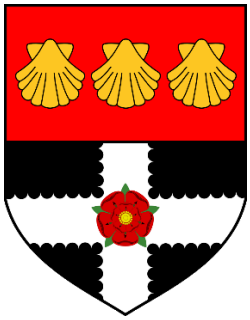

University of Reading

Department of Computer Science

A Study into the Effectiveness of Recurrent Neural Networks for Trading

Kyle Blue Doidge

Supervisor: Varun Ojha

A report submitted in partial fulfilment of the requirements of

the University of Reading for the degree of

Bachelor of Science in Computer Science

April 29, 2021

Declaration

I, Kyle Blue Doidge, of the Department of Computer Science, University of Reading, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

Kyle Blue Doidge

29/04/2021

Abstract

Manual trading comes with some pitfalls: poor trading discipline, stress, and dealing with misinformation only to name a few. There has been a rise in automated trading to combat these downsides and leverage machine learning. To contribute to this field, we created an RNN model to forecast the price direction of Bitcoin. In the process, we experimented with RNN architectures, dataset parameters and optimised model hyperparameters to create the most effective model whilst gaining insights that may contribute to the field. Our dataset only included 1-minute Bitcoin price and volume data, nothing more. Although hyperparameters such as number of neurons or hidden layers varied in testing, all models followed the same base structure which featured repeated RNN, dropout, and batch-normalisation layers. Through experimentation, we discovered forecast period was not most optimal when lowest, likely due to noise in markets at low timeframes. Addition of technical indicators in the dataset did not increase accuracy. GRU prevailed over LSTM and bidirectional variants did not result in increases in accuracy despite findings in existing literature. After some experimentation, we chose an optimal sequence length of 100, GRU architecture, and a 10-minute forecast period. We then optimised model hyperparameters using Genetic Algorithm to obtain a final model. Our final model achieved a directional accuracy of 54.8%. When only the top 20% of predictions were used, this increased to 59.2%. When tested in a simulated trading environment, the model gained 38.94% account balance in 565 trades and traded the correct direction 60.11% of the time.

Acknowledgements

An would first and most importantly like to thank my Supervisor, Varun Ojha, who was extremely knowledgeable and helpful throughout the duration of this project. Any queries were responded to promptly and valuable suggestions were provided to further the project.

I would also like the Computer Science department, the “CS Cluster” and Julian Kunkel (who maintained the cluster) for allowing me to make use of their system which included a powerful v100 16GB GPU, which was a drastic improvement over my laptop for Machine Learning.

Contents

Abbreviations

ANN: Artificial Neural Network

RNN: Recurrent Neural Network

CNN: Convolutional Neural Network

LSTM: Long Short Term Memory (this is an RNN architecture)

GRU: Gated Recurrent Unit (this is an RNN architecture)

MAE: Mean Absolute Error

GPU: Graphics Processing Unit

CPU: Central Processing Unit

RAM: Random Access Memory

VRAM: Video Random Access Memory

CFD: Contract For Difference (a type of contract brokers sell)

Chapter 1
Introduction

1.1 Background

Trading currencies and shares is known to be a complex activity, with few traders being able to provide consistent profits. Many retail traders (often referred to as individual traders) trade primarily using volume (amount traded in a given timeframe) and price history represented in a graphical chart format. Traders who approach market analysis in this way are called technical analysts [1]. The graphical format only abstracts the original numbers and prices in order to make it more digestible and presentable, however, it could be argued that these traders trade purely based on numerical price history. The majority of modern studies have found potential for profitability following this approach to trading [2]; however, the process is still highly convoluted. There is a large learning process involved in becoming a successful trader, however, a lot of the commonly distributed knowledge is largely unhelpful and often erroneous. On top of this, human traders can suffer from a variety of disciplinary issues (one may not stick to a predefined rule he previously made to protect his balance). This approach has many pitfalls; thus, a new approach is in order.

More recently, large trading firms have begun to make use of automated trading systems. Here, a computer will automate the analytical, and executional process of trading [3].

A rising automated trading approach (the approach detailed in this report) makes use of machine learning. Price history data is an example of time series data; each price quote is associated with a timestamp, and proceeds/precedes other price quotes. Recurrent Neural Networks (RNNs) are a class of Artificial Neural Networks (ANNs) that make use of time-series data to solve temporal problems, therefore price history data could be fed into an RNN with the aim of it learning to predict future price direction to automate the trading decision process. We can predict future direction through either creating a classification model which predicts whether the future price will be more (class 1) or less (class 2) than the current price, or creating a regression model, which would aim to predict future price. There are a number of RNN architectures currently available, each of which poses different benefits and drawbacks. Two popular RNN architectures that perform better than a simple RNN include GRU and LSTM [4] (both their unidirectional and bidirectional variants), both of which will be explored in this document.

1.2 Aims and objectives

There are a number of things we aim to achieve or discover throughout this study. We must ensure these aims are met by setting measurable objectives for each aim.

The primary aim of this study is to discover whether Recurrent Neural Networks can be used to accurately predict the future price direction of financial markets using only price and volume history. To explore this, we must create a Recurrent Neural Network models, and train it to predict future

prices direction via feeding it price history of a financial vehicle (we chose Bitcoin in this case). There are two potential models types we could create, a regression model (which would predict future price), and a classification model (which would predict future price direction). We will measure the accuracy for each of these models (using metrics such as R Square, Mean Absolute Error, Categorical Cross-Entropy and accuracy) on unseen test data, and place them in a simulated trading environment to test and demonstrate their real-life performance.

A secondary aim of this study is to discover how changes to the RNN architecture and how modifications to the dataset affect the predictive accuracy of the models. To explore this, we can train our model using consistent hyperparameters, only varying the RNN architecture used (between LSTM, GRU and their bidirectional variants), and measure changed in the accuracy metrics. Additionally, using the same approach, we can modify various parameters of our dataset, such as the sequence length and forecast period, and again measure the changes in the capabilities of the resultant models.

Our final secondary aim is to create a final model with near-optimal hyperparameters. This is often a lengthy and tedious process of trial and error [5], therefore, this is best done through the use of an optimisation algorithm such as Genetic Algorithm, thus to discover near-optimal hyperparameters, we will use Genetic Algorithm to tweak and discover the most optimal hyperparameter combinations.

1.3 Solution approach

1.3.1 Tools, Libraries and Frameworks

Python provides a variety of libraries to aid data science and machine learning. For this reason, Python has been used to accelerate the experimental process. TensorFlow is an open-source library developed and maintained by Google which is used for Machine Learning. TensorFlow is widely and thoroughly documented, thus it makes a sensible choice to create and test machine learning models for this project. NumPy and Pandas are adopted python data-manipulation/general-purpose libraries. These libraries are utilised in this project primarily for data pre-processing. Matplotlib has been used for plotting and visual analysis.

1.3.2 Methodology Overview

Firstly, an open-source dataset was obtained from Kaggle [6]. This dataset included prices and volume history from a range of cryptocurrencies pairs (1287 in total) present on a cryptocurrency exchange called Binance. Only Bitcoin was used within this project, though it is possible to experiment with different cryptocurrencies using the same models generated in this project.

