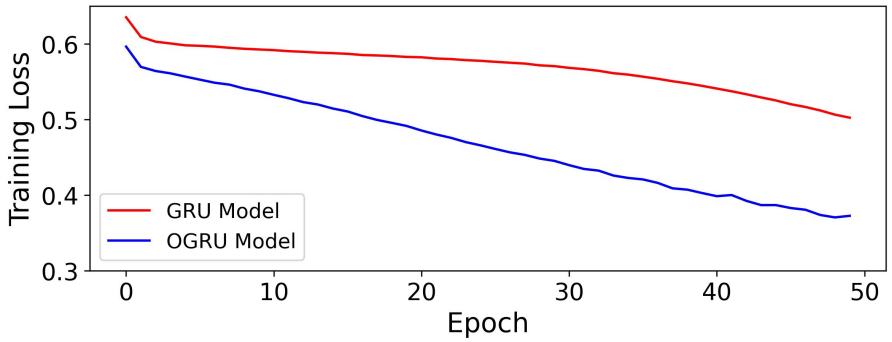


Figure 6.3: This figure compares the learning efficiency of the OGRU against the GRU model. The code which produced this plot can be found in [25].

Practical Note. Previously, for all of the models trained in this project, we used the Keras Python package [13] to develop our architecture and train our models. However, there are limitations to this package as it does not not allow us to directly modify the flow of information in each GRU cell. Therefore we were required to develop and initialise a custom made OGRU cell using the Tensorflow [4] library, allowing us to amend the standard design. The Python code we used to develop the OGRU cell can be found in [25].

Chapter 7

Forecasting Intraday Realised Volatility

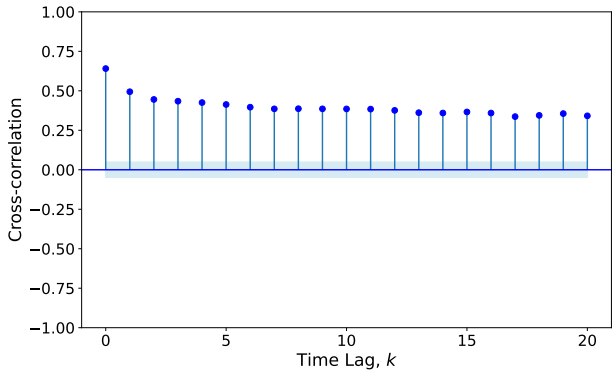
In this Chapter, we will apply everything we have learnt about recurrent neural networks and time series data to forecast the directional movement of intraday realised volatility. We learnt that the LSTM and GRU were both suitable models to be trained with the **Reconstructed Intrahour Volatility Dataset**. Therefore, these are the models we will focus on in this chapter. Throughout the previous chapters, when we trained our models we did not undergo a validation stage. We simply initialised the model with a set of hyperparameters. The reason for this was to compare the strengths and weaknesses of our model in an unbiased way. Having achieved this, in this chapter we will also validate our models to maximise the out-of-sample performance. The main performance metrics we will be evaluating are out-of-sample accuracy and AUC, as these provide the best general performance of a model. It may also be of interest to explore precision, sensitivity and specificity to determine which type of finance practitioner our models would be most useful for.

We will use the **Intraday Volatility Dataset** to train all of our models and we will begin with a feature analysis to narrow down our choice of features. As stated in Chapter 2, 20% of our data, from the years 2021 to 2024, will be used as the test set. Clearly, this data cannot be included in any of our feature analysis because we must treat this data as unseen.

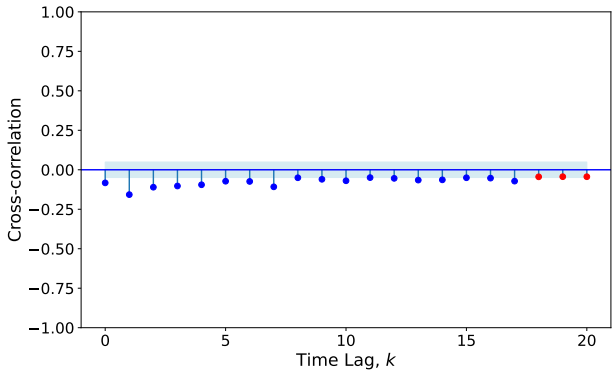
7.1 Feature Analysis

Before we begin, we must understand what we are looking for from each feature. As we learnt in Chapter 5, the key driver in selecting our features will be choosing ones which exhibit temporal correlations with Intraday Volatility. Therefore we will look for features where significant cross-correlation exists between *Intraday Volatility* over the given feature. On the other hand, we want to avoid features which exhibit a high linear correlation with each other. This is to avoid including variables which provide redundant information.