The regression model would attempt to predict future price, while the classification model would attempt to predict future price direction. The dataset was pre-processed by standardising the data using z-scores (this makes more sense since our price data when converted to percent change fits a Gaussian distribution [7]) creating sequences with a fixed length, and adding targets to each sequence (future price) for the RNN to aim for.

The dataset was then split into a training set, a validation set, and a test set in a 60 20 20 split respectively. The training data was used to train a series of RNNs. Each of the RNNs followed the same base structure, while the number of hidden layers, neurons, hyperparameters, and the architecture (LSTM or GRU) varied throughout tests. Testing was conducted on the dataset and model architecture and intriguing results (and accuracy metrics) were noted. Later, a final model was created following results from previous testing and by optimising hyperparameters using Genetic Algorithm. The final model was tested on unseen test data and in a simulated trading environment.

1.4 Summary of contributions and achievements

Several interesting results were discovered in our experimentation which significantly contribute to the field of price forecasting using RNNs.

The most significant achievement from this study was the creation of our final model. The final model was created using near-optimal hyperparameters discovered through the use of Genetic Algorithm. This model managed to generated impressive profits in our simulated trading environment, which includes transaction fees and leverage. The final model traded in the correct direction 60.11% of the time in our simulated trading environment while existing literature that attempts to forecast bitcoin price direction struggles to get a classification accuracy of over 55% [8].

Additionally, we discovered that the optimal forecast period was not necessarily the lowest, and a large sequence length did not significantly increase accuracy. These findings subverted expectations. Our experimentation provided additional findings which are detailed in Chapter 5.

1.5 Organization of the report

This report is separated into several numbered chapters, each with its own subsections. Subsections are also numbered using the format:

[Chapter Number].[Subsection 1 Number].[Subsection 4 Number]

For example, the current section of the report (1.5) refers to the fifth subsection of the first chapter.

There are six main chapters overall. The first chapter (the current chapter) introduces the project and summarises the approach and results. The second chapter reviews existing literature and relates them to our project and experimentation. The third chapter gives an in-depth explanation of the methodology used to implement our models and conduct our study. The fourth chapter shows the results of our experimentation and provides some discussion and analysis. The fifth chapter summarises the findings and conclusions from the study and details some potential future work. The sixth and final section is a reflection on the learning experience this project elicited.

Additional headings that aren't referred to by chapter number include the abstract (at the beginning of the document), the references, and the appendices.

There are a number of graphs and graphic used throughout the report to guide explanations. Each of these is marked by figure numbers in the format:

[Chapter Number].[Figure Number]

The figure number of each graphic is displayed underneath the graphic (alongside a caption). Similarly, any tables displayed throughout this document are referred to by table number (with the same format). This table number (and a caption) is displayed above each associated table. Lastly, any code snippets (also known as listings) follow the same format. Captions including the listing number are provided below each code snippet.

Chapter 2
Literature Review

By analysing existing literature associated with this project, we can identify how we can create a solution that further augments existing research. Additionally, we can identify common results and findings which can be used for future comparison with our own findings.

2.1 Bitcoin Price Forecasting

There are a few existing studies in the literature that directly relate to our study; they attempt to forecast the future price direction of Bitcoin using Recurrent Neural Networks. The success of accurately broadcasting the future price direction of Bitcoin is varied. McNally et al [8] managed to achieve a classification accuracy (up or down) of 52% using an LSTM model. This is not entirely impressive, and while above 50%, the model would likely either break even, or consistently lose money when used in a real-life environment due to trading fees and commissions. Madan et al [9] managed to create a model which elicited 50-55% accuracy (for predicting price direction) when forecasting 20 minutes into the future; an accuracy marginally superior to that achieved by McNally et al. Madan et al additionally claims to have created a model which boasts 98.7% directional accuracy when predicting daily price change, though these claims are to be taken lightly. Although Madan made use of Blockchain data in addition to price data, such high accuracy is likely not possible using only this information; markets are too complex. Greaves and Au [10] took a slightly different approach in predicting the price direction of Bitcoin. While others used price and volume history as their primary data source, Greaves and Au made use of bitcoin transaction history (from the blockchain) only. This elicited 55% directional accuracy. While the use of Bitcoin blockchain data may seem beneficial for predicting future price, we aim to make our findings more generalisable to other financial markets, such as stocks or currencies, thus this will not be adopted in our project.

2.2 Data Pre-processing Approaches

There are differences across the literature for the way in which data was pre-processed before feeding it to the neural networks during training. Bodyanskiy et al [11] (among others) approached training of the RNN using only raw price data (open, high, low, close and volume). Raw price data can generally be very noisy, though using raw price data will ensure no data is obscured or lost. Hsu et al [12] hypothesized that predictive accuracy (and thus potential profits) is higher if a model incorporates

technical indicators[1] (often smoothed and derived data) into its dataset. Of the related existing literature Hsu reviewed, the majority made use of technical indicators in an attempt to improve predictive accuracy. Despite this, Hsu et al concluded through their own experimentation that the use of technical indicators did not significantly improve model accuracy. This contrasted the findings of Demir et al [13], who found that the addition of technical indicators drastically increased accuracy when forecasting electricity prices. Indicators may help in smoothing data (and filtering noise) which may in theory increase the model's ability to generalize, however, Boonprong et al proposed that the optimization of hyperparameters may increase generalization capabilities under heavy noise [14]. For example, having too many neurons and hidden layers, for example, could decrease generalization capability, and potentially lead to overfitting since the ANN may learn complex relationships that don't model the data correctly. This is best represented in a diagram:

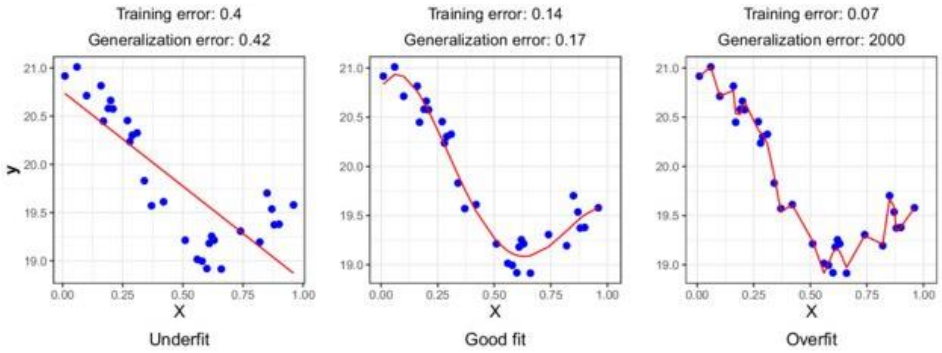

Figure 2.1 - Visual representation of underfitting, overfitting, and a good fit [15]

It is clear that there is fierce debate in the literature regarding the predictive ability of technical indicators. Advocates stress the value in the use of technical indicators [16] [17], while opposition refutes its predictive value [18] [19]. In this situation, it is most sensible for us to test model accuracy when trained with and without technical indicators hereby drawing our own conclusions on this controversial topic.