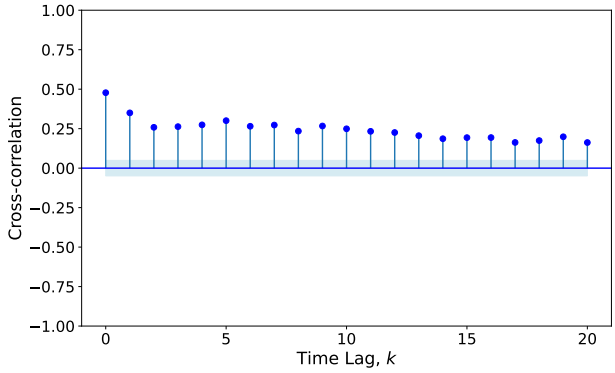
From Figure 7.1b and 7.1d, we see that the *Return* and *Range* variables exhibit weak cross-correlation with *Intraday Volatility*. In particular, the *Range* variable does not exhibit statistically significant cross correlation for a lag of $k = 5, ..., 18$. Therefore these two variables are contenders to be removed from the feature set. From Figure 7.2 we can see that *Range* variable also exhibits a high linear correlation of 0.47 and 0.58 with *Return Squared* and *Volume* respectively. Therefore we decide to remove *Range* from the feature set. As *Return* exhibits very low linear correlations with all of the other variables, we decide to keep this variable in the feature set. This is because there may be spatial relationships between the *Return* variable and the *response*, which could be learnt by the neural networks.



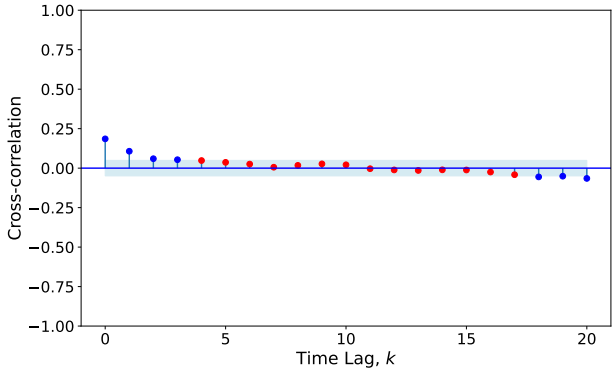
(a) Volume



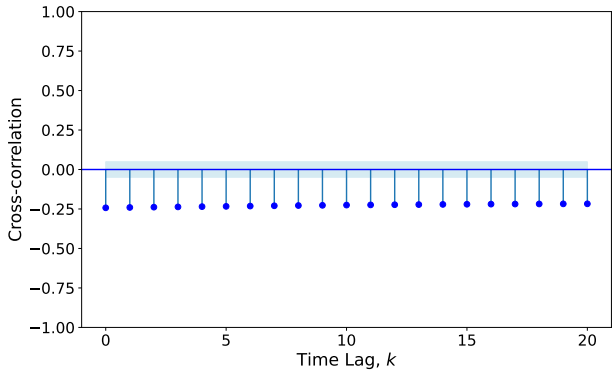
(b) Return



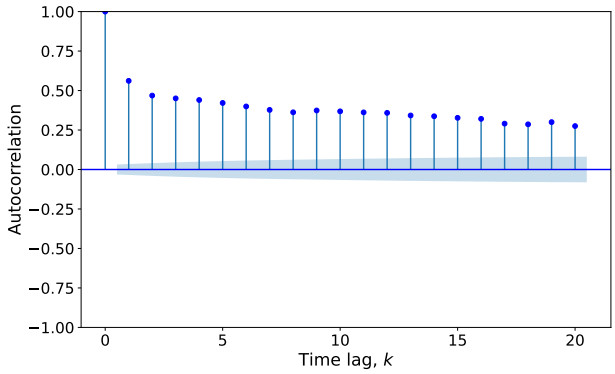
(c) Return Squared



(d) Range



(e) EMAF



(f) Intraday Volatility

Figure 7.1: Plots 7.1a to 7.1e are CCF plots for *Intraday Volatility* over the stated feature. The final plot 7.1f is the ACF plot for *Intraday Volatility*. The shaded blue region around the x-axis represents the confidence interval for cross correlation, c_k or autocorrelation, r_k at each time lag , k based off the hypothesis tests described in Chapter 3.

The additional benefit of exploring autocorrelation and cross correlation in our feature analysis is it shows us how far back the variables are correlated. We can see that for most of our variables there is significant cross correlation up to a time lag of $k = 20$. We need to be aware that due to the nature of autocorrelation and cross correlation, even if there is statistically significant correlation at larger time lags (e.g. $k = 20$), this correlation may be as a result of the correlation between *intraday volatility* and the feature at shorter time lags (i.e. from $k = 1, 2, 3$). With this in mind we will choose our sequence length for our RNN models to be of length 10.