2.3 Machine Learning Approaches to Price Forecasting

A variety of deep learning methodologies have been employed in a number of areas. Autoregressive Integrated Moving Average (ARIMA) models are popular in literature for forecasting future market prices, however, McNally et al found the predictive accuracy or ARIMA models to be significantly worse than more modern deep learning architectures such as LSTM when applied to Bitcoin price forecasting (despite the drastically increased learning time) [8]. In their experimentation, the LSTM model achieved a Root Mean Squared Error (RMSE) of 6.87%, while the ARIMA model achieved an RMSE of 53.74% (lower is better). Because of its ability to deal with complex relationships and operate on large sequential data, recurrent neural network architectures such as GRU and LSTM are increasingly used to develop forecasting models [20].

GRU is a more recent recurrent neural network architecture that rivals the predictive accuracy of LSTM architecture whilst simultaneously having a simplified neuron/memory-cell structure [21]. Yang et al

[1] A technical indicator is a mathematical calculation derived from raw price data which aims to forecast future market direction. Technical indicators are often used by retail traders when conducting manual market analysis.

found simpler structure provided by GRU would also translate into a faster training process (around 29.29% faster on the same dataset) [22]. Zhou et al proposes a Recurrent Neural network architecture called Minimal Gated Unit (MGU) which again boasts a simpler structure (than both GRU and LSTM), and thus faster training times, whilst maintaining similar accuracy on large sequential data [23]. Despite this, popularity and thus documentation is lacking, making this a poor candidate for this project.

From the aforementioned literature, we can surmise that the use of Recurrent Neural Network architectures such as GRU and LSTM are among the most effective current machine learning methodologies for time series forecasting, however, which architecture do we expect will result in better predictive accuracy? While in the majority of problems, GRU and LSTM can produce similar results, Gao et al [21] argues that the use of LSTM results in better accuracy when dealing with larger sequential data. This is due to the larger number of gates per memory cell in LSTM [24].

The predictive accuracy of GRU and LSTM models often vary for the unidirectional and bidirectional variants. Studies have concluded for a majority of applications bidirectional variants lead to higher accuracy than their unidirectional counterparts [25]. This is likely because in longer sequences unidirectional models have to make a trade-off between "remembering" past input information and "knowledge combining" new information with input information already processed [26]. Additionally, Schuster et al [26] claims that despite having double the neurons/memory-cells (one set connected in the forward direction, and another independent set connected in the backward direction), bidirectional RNNs learn at a similar rate to conventional unidirectional RNNs.

2.4 Critique of the Review

The existing literature provides some important pointers that we can use to guide this project, and optimize the accuracy of our resultant machine learning model.

Since the literature behind the predictive utility of technical indicators is mixed, it is important for us to conduct our own experimentation. While we could theoretically draw conclusions by comparing accuracy metrics from opposing studies, comparisons are complicated via the use of different accuracy metrics. As an example, Bodyanskiy et al [11] measured predictive accuracy using NMAE (normalised MAE) and NMSE (normalised MSE) while Hsu et al [12] measured predictive accuracy using R^2 (among other metrics).

One criticism we can draw from the existing literature is that the majority tend to lack comprehensive and practical (real-world) testing. While studies such as those conducted by Bodyanskiy et al [11] and Greaves et al [10] provided a series of metrics illustrating the predictive capabilities of the generated models, it is not immediately clear by reading these metrics how this would translate into real-life performance and profits (which I estimate is a major motivator behind a significant amount of these projects). We intend to tackle this problem by providing the results of simulating trading on the unseen test dataset.

Many of the studies mentioned here cease to detail the experimental process involved in obtaining optimal hyperparameters for their model. It is unclear how chosen hyperparameters were obtained in related literature. Optimising model hyperparameters using an optimisation algorithm such as Genetic Algorithm may have resulted in more optimal hyperparameters and thus better results in the

detailed studies. Following this line of reasoning, we will make use of Genetic Algorithm for hyperparameter optimisation.

2.5 Summary

There were a number of important discoveries and findings in existing literature that influence our project. Firstly, there are a number of approaches to forecasting Bitcoin prices specifically. Some studies make use of Bitcoins blockchain transactions as a means of forecasting price [10], while others simply make use of price alone [8]. Both methods obtain similar accuracies (around 52-55%). It seems that forecasting longer-term prices (such as daily prices) may increase accuracy [9], but the validity of these findings is in question.

There is hot debate in literature about the effectiveness of technical indicators in forecasting future price movements. Hsu et al concluded they did not increase accuracy, while Demir et al [13] concluded that the use of technical indicators lead to an increase in accuracy.

There were a number of different machine learning approaches to forecasting future prices, such as the use of ARIMA and RNN models such as LSTM and GRU. RNN models tended to outperform ARIMA [8], which was a popular machine learning approach in the literature for time-series forecasting.

Literature generally agrees that the use of bidirectional RNN variants may lead to an increase in accuracy [25]. Moreover, LSTM may likely outperform GRU when larger sequence length are used [21].

Chapter 3
Methodology

3.1 Problems (tasks) description

Ultimately, this experimental process aims to explore whether or not an automated trading process using an RNN model and price/volume history can be used as a profitable replacement for manual trading. In the process, we discuss whether our classification model or our regression model are more likely to translate into profitable trading. Additionally, we investigate the differences in accuracy between the LSTM and GRU architectures (and bidirectional variants) for this particular problem. Moreover, we find out how the use of technical indicators, the dataset sequence length and the forecast period affect resultant model statistics. Lastly, we find near-optimal hyperparameters for our model through the use of Genetic Algorithm.

3.1.1 Overview of used Technologies and Tools

Language: Python3

Libraries / Technologies:

- TensorFlow
- Keras
- NumPy
- Pandas
- Matplotlib

Hardware:

- **RAM:** 128GiB
- **CPU:** 2x Intel Xeon Silver 4108 CPU @ 1.80GHz – 8 core – 16 thread
- **GPU:** Tesla V100-PCIE 16GB

3.1.2 Justification of used Technologies and Tools

Though we have briefly looked at the technologies used, we will discuss them here in more detail.

When conducting experiments in machine learning, it is important to have the ability to make swift changes to a range of aspects of the machine learning model such as the architecture, the optimisation function and the hyperparameters. Manually developing each feature would drastically increase the amount of time taken to conduct the experiment, therefore it is helpful to make use of existing technologies. Python provides an array of technologies and libraries which have been made to simplify and accelerate machine learning development such as TensorFlow and PyTorch. For this reason, Python was the chosen language for this experiment.

TensorFlow was chosen over PyTorch due to its popularity and extensive documentation. TensorFlow now comes installed with a library called Keras, which further simplifies the development process in aspects of model creation.

NumPy and Pandas are libraries that are commonly used alongside one another in Python. Both libraries attempt to simplify the data manipulation process. Data manipulation is imperative in the data pre-processing stage, thus explaining the adoption of these two libraries in this project. Pandas specialises in two-dimensional array structures while NumPy specialises in more complex array structures (also known as matrices or tensors). Additionally, NumPy is written in C (and dynamically linked to Python) thus increasing execution speed when manipulating data (which is known to be a computationally expensive task).

Matplotlib provides a simple interface for charting in Python, which will be useful in displaying the results of our experimentation.

3.2 Main Implementation

3.2.1 Data Retrieval

As previously mentioned, the dataset was downloaded from Kaggle [6]. The dataset includes price and volume history for Bitcoin. Due to the large size of the dataset, it was compressed in a parquet file. Loading from a parquet file is made easy by pandas (`pd.read_parquet()`). The dataset includes price data from 17/08/2017 to 11/02/2021 and includes 1828151 overall data points with each data point representing 1 minute of price action. Almost 2 million data points was considered too much to feasibly train the data on. Additionally, the Adaptive Markets Hypothesis states older data points are often less relevant in markets [27]; markets highly competitive and adaptive. They change and develop. Because of this, only the most recent 100,000 data points were used.

3.2.2 Data Pre-Processing

It is important to first note that Recurrent Neural Networks require training data in the form of a number of sequences with an accompanied target. Simply loading a raw time-series dataset, and using this to train our RNN will not work. The data must undergo some pre-processing and transformation in order for it to both be in an acceptable format and useful for our RNN.

When pre-processing continuous time-series data (such as price history) there are a number of common steps taken in pre-processing data. All common pre-processing steps were encapsulated within a “DataPreprocessor” class, simplifying the process of pre-processing further datasets. The common data pre-processing steps are represented in the flow chart below:

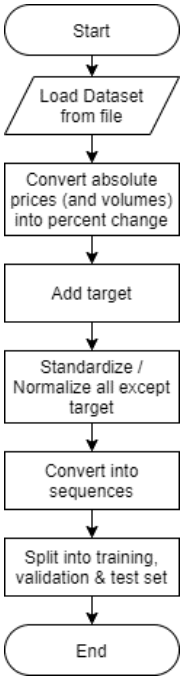

Figure 3.1 - Flow chart showing the data pre-processing procedure

First, our DataPreprocessor class loads the price and volume history into a DataFrame. A DataFrame is a multi-type 2D array type provided by the Pandas data manipulation library. Input data normalisation before training leads to significantly improved accuracy (and training speed) when training an artificial neural network [28], thus before creating sequences from our loaded data, it must be transformed and normalised.

Before normalisation, all prices must first be converted into percent change. In other words, each absolute price value must be replaced with the amount it has increased or decreased (as a percentage) since the previous price. Prices in markets rarely remain the same; current prices may drastically differ from prior year's prices, and training using completely different (old) prices values will not be useful for future price values. Converting each price into percentage change will make old price values useful for future prediction. After converting prices into percent change, we can proceed to adding the target value.

The target value for the regression model and the classification model differs though they are based on the same premise. Both require calculation of the future price value (again, in percent change). The future price value is dependent on the forecast period. Should we want to forecast the next price value, the future price value is simply the next price, however, should we want to forecast the price after 10 steps (forecast period of 10), we must compile the percentage changes of the next 10 prices, and use this as the target. We compile the percentages using the following formula:

(percentages are portrayed as decimals between 0 and 1 in our code)

```
combined_pct = 1
for x in values[i + 1:i + self.forecast_period + 1]:
    combined_pct *= (1 + x)
combined_pct -= 1
```

Listing 3.1 - Code showing how percentages were compiles for sequence targets

For our regression model, this forecasted percentage is the target, however, for the classification model, this target is 1 if the forecasted percent change is positive, and 0 if negative. After adding the target price for the ANN to aim for, we can normalise the data. Common normalisation techniques include MinMax normalisation and Z-Score normalisation. MinMax normalisation involved scaling all data linearly between 0 and 1 whilst Z-Score normalisation involves scaling the data such that the mean of the data becomes 0 and the standard deviation becomes 1. Since the term "normalisation" often refers to scaling between two limits, Z-Score normalisation is better known as "standardisation". The formula for converting a raw value into a standardised value is as follows (where μ is the mean of the data and σ is the standard deviation of the data):

$$standardised\ value\ = \frac{value\ -\ \mu}{\sigma}$$

Standardisation is more suitable for data that assumes a Gaussian (normal) distribution [7], such as our price change data (Figure 3.2), thus we will choose to standardise our data over normalising our data (using Min-Max Normalisation).

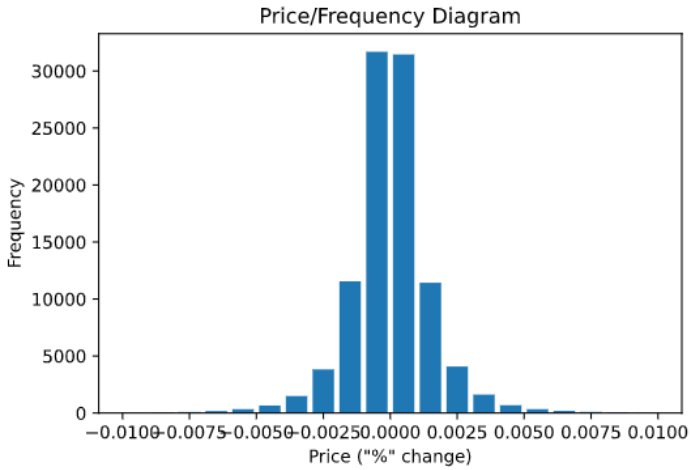

Figure 3.2 - Price % change/frequency diagram of Bitcoin showing Gaussian distribution

It is important that we avoid standardising the target, since a Z-Score prediction is not inherently useful (to a trader), and it is much more cumbersome to convert a Z-Score to a price.

After standardising our data, we convert the data into an array of sequences of a defined length. Each single data point in a sequence includes the "open", "high", "low", "close" and "volume" (standardised) values (and a sequence includes a number of these data points). The "target" value is separated out of the sequence into a separate array (this array is also known as the sequence labels). Each value in this array corresponds to the sequence at the same index.

These sequences (and labels) are further split into 3 separate sets: the training set, the validation set, and the test set, in a 60/20/20 split. The training and validation set are further shuffled together (randomising their order) in order to reduce order effects on training. Shuffling training and validation sets has been made common practice due to the statistical gains it promotes in model accuracy [29]. The test set was left unshuffled since data would typically appear in chronological order in a lifelike scenario.