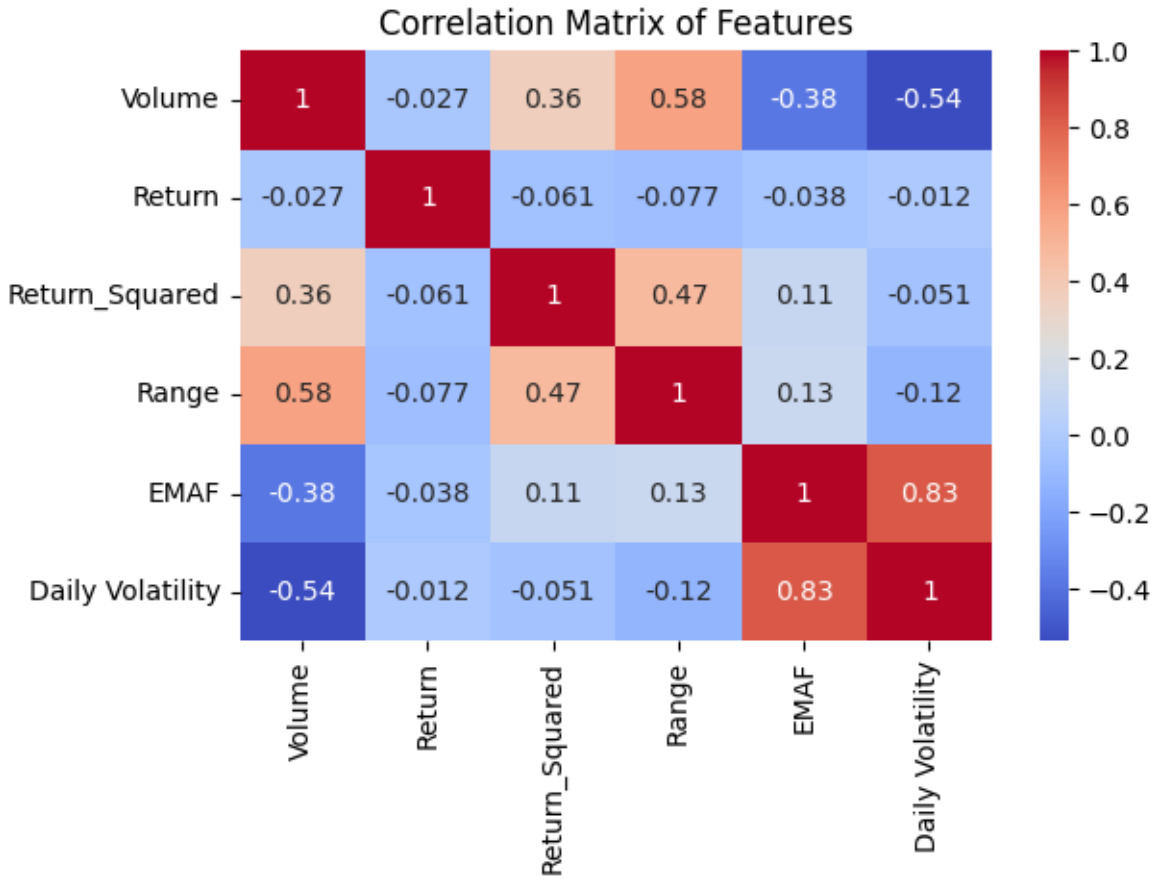


Figure 7.2: Correlation Matrix for the 6 possible feature variables. This matrix shows the simple linear correlations between each pair of variables.

Analysis of the Response Variable

Before we train our models on the dataset, we would like to explore the distribution of our response variable in the training data. When tackling a binary classification problem, it is helpful if we have an even distribution of 0s and 1s. If the response data is unbalanced the models could become biased and simply predict the majority class more of the time. This could lead to a large disparity in some of the performance metrics. This is why it's important to consider a variety of metrics and explore the ROC.

The response variable from the **Intraday Volatility Dataset** is shown in Table 7.1. We can see that our response data contains a relatively even distribution of 0's and 1's and therefore it is suitable to use. This is expected due to the stochastic nature of the stock market.

Response	Frequency	Percentage
0	672	56.2%
1	524	43.8%

Table 7.1: Distribution of the response variable in the training data.

7.2 Applying our Classification Algorithms

Now that we have selected our feature set and explored the response data we will train and validate our models. For the optimal GRU and LSTM models we obtain, we will record their out-of-sample performance.

7.2.1 Baseline Models

It is important that we develop a benchmark to compare our models to. By establishing the performance of traditionally used models when trained with our data, we can gauge the improvement in predictive performance of the RNN models. If the RNN models don't outperform the baseline models, then they have failed to identify additional patterns within the data. We will consider 2 baseline models and provide a motivation for each choice:

Baseline Model 1: **FFNN** - We train a single layered FFNN with 80 neurons in the hidden layer. This model should be able to capture non-linear, spatial relationships in the data however it will fail to capture temporal relationships.

Baseline Model 2: **Multivariate Logistic Regression** - We train a multivariate logistic regression model on the same 50 variables that are used to make a single prediction in our RNN model, which are the previous 10 days of observations for each of the 5 chosen features. This model will use the same data sequences as the RRN uses to make a prediction. However, we would not expect this model to learn temporal correlations within the sequence. It will be limited to finding linear relationships between each of the 50 predictors and the response. We will not explore the mathematics of a multivariate logistic regression as it is out the scope of this paper, however details of this model can be found in [8].

We recorded the performance metrics for the baseline models to see if we can improve with our RNN models. Table 7.2 shows the out-of-sample performance for both baseline models.

Performance Metric	**Accuracy**	**Precision**	**Sensitivity**	**Specificity**	**AUC**
FFNN	0.603	0.730	0.295	0.896	0.60
Logistic Regression	0.675	0.660	0.682	0.669	0.68

Table 7.2: Out-of-Sample Results for the baseline models. The Python code producing these results can be found in [25].

We will train and validate our RNN models on a reconstructed version of the **Intraday Volatility Dataset**, with sequence lengths of 10. This dataset is reconstructed in the same manner as the intrahour dataset from Chapter 5.

7.2.2 Final Comparisons

To train our RNN models, there are a number of hyperparmeters to choose from, each of which could produce significantly different results. In order to tune our hyperparameters for the RNN models we used a random search optimizer with 100 iterations of the space to choose sets of values. We used walk-forward time series cross validation to validate each model. Table 7.3 provides an overview of the search space and optimal values found in the validation process.

RNN Model	Hyperparameter	Search Space	Optimal Value
LSTM	Number of Epochs	(10, 30)	28
	Batch Size	[16, 32, 64]	16
	Hidden Layer Dimension	(50, 150)	147
GRU	Number of Epochs	(10, 30)	22
	Batch Size	[16, 32, 64]	16
	Hidden Layer Dimension	(50, 150)	119

Table 7.3: Hyperparameter Space

After finding the hyperparameters which optimized the models for our data, we trained our final models on this set of values and used our two optimal models to make predictions on the out-of-sample data.

Performance Metric	Model	Accuracy	Precision	Sensitivity	Specificity	AUC
In-Sample(2005-2020)	LSTM	0.684	0.676	0.596	0.758	0.74
	GRU	0.686	0.697	0.558	0.887	0.74
Out-of-Sample (2021-2024)	LSTM	0.674	0.676	0.645	0.708	0.75
	GRU	0.675	0.720	0.527	0.807	0.76

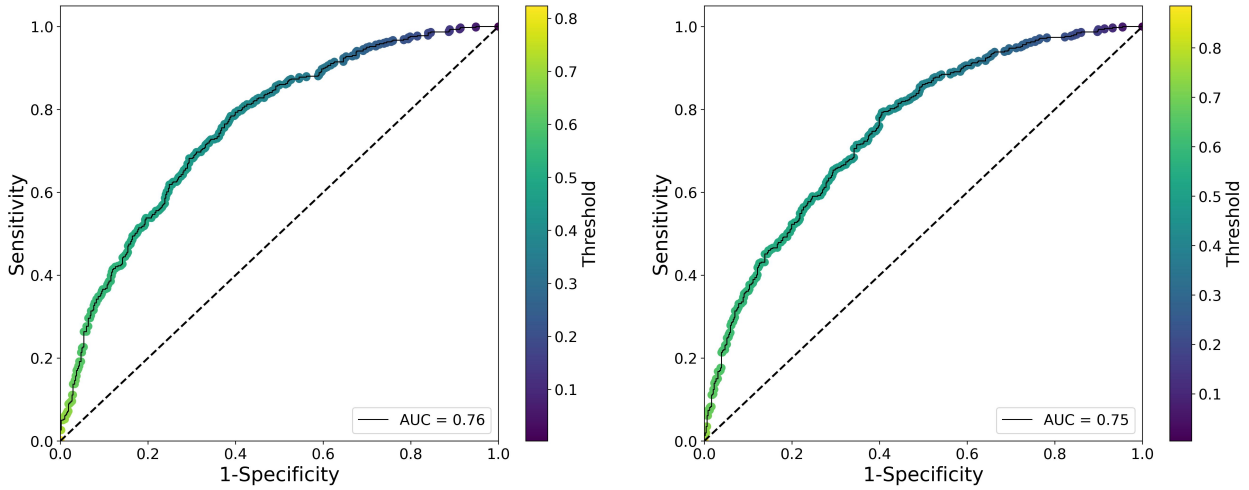
Table 7.4: Recurrent Neural Network Results. The code producing these results can be found in [25].

7.2.3 Discussion of Results

In this chapter, we compared the out-of-sample predictive performance of the GRU and LSTM model against our two baseline models, the FFNN and Multivariate Logistic Regression, in predicting the directional movement of intraday realised volatility.