3.2.3 Creating the Model

Since we are experimenting with various hyperparameters and architectures, it is important to make the manipulation of these parameters simple, therefore a "Model" class was created (full code in Appendix 2.3). The constructor takes a number of parameters, each of which modify the underlying TensorFlow model in some manor, allowing us to easily manipulate various model parameters:

```
class Model():
    def __init__(self, train_x, train_y,
        validation_x , validation_y, seq_info:str, *,
        max_epochs = 100, batch_size = 1024, hidden_layers = 2,
        neurons_per_layer = 64, architecture = Architecture.LSTM.value,
        dropout = 0.1, is_bidirectional = False, initial_learn_rate = 0.001,
        early_stop_patience = 6, is_classification=False):
        ...
        ...
```

Listing 3.2 - Custom Model class constructor

Although hyperparameters can be modified, there is a base structure to our machine learning model (Figure 3.3):

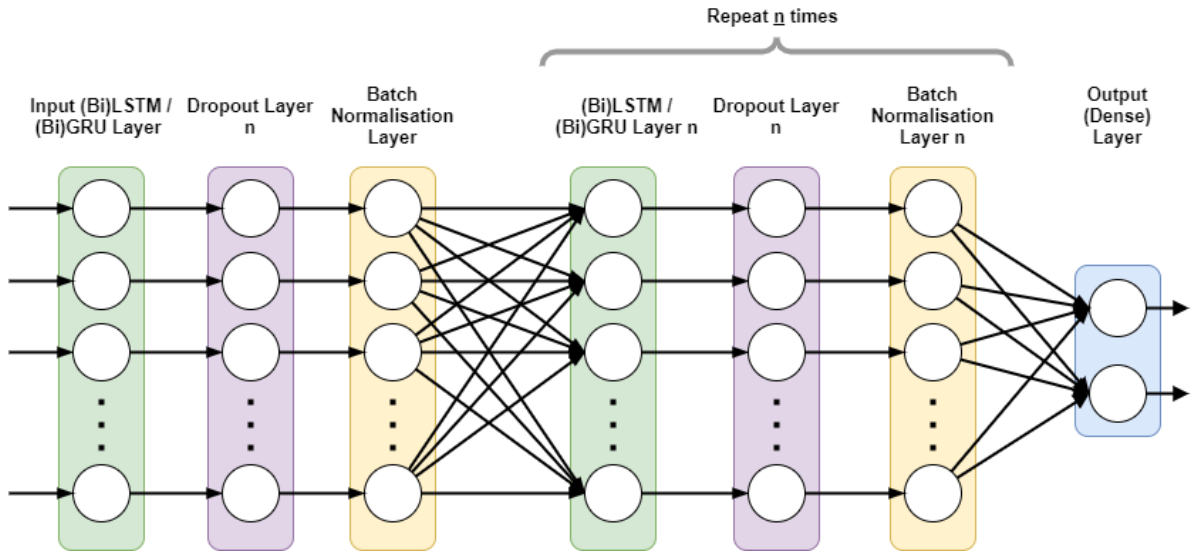

Figure 3.3 - The Recurrent Neural Network model structure used (classification model)

While the number of layers, the number of neurons, and the RNN architecture may vary, our Recurrent Neural Network will follow the above structure (regardless of the hyperparameters).

In the above diagram, the notation (Bi) means the RNN layer could potentially be bidirectional. A bidirectional RNN architecture has two independent sets of neurons, one set (theoretically) facing one direction, and one set facing another. One set of neurons pass the sequence in the traditional forward order, while the other set passes the sequences in the backward direction. This aims to mitigate the effects of the exploding and vanishing gradient problem present in basic RNNs.

You may notice that after every (Bi)LSTM/(Bi)GRU layer exists a dropout layer. Dropout is a simple regularisation technique first proposed by Srivastava et al [30] to address the overfitting problem. Regularisation techniques work by limiting the capacity of machine learning models to increase

generalisation capabilities (and reduce overfitting). Dropout works by randomly ignoring (disabling) a chosen percentage of neuron connections in the previous layer. This reduces the model's ability to learn complex relationships, thereby reducing the risk of overfitting.

After each dropout layer exists a batch normalisation layer. Despite its name, the batch normalisation layers standardise neuron outputs between layers in our deep neural network. This has the effect of preventing exploding and vanishing gradients (large error gradients that result in large updates to the ANN making it hard to learn) and keeping activations away from their saturation regions [31]. Both of these lead to a more stable training process and a more accurate model.

It is worth noting that Figure 3.3 displays the base RNN structure for our classification model. The classification model includes 2 neurons in the output layer, meaning there are two output values. The first output value pertains to the confidence that future price will decrease, while the second output value refers to the confidence that future prices will increase. This layer makes use of a sigmoidal activation function, bounding our output values between 0 and 1. Our regression model is completely identical to the classification model with the exception that there is only a single neuron in the output layer, meaning there is only a single output value. Additionally, the output value has no limits, since the activation function is linear. In code, model structure this is represented like so:

```
##### Create the model ####
self.model = Sequential()

if self.is_bidirectional:

self.model.add(Bidirectional(self.architecture(self.neurons_per_layer,
input_shape=(self.train_x.shape[1:]), return_sequences=True)))
else:
    self.model.add(self.architecture(self.neurons_per_layer,
input_shape=(self.train_x.shape[1:]), return_sequences=True))

self.model.add(Dropout(self.dropout))
self.model.add(BatchNormalization())

for i in range(self.hidden_layers):
    return_sequences = i != self.hidden_layers - 1 # False on last iter
    if self.is_bidirectional:

self.model.add(Bidirectional(self.architecture(self.neurons_per_layer,
return_sequences=return_sequences)))
    else:
        self.model.add(self.architecture(self.neurons_per_layer,
return_sequences=return_sequences))
    self.model.add(Dropout(self.dropout))
    self.model.add(BatchNormalization())

if self.is_classification:
    self.model.add(Dense(2, activation="sigmoid"))
else:
    self.model.add(Dense(1))
```

Listing 3.3 - Code showing implementation of model structure

Each call to the `model.add()` (TensorFlow) function adds the passed layer to the model.

Upon creation of a model, we detect whether or not a graphics card exists on the system. If a graphics card exists, we use the CuDNN variants of LSTM and GRU layers. This allows us to drastically accelerate the learning process by taking advantage of the superior parallelism capabilities of graphics processing units.

The classification model makes use of two metrics, accuracy (the percentage of correct classifications) and (sparse) Categorical Cross-Entropy. Categorical Cross-Entropy measures the distance between prediction vectors and the labels. The higher the distance, the worse the predictions. This means Categorical Cross-Entropy also takes into account the extent to which the classifications were correct (or how confident the RNN during a classification). While the accuracy metric is easier to comprehend and interpret, Categorical Cross-Entropy is a more complete classification metric, therefore it is also used as the loss function. The regression model on the other hand makes use of two common regression metrics: Mean Absolute Error (MAE) and R Squared. Mean Absolute Error is self-explanatory; it is the mean of the (absolute) errors (actual – predicted). This is used as the loss function. R Square measures the proportion of the variance in the dependent variable (price) that is explained by the input data in our model. An R Squared score below zero means our model was unable to fit the data well while a score of 1 means the model fits perfectly.

Adam optimiser makes use of advanced momentum techniques and adaptive learning rates in minimise model loss. It is widely known to be among the best performing optimisers (on average), and thus it retains high status in machine learning practice [32]. Because of this, the Adam optimiser was adopted into our RNN model.