The LSTM and GRU models outperformed both of the baseline models. The best metric to use for comparison is the AUC, as it compares the performance over all threshold values. The LSTM and GRU achieved similar AUC scores of 0.75 and 0.76, respectively. The FFNN and Logistic regression model achieved AUC scores of 0.60 and 0.68, respectively. We can conclude that both RNN models were able to uncover and learn temporal correlations within the dataset, where as the baseline models were not.

Both of the RNN models achieved high specificity and low sensitivity scores. The outcome stems from the models having a slight bias towards predicting the decrease (0) class, which led to an increased count in both true negatives and false negatives. The models exhibited this bias because the training data contains a larger proportion of the decrease (0) class than increase (1) class. To counter this bias and obtain a more balanced number of false increases and decreases, we can decrease the threshold value. For the LSTM model, we can see from Figure 7.3b, by setting the threshold at 0.49, this results in a sensitivity and specificity score of 0.670 each, Similarly, for the GRU model, we can see from Figure 7.3a a threshold value of 0.48 achieves a sensitivity and specificity score of 0.667 each.



(a) Optimal GRU ROC

(b) Optimal LSTM ROC

Figure 7.3: These plots give the ROC for the out-of-sample results from the optimal RNN models.

7.3 Conclusion

Within this project we formulated a time series binary classification problem to forecast the directional movement of both intrahour and intraday realised volatility of the AAPL stock. The purpose was to compare and evaluate neural network models for a time series forecasting task.

It is clear from the intraday results that the recurrent neural network models outperformed the feedforward neural network and logistic regression models, achieving greater AUC and out-of-sample accuracy scores. We learnt that the feedforward neural network model is limited to learning spatial relationships in a data set, where as the recurrent neural network models are able to learn both spatial and temporal relationships in a data set.

From both the intraday and intrahour results, we can see that the LSTM and GRU exhibited minimal difference in performance. However, we can conclude that the GRU was the better model for 2 reasons. Firstly, from the intraday results, the GRU achieved a greater out-of-sample accuracy than the LSTM. These results were taken after each model's hyperparameters had been optimised using walk-forward cross validation, therefore providing an unbiased comparison. Secondly, from the intrahour results, we observed that the GRU model had a considerably shorter training time than the LSTM model (15% reduction).

We also explored the optimised GRU in an attempt to further reduce the training time of the standard GRU model. We learnt that by combining the update and reset gate in the standard GRU model, we were able to increase the learning efficiency of the model, meaning that the model's parameters would converge to their optimal values in a smaller number of epochs. The main challenge we faced was comparing the out-of-sample accuracies of the GRU and OGRU. As our OGRU model was custom made using the Tensorflow library, rather than taken from the Keras library, it was difficult to undertake walk-forward cross validation. Therefore, it was not possible to compare the models in an unbiased way.

Finally, we learnt by using feature variables that exhibit statistically significant time series correlation (autocorrelation or cross correlation) with the response variable, we can improve the performance of a recurrent neural network model. We used this conclusion to conduct a feature analysis.

Overall, this project has provided fascinating results in the realms of recurrent neural networks, time series analysis and volatility forecasting, leaving several directions for further research.

7.3.1 Further Research

Firstly, we could explore higher frequency datasets (more observations) with a larger parameter space. While a single hidden layer proved to be adequate for our datasets, we may be able to improve our predictive power by investigating larger datasets that exhibit more complex, non-linear characteristics. These datasets could be used to train deep recurrent neural network models, where we could explore the effects of additional hidden layers.

Secondly, within these deep recurrent neural networks, there is potential to improve predictive power by further modifying the gating mechanisms of the standard LSTM and GRU models. We were introduced to this idea in Chapter 6 to increase learning efficiency, however, by further enhancing the gates it may lead to the models learning additional temporal relationships.

Finally, from the point of view volatility forecasting, it would be beneficial to incorporate the effects of seasonality into our models. Seasonality refers to fluctuations in the data occurring at regular time intervals. In our case, a stock's volatility tends to be greater during the first and final hour of the trading day. By either adjusting the neural network's architecture or gating mechanism we may be able to include the effects of seasonality into the models.

Appendix

Appendix A: Liquidity Provision in Kyle's Model

In this section, we aim to give an explanation of how a market maker can adjust their bid-ask spread according to the the market dynamics and the type of market participant that they are trading with. Consider an asset from time 0 to time t, evolves as

$$S_t = S_0 + r_t$$

where $r_t \sim N(0, \sigma_S^2)$.

Suppose the asset is traded by three types of market participant:

1. **Noise** traders, who we will denote $\mathcal{T}^N$, who buy $N_0 \sim N(0, \sigma_N^2)$ amount of shares, independent of the information available and the market price, they simply want to execute an order regardless. These types of trader could be retail traders or institutions rebalancing their capital.
2. An **informed** or **insider** trader, denoted $\mathcal{T}^I$, who has an informational advantage and that they know exactly what r_t will be, but they do not know what the volume of stock traded by the noise trader N_0 will be. $\mathcal{T}^I$ will trade M_0 shares.
3. A **liquidity provider**, denoted $\mathcal{T}^{LP}$, who will buy/sell all of the shares listed by the other two traders for a price of $\lambda Y_0 = \lambda(M_0 + N_0)$, which is linear in the total order flow.

We can determine that:

- The liquidity provider $\mathcal{T}^{LP}$ can see both the order flow of $\mathcal{T}^N$ and $\mathcal{T}^I$.
- The informed trader $\mathcal{T}^I$ will adjust their position M_0 based on the value of r_t to maximise their utility.
- $\mathcal{T}^{LP}$ then adjusts their price based on the total flow in order to remain market-neutral.

In order to solve for the optimal bid-ask spread of the liquidity provider, we need to determine the optimal position of the informed trader. Let us denote $\mathcal{P}(\mathcal{T}^I)$ to be the profit of the informed trader. Therefore we have that the expected profit is given as

$$\begin{aligned}
\mathbb{E}^I\left[\mathcal{P}(\mathcal{T}^I)\right] &= \mathbb{E}^I\left[M_0\left(\underbrace{S_0 + r_t}_{S_t} - \underbrace{\lambda Y_0}_{\text{tradeable price}}\right)\right] \\
&= M_0(S_0 + r_t - \lambda\mathbb{E}^I\left[M_0 + N_0\right]) \\
&= M_0(S_0 + r_t - \lambda M_0).
\end{aligned}$$

This is a quadratic function in M, which in order to be maximised can be solved simply under first order conditions, such that the optimal trade of the informed trader is given as

$$M^* = \frac{S_0 + r_t}{2\lambda}$$

where $\frac{1}{\lambda}$ can be interpreted as the *market depth*. The value of λ is therefore at the discretion of the liquidity provider and should be chosen such that their expected profit is zero (market-neutral), that is they should choose λ^* such that

$$\begin{aligned}\mathbb{E}\left[\mathcal{P}(\mathcal{T}^{LP}) \mid Y_0\right] =& 0 \\ =& \mathbb{E}\left[Y_0(\lambda^* Y_0 - S_0 - r_t) \mid Y_0\right] \\ =& \lambda^* Y_0^2 - Y_0 \mathbb{E}\left[S_0 + r_t \mid Y_0\right] \end{aligned} \tag{7.1}$$

Assuming that $\mathcal{T}^I$ is trading optimally, from previously their position will be $M^* = \frac{S_0 + r_t}{2\lambda}$. Therefore $Y_0 = N_0 + \frac{S_0 + r_t}{2\lambda}$. For the conditional expectation, we can show

$$\begin{aligned}\mathbb{E}\left[S_0 + r_t \mid Y_0\right] &= \mathbb{E}[S_0 + r_t] + \frac{\mathrm{Cov}(Y_0, S_0 + r_t)}{\mathrm{Var}(Y_0)}(Y_0 - \mathbb{E}[Y_0]) \\ &= \frac{\frac{\sigma_S^2}{2\lambda}}{\frac{\sigma_S^2}{4\lambda^2} + \sigma_N^2} Y_0. \end{aligned} \tag{7.2}$$

Therefore, using (7.2) in (7.1), we can solve for λ^*, that is

$$\begin{aligned} 0 &= \lambda^* Y_0^2 - Y_0 \mathbb{E}\left[S_0 + r_t \mid Y_0\right] \\ &= \lambda^* Y_0^2 - \frac{\frac{\sigma_S^2}{2\lambda}}{\frac{\sigma_S^2}{4\lambda^2} + \sigma_N^2} Y_0^2 \\ \implies \lambda^* &= \frac{\sigma_S}{2\sigma_N}. \end{aligned}$$

Intuitively, we can observe that the liquidity provider will change their price according to the volatility of the underlying asset price S and the randomness of the noisy trader. This means that the equilibrium price impact λ^* is increasing in the volatility of the asset value and decreasing in the volatility of noise trades. It is intuitive as the liquidity provider tries to see what the asset value is through total order flow, in which the insider's trade is 'masked' by the noise trade. This shows that the volatility of the underlying asset plays a crucial role in the decision making process of the liquidity provider and the prices they quote. Therefore, by forecasting increases or decreases in realized volatility a market maker is able to determine whether to widen or narrow their bid-ask spread.

Appendix B: Chen's Algorithm

In this section, we provide the backpropagation through time algorithm used in [12] to train a basic recurrent neural network designed for sequence labelling. That is, given an sequence of data points $\mathbf{x} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_T}\}$, the model maps each data point to a corresponding response variable $\hat{y} = \{\hat{y_1}, \hat{y_2}, ..., \hat{y_T}\}$. We have altered the notation to remain consistent with the rest of the paper. The aim is to minimise the following loss function:

$$\mathcal{L} = -\sum_t y_t \log(h(\mathbf{x^{(t)}})), \tag{7.3}$$

where,

$$h(\mathbf{x^{(t)}}) = \mathrm{softmax}(W_{hy}\mathbf{h}_t + b_y) \tag{7.4}$$

and

$$\mathbf{h}_t = \phi(W_{xh}\mathbf{x}_t + W_{hh}\mathbf{h}_{t-1} + b_h). \tag{7.5}$$

Chen derived the derivative of the loss function, with respect to the following 4 parameters, $\mathbf{W} = (W_{xh}, W_{hh}, W_{hy}, b_y)$.