During the training of our models, we made use of early stopping to reduce the chance of overfitting [33]. An early stopping patience of 6 epochs was used to account for potential outliers in validation loss.

Tensorboard

Though a GUI is not completely necessary for this experiment, we have adopted the use of TensorBoard: a visualisation toolkit that links directly to TensorFlow. This allows us to monitor training and validation loss (and additional metrics) through dynamic line charts during training.

3.3 Experimentation and Simulation

3.3.1 Technical Indicators

As previously mentioned, a technical indicator is a calculation made on raw price data. The calculated values are often used by traders to help forecast future price direction and to help understand market sentiment.

There are a vast number of technical indicators traders commonly used. A technical indicator library (Python TA) was used to bypass having to manually create these calculations. The library included 123 of the most popular indicators. Running our tests with all 123 indicators would drastically limit the range of hyperparameters we could feasibly use for our model, and dramatically increase training times. Because of this, we decided to reduce the count of indicators. To capture most of the variability in the indicator values, we reduced the indicators to the 15 indicators with the least collective correlations.

This was done by first calculating the correlations between all the indicators. Then, the lowest absolute correlation value is found (closest to 0). These two indicators are added to the list of indicators.

We then find out which indicator has the lowest collective correlations with our current list of indicators. This then added to the list of indicators. This process repeats until we have 15 indicators in the list. This process can be seen in greater depth in the code shown in Appendix 2.4.

This correlation reduction method produced the following correlation matrix/heatmap:

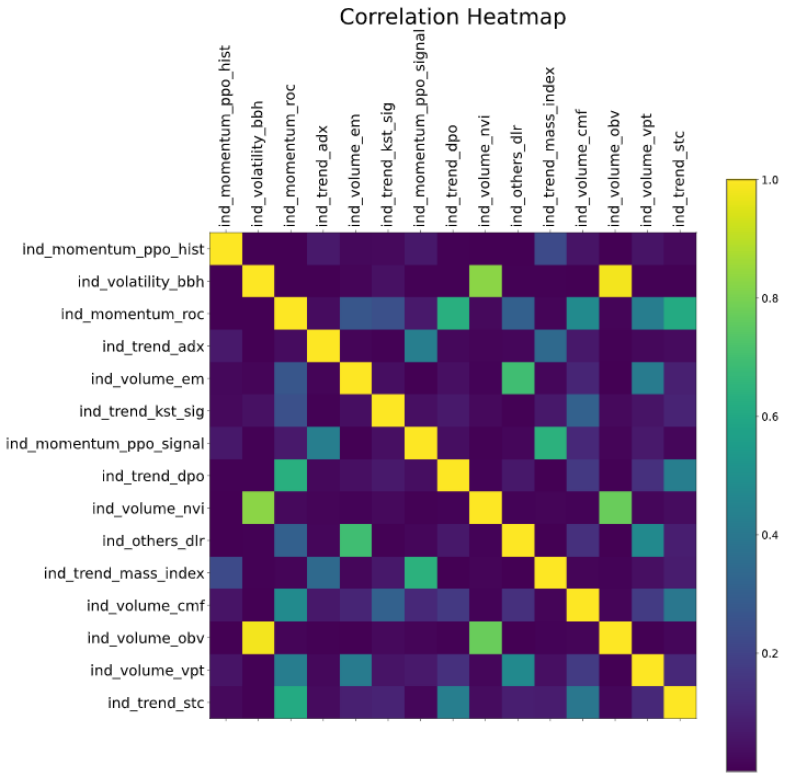

Figure 3.4 - Correlation Heatmap after Reducing Indicator Correlations

The majority of correlations now reside around zero, implying correlation reduction was successful.

3.3.2 Testing Architecture and Dataset Modification

Training an RNN model can take significant amounts of time (depending on the hyperparameters and dataset), thus, creating numerous models in the name of experimentation would take significant amounts of time; there is no way around this. Additionally, having to manually stop and start training more new models would require a huge amount of engagement with the computer. Because of this, we decided to automatically generate new models based on pre-set parameters in a loop.

To do this in TensorFlow, we must clear the old "Session" and create a new "Session" when creating each new model, otherwise the previous model in not cleared from memory, and a memory error will be thrown when hardware inevitably runs out of GPU memory.

Each of our tests had base parameters; parameters that would be used for every test. Base parameters were captured in a dictionary:

```
base_params = {
    "architecture": Architecture.GRU.value,
    "is_bidirectional": False,
```

```
    "indicators": False,
    "sequence_length": 200,
    "forecast_period": 10,
}
```

Listing 3.4 - Base dataset parameters and architecture

In each test, a select few of these parameters may be modified. To go about this, parameters to be modified would be captured in a dictionary:

```
tests = [
    # Format:
    # (num_repetitions, {"param to change": new_value})
    (5, {"architecture": Architecture.LSTM.value, "is_bidirectional": False}),
    (5, {"architecture": Architecture.GRU.value, "is_bidirectional": False}),
    (5, {"architecture": Architecture.LSTM.value, "is_bidirectional": True}),
    (5, {"architecture": Architecture.GRU.value, "is_bidirectional": True}),
    (5, {"indicators": False}),
    (5, {"indicators": True}),
    (1, {"sequence_length": 50}),
    (1, {"sequence_length": 100}),
    (1, {"sequence_length": 150}),
    (1, {"sequence_length": 200}),
    (1, {"sequence_length": 250}),
    (1, {"sequence_length": 300}),
    (1, {"sequence_length": 350}),
    (1, {"sequence_length": 400}),
    (1, {"forecast_period": 1}),
    (1, {"forecast_period": 5}),
    (1, {"forecast_period": 10}),
    (1, {"forecast_period": 20}),
    (1, {"forecast_period": 30}),
    (1, {"forecast_period": 50}),
]
```

Listing 3.5 - Code showing data structure used to automate experimentation on dataset and architecture

Then, during each loop (one loop represents one test) we would combine the base parameters and the modifications:

```
for test_num, test in enumerate(tests):
    additional_params = test[1]
    new_params = {**base_params, **additional_params} ## Combine parameters
    ...
```

Listing 3.6 - Code used to loop and replace base parameters with additional parameters used in testing

The combined parameters would then be used to create the new model. Results for each test were stored in a CSV file for later analysis.

3.3.3 Hyperparameter Optimisation using Genetic Algorithm

Genetic Algorithm is an optimisation algorithm based on Darwin's Evolutionary theory widely known as "Survival of the Fittest" [34]. Parts of its implementation can vary based on preference, though the general concept remains the same.

Firstly, a population of individuals are created. Each individual has a "chromosome"; a collection of genes. Each gene in the chromosome has a value (in our case, it is numerical). Changing the value of

a gene would modify a certain behaviour of the individual, therefore the chromosome values define the individual's behaviour.

After a population of individuals is created, their initial "fitnesses" must be evaluated. The way in which fitnesses are calculated differ by application.

Next, parents are selected for recombination. There are a number of different selection methods, though Rank-Based selection was used in our experimentation. In Rank-Based selection, each individual is first given a rank based on its fitness rating, with the individual with the highest fitness having a rank of 1. We then pick two weighted random individuals using *1 / rank* as their weights. These two individuals are now a parent pair. We continuously select in this manner until all individuals have been selected as parents.

Next, there is a strong chance (defined by the crossover rate) parents recombine and crossover their chromosomes to make two new individuals; their children. There are a number of crossover methods. One of the simplest methods, Uniform Crossover, was used here. Uniform Crossover is able to distribute parent features more evenly than other selection methods in an unbiased manner [35]. In Uniform Crossover, we recombine by first selecting at random one of the parents. Child one then gets the first gene value of the parent we picked, while child two gets the gene value of the other parent. This process repeats for every gene.

These children then undergo some mutation; there is a chance (this is defined by the mutation rate) for each gene value to be set to a random number (within pre-defined limits). This adds some variation to the population.

These children make up the next population, also known as a generation. Next, the fitnesses of these children must be calculated, and the process repeats. Over time, fittest individuals combine their best features causing increases in fitness over generations, making the "behaviour" of individuals more effective at solving the defined problem.

Some genetic algorithms, such as our implementation, make use of a concept called "elitism". This is where a number of the fittest individuals get a free pass to the next generation, despite not being offspring. In our experimentation, we only picked two elite individuals, since too many can lead to premature convergence [36]. The elite can still be selected as parents.

In our experiment, we used genetic algorithm to select optimal hyperparameters, therefore each gene value of an individual referred to a hyperparameter value of a model. To ensure hyperparameter values were sensible, and conformed to the capabilities of the system, there where upper and lower limits to each hyperparameter:

```
## Limits are inclusive
limits = {
    "hidden_layers": Limit(1, 4),
    "neurons_per_layer": Limit(16, 128),
    "dropout": Limit(0.0, 0.5),
    "initial_learn_rate": Limit(0.000001, 0.1),
    "batch_size": Limit(50, 2000),
}
```

Listing 3.7 - Hyperparameter limits for Genetic Algorithm

These limits were identical for both the regression and the classification models.

Fitness was calculated by training a model with the hyperparameters specified in a chromosome and evaluating its performance using its validation metrics. For the classification model, fitness was

$$-1 * Sparse\ Categorical\ Cross\ Entropy$$

and for the regression model fitness was the R Square score.

For the genetic algorithm hyperparameters, we selected a population size of 10, mutation rate of 0.2, crossover rate of 0.9, and run it for 15 generations. Genetic algorithm is often more effective at lower mutation rates, however, elitist genetic algorithms tend to have significantly higher optimal mutation rates [37].

3.3.4 Trading Simulation

A trading simulation was created to test the effectiveness of the final models produced in real-life scenarios. This was something that was not present in the existing literature.

The trading simulation would begin with a starting balance of £10,000. It would then test each predicted value against each actual value in unseen test data. If the direction of the predicted value conformed with the actual market direction, the percentage change in that forecast period would be added to the account, otherwise it would be subtracted. To enhance realism, commission[2] structures and leverage[3] were added to the simulation, thus, each "trade" would result in the subtraction of a small amount of money despite the success of the trade.

The leverage provided in the simulation was 2x, while commissions were placed at 0.0122% of trade value, which is around the area provided by some CFD brokers that allow trading of Bitcoin.

Lastly, if a trade were to hypothetically still be in progress, another trade cannot be made. As an example, if the forecast period of the model was 10 minutes, a single trade can only be made (maximum) over each 10-minute period.

Code implementing the trading simulation can be seen in Appendix 1.1 (regression model) and 1.2 (classification model).

3.4 Summary

In order to train a Recurrent Neural Network, we had to feed it a dataset. For our dataset, we used Bitcoin price history for 1-minute intervals. This was obtained from Kaggle. We then pre-processed the dataset by adding converting prices into percentage change, adding the target, standardizing the data, converting the data into sequences, and finally splitting those sequences into a training, a validation and a test set (using a 60/20/20 ratio).

This data was to be used to train and test a series of RNN models. Our models all followed a base structure. Generally, the model consisted of a (potentially bidirectional) LSTM/GRU layer followed by

---

[2] Commissions – Trading fees that the broker charges per trade. Some brokers just charge the spread – the difference between buying and selling price

[3] Leverage – Additional money brokers offer to clients to increase buying power, thus increasing size of both wins and losses. A leverage of 10x would increase a traders buying power 10-fold, allowing them to trade 10 times their deposited account balance

a Dropout layer, then followed by a Batch Normalisation layer. The number of hidden layers and neurons per layer were variable. The output layer was the only layer that differed between our classification and regression models. Classification models had 2 neurons in the output layer, while regression models had only one neuron in the output layer.

The technical indicator library we used provided 123 of the most popular technical indicators, however, we reduced this to just 15 by minimising the correlations between them; 123 was too much to feasibly test.

Testing the effects of modifications to the dataset and architecture was made simpler by looping the experiments. To do so error-free, TensorFlow sessions had to be cleared and recreated on each iteration.

Lastly, Genetic Algorithm was used to optimise model hyperparameters. Our implementation made use of Uniform Crossover, Rank-based Selection, and elitism. Hyperparameters included a mutation rate of 0.2, a crossover rate of 0.9, 2 elite individuals per generation, 15 generations and a population size of 10.

Chapter 4
Results and Discussion

Since we experimented with both regression models and classification models, results for each are isolated into their own sub-section.

As previously mentioned, results for the regression model were less significant than results for the classification model; predictions were not as accurate. Results for the regression model will not be explored in the same depth as there is less to learn from doing so. Full results are still provided in Appendix Chapter 1 (these results pertain to the experimentation on the dataset and RNN architecture).

4.1 Regression Model

4.1.1 Hyperparameter Optimisation using Genetic Algorithm

Previous experimentation on RNN architecture and dataset (sequence length and forecast period) showed insignificant differences in resultant (validation) accuracy, however, training times drastically differed, thus favourable training times had a stronger impact on architecture and dataset parameters used for our final regression model (since hyperparameter optimisation generally consume a large amount of time). Following this line of reasoning, we settled on a sequence length of 100, a forecast period of 10, and a unidirectional GRU architecture.

The number of hidden layers, the number of neurons per layer, the batch size, the dropout, and the initial learn rate were all continuously modified by Genetic Algorithm in the hopes of continuously improving fitness (which is the R Square value) and finding near-optimal hyperparameters.