We proceed by taking the partial derivative of $\mathcal{L}$ with respect to W_{hy}. As $\mathcal{L}$ is a function of $h(\mathbf{x^{(t)}}) = W_{hy}\mathbf{h}_t + b_y$, and each $h(\mathbf{x^{(t)}})$ is a function of W_{hy}, then via the chain rule we write:

$$\frac{\partial \mathcal{L}}{\partial W_{hy}} = \sum_t \frac{\partial \mathcal{L}}{\partial h(\mathbf{x^{(t)}})} \frac{\partial h(\mathbf{x^{(t)}})}{\partial W_{hy}}. \tag{7.6}$$

In the same manner, $h(\mathbf{x^{(t)}})$ is a function of b_y, therefore yielding:

$$\frac{\partial \mathcal{L}}{\partial b_y} = \sum_t \frac{\partial \mathcal{L}}{\partial h(\mathbf{x^{(t)}})} \frac{\partial h(\mathbf{x^{(t)}})}{\partial b_y}. \tag{7.7}$$

To derive the next partial derivatives we must consider individual components of the loss function at each time step. We define $\mathcal{L}(t)$ as the component of the loss function at time step, t. We have that $\mathcal{L}(t) = y_t \log(h(\mathbf{x^{(t)}}))$. Therefore to take the derivative with respect to W_{hh}, we compute the gradient with respect to $\mathbf{z}_t$ and then backpropagate from t-1 to 0 to calculate the gradient with respect to W_{hh}.

$$\frac{\partial \mathcal{L}(t)}{\partial W_{hh}} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}(t)}{\partial h(\mathbf{x^{(t)}})} \frac{\partial h(\mathbf{x^{(t)}})}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W_{hh}}. \tag{7.8}$$

By summing over all the time steps in the sequence we obtain the derivative of the loss function:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_t \sum_{k=1}^{t} \frac{\partial \mathcal{L}(t)}{\partial h(\mathbf{x^{(t)}})} \frac{\partial h(\mathbf{x^{(t)}})}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W_{hh}}. \tag{7.9}$$

In a similar manner, we obtain the partial derivative with respect to W_{xh}. We use the chain rule and backpropagate from t-1 to 0. However in this case, $\mathbf{h}_t$ is simultaneously a function of both W_{xh} and $\mathbf{h}_{t-1}$. As $\mathbf{h}_{t-1}$ is also a function of W_{xh}. We must consider each part separately,

$$\frac{\partial \mathcal{L}(t)}{\partial W_{xh}} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}(t)}{\partial h(\mathbf{x^{(t)}})} \frac{\partial h(\mathbf{x^{(t)}})}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W_{xh}}.$$

Again, summing over each time step yields the following:

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}(t)}{\partial h(\mathbf{x^{(t)}})} \frac{\partial h(\mathbf{x^{(t)}})}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W_{xh}}. \tag{7.10}$$

Bibliography

[1] Apple Inc. (AAPL) stock historical prices & data – Yahoo Finance. *Available: https://finance.yahoo.com/quote/AAPL/history.*

[2] Download Historical Apple (AAPL) Stock Price Data (20 Years Data). *Available: https://firstratedata.com/i/stock/AAPL.*

[3] Dropbox, AAPL Raw Data. *Available: https://www.dropbox.com/scl/fo/qe9c7fcbu021gtaphk60r/AHeNT6gR-86ZvPJamdqMNA8?rlkey=3v1e2204nik6gmgu6bp8ju73i&st=ou8ywz6z&dl=0.*

[4] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, and others. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *Available: http://tensorflow.org/*, 2015.

[5] Federico M. Bandi, Nicola Fusari, and Roberto Renò. 0DTE Option Pricing. *SSRN Electronic Journal*, 7 2023.

[6] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. Problem of learning long-term dependencies in recurrent networks. *1993 IEEE International Conference on Neural Networks*, pages 1183–1188, 1993.

[7] Colin Bennett. Trading Volatility, Correlation, Term Structure and Skew TRADING. 2014.

[8] Christopher M. Bishop. Pattern Recognition and Machine Learning. *Pattern Recognition and Machine Learning*, 12 2006.

[9] Ane Blázquez-García, Angel Conde, Usue Mori, and Jose A. Lozano. A Review on Outlier/Anomaly Detection in Time Series Data. *ACM Computing Surveys (CSUR)*, 54(3), 4 2021.

[10] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. Springer International Publishing, Cham, 2016.

[11] Andrea Bucci. Realized Volatility Forecasting with Neural Networks. *Journal of Financial Econometrics*, 18(3):502–531, 6 2020.

[12] Gang Chen. A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation. 10 2016.

[13] Francois and others Chollet. Keras . *GitHub - Available: https://github.com/fchollet/keras*, 2015.

[14] Violetta Dalla, Liudas Giraitis, and Peter C B Phillips. ROBUST TESTS FOR WHITE NOISE AND CROSS-CORRELATION Robust Tests for White Noise and Cross-Correlation *. 2019.

[15] Jefferson Duarte, Christopher S. Jones, Mehdi Khorram, and Haitao Mo. Too Good to Be True: Look-ahead Bias in Empirical Option Research. *SSRN Electronic Journal*, 10 2023.

[16] Julien Guyon and Jordan Lekeufack. Volatility is (mostly) path-dependent. *Quantitative Finance*, 23(9):1221–1258, 9 2023.

[17] James Douglas Hamilton. *Time Series Analysis*. Princeton University Press, 1994.

[18] Jiayu He, Matloob Khushi, Nguyen H. Tran, and Tongliang Liu. Robust dual recurrent neural networks for financial time series prediction. *SIAM International Conference on Data Mining, SDM 2021*, pages 747–755, 2021.

[19] Ian J. Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[20] Alan J. Izenman. *Modern Multivariate Statistical Techniques*. Springer New York, New York, NY, 2008.

[21] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. An Introduction to Statistical Learning. 2023.

[22] Georgios Karagiannis. Artificial Neural Networks Lecture Notes, Department of Mathematical Sciences, Durham University. *Available: https://github.com/georgios-stats/Machine_Learning_and_Neural_Networks_III_Epiphany_2024*, 2024.

[23] Georgios Karagiannis. Stochastic Gradient Descent Lecture Notes, Department of Mathematical Sciences, Durham University. *Available: https://github.com/georgios-stats/Machine_Learning_and_Neural_Networks_III_Epiphany_2024*, 2024.

[24] Diederik P Kingma and Jimmy Lei Ba. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. 2015.

[25] William Kollard. Python Programming. *Available: https://github.com/williamzkollard/Recurent-Neural-Networks-to-Forecast-Directional-Movement-of-a-Stocks-Realised-Volatility*, 2024.

[26] Albert S. Kyle. Continuous Auctions and Insider Trading. *Econometrica*, 53(6):1315, 11 1985.

[27] Henry A. Latane and Richard J. Rendleman. Standard Deviations of Stock Price Ratios Implied in Option Prices. *The Journal of Finance*, 31(2):369, 5 1976.

[28] Steven Manaster and Gary Koehler. The Calculation of Implied Variances from the Black-Scholes Model: A Note. *The Journal of Finance*, 37(1):227, 3 1982.

[29] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. MIT Press, 2 edition, 2018.

[30] Marc Peter, Deisenroth A Aldo, Faisal Cheng, and Soon Ong. MATHEMATICS FOR MACHINE LEARNING. 2020.

[31] Lihki Rubio, Adriana Palacio Pinedo, Adriana Mejía Castaño, and Filipe Ramos. Forecasting volatility by using wavelet transform, ARIMA and GARCH models. *Eurasian Economic Review*, 13(3-4):803–830, 12 2023.

[32] Sebastian Ruder. An overview of gradient descent optimization algorithms. 9 2016.

[33] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. ACTIVATION FUNCTIONS IN NEURAL NETWORKS. *International Journal of Engineering Applied Sciences and Technology*, 4:310–316, 2020.

[34] Guizhu Shen, Qingping Tan, Haoyu Zhang, Ping Zeng, and Jianjun Xu. Deep Learning with Gated Recurrent Unit Networks for Financial Sequence Predictions. *Procedia Computer Science*, 131:895–903, 1 2018.

[35] Tran Thanh Ngoc, Le Van Dai, and Dang Thi Phuc. Grid search of multilayer perceptron based on the walk-forward validation methodology. *International Journal of Electrical and Computer Engineering (IJECE)*, 11(2):1742–1751, 2021.

[36] Xin Wang, Jiabing Xu, Wei Shi, and Jiarui Liu. OGRU: An Optimized Gated Recurrent Unit Neural Network. *Journal of Physics: Conference Series*, 1325(1):012089, 10 2019.

[37] Shudong Yang, Xueying Yu, and Ying Zhou. LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example. 2020.

[38] Chao Zhang, Yihuang Zhang, Mihai Cucuringu, and Zhongmin Qian. Volatility Forecasting with Machine Learning and Intraday Commonality. 2023.

[39] Yu-Long Zhou, Ren-Jie Han, Qian Xu, and Wei-Ke Zhang. Long Short-Term Memory Networks for CSI300 Volatility Prediction with Baidu Search Volume. *Concurrency and Computation: Practice and Experience*, 31(10), 5 2018.