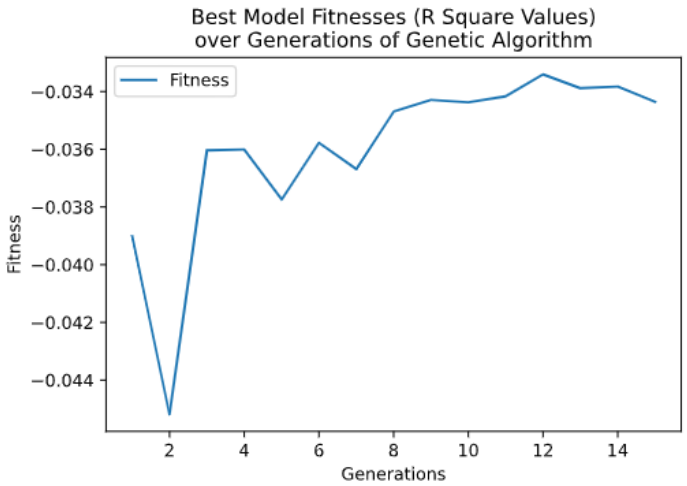

Figure 4.1 - Best Model Fitness over Generations of Genetic Algorithm for Regression Model

From Figure 4.1 we can see that fitness was fairly stable over generations, and it seems as if fitness is generally improving. Despite this, improvements are fairly minimal, and this stability may be the

result of the model's inability to learn beyond a certain point. Additionally, the R Square value fails to reach a point higher than 0, meaning the model fits the problem poorly. The reasoning for these results is made clearer when we look at predicted vs actual prices (Figure 4.2).

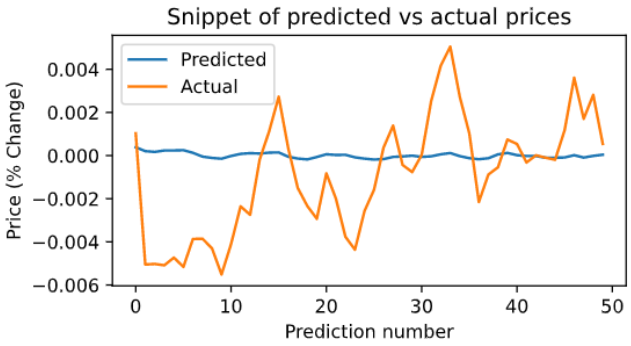

Figure 4.2 - A Snippet of Predicted vs Actual Prices for Regression Model

Figure 4.2 shows us that predicted prices tend towards 0, the mean price (in % change). In all predictions, our model is not particularly biased in any sort of direction, making its utility fairly futile in real-life scenarios. This suggests our RNN underfitted our data. Similar results were found in a number of the studies that implemented regression models for predicting market prices, such as those found by Khan et al [38]. Efforts were made to solve this problem, such as artificially increasing the variability of the data, and increasing the number of neurons / hidden layers. Despite efforts, no increase in predictive accuracy was seen.

4.1.2 Final Regression Model Results

Regardless of its unsatisfactory predictive abilities, we decided to continue with experimentation on unseen test data with the expectation directional accuracy may still prove useful. When tested on unseen test data, we saw an R Square score of 0.00026, and a Mean Absolute Error of 0.00334. Directional accuracy was 50.3%, which is not far above random directional prediction.

When only looking at the top 20% of predictions (the most extreme values) accuracy and R Square values increase to 51.6% and 0.00106 respectively, while MAE increased to 0.00358. An increase in MAE is to be expected here since MAE is relative to the variance/deviation in data [39] (and we can expect higher variance in data when looking at extremes).

The top 20% of predictions were placed in a simulated trading environment to test potential real-life performance.

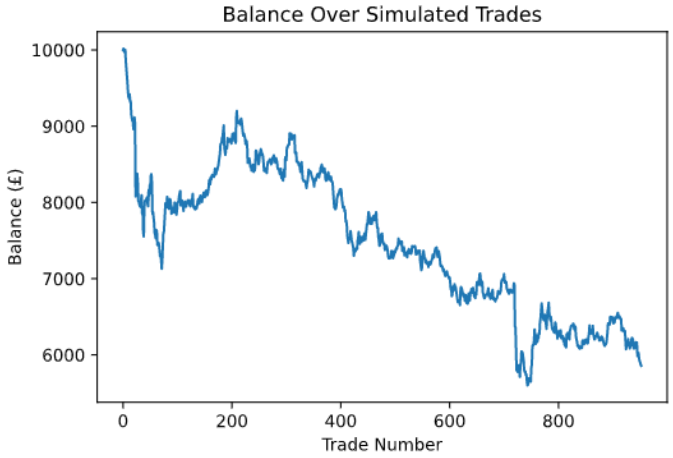

Figure 4.3 - Balance over simulated trades for the final regression model

As expected, results lead to a drastic decrease in account balance (41.43% loss) over around 900 trades, despite a near 50% accuracy. Commission fees whittled away at the account balance periodically. The regression model was unable to perform effectively under real-life conditions.

4.2 Classification Model

The classification model's performance will be analysed more rigorously. The results provide greater contributions to this area of study, therefore deeper analysis will prove valuable.

4.2.1 Base Parameters

Several following sections in this chapter detail results from experimentation with certain parameters or modifications to the model or dataset. Each of these experiments (with the exception of hyperparameter optimisation) have base model and dataset parameters; only one of these parameters is changed during each experiment. The rest remain the same.

These parameters were based on recommendations found in existing literature, having poor base parameters would not provide meaningful results.

The base parameters are as follows:

Table 4.1 - Base Model and Dataset Parameters for Experimentations of Classification Model

Parameter	Value
Architecture	GRU
Bidirectional	No
Hidden Layers	2
Neurons per Layer	100
Batch Size	1024
Dropout	0.2
Initial Learn Rate	0.001
Sequence Length	200
Forecast Period	10

Indicators	No

Firstly, more layers in a neural network exponentially increases training times. Generally, one hidden layer is enough to learn a problem to reasonable accuracy, though a second hidden layer is often more optimal [40] [41].

As for the number of neurons per layer, Jeff Heaton states "the optimal size of the hidden layer is usually between the size of the input and size of the output layers" [42]. Our classification model requires two outputs and a vast number of inputs if we take into account the sequence length of 200 (and the potential use of technical indicators in our dataset). We will later be optimising the number of neurons per hidden layer, thus we left this at 100.

Cheng et al discovered that a dropout of around 0.3 or less works well with the majority of models [43]. Following these recommendations, a dropout rate of 0.2 was chosen.

Initial learn rate was kept at 0.001 which is the default learning rate for Adam optimiser provided by TensorFlow. The initial learn rate (should it not be too high) should not significantly impact resultant accuracy, since Adam optimiser features an adaptive learn rate.

The Batch size is 1024, which is the highest possible allowed by hardware (we are using a v100 GPU with 16GB of VRAM) when considering the most complex model to be trained in our experimentation.

No additional indicator data was used following Hsu et al's findings [12].

4.2.2 RNN Architecture

Architecture of the classification model was modified and the predictive accuracy of the models was monitored. Our first priority here would be to find models with the highest accuracy and lowest Sparse Categorical (Cross) Entropy. If differences were insignificant, training times would decide the most favourable architecture.

Results were averaged over 5 test runs to eliminate the effects of outliers. Metrics stored in the below tables (Table 4.2 and Table 4.3) are final validation metrics after training.

*Table 4.2 - Comparison of validation metrics for **unidirectional** LSTM and GRU architectures for classification model*

	LSTM			GRU		
	Accuracy	**Sparse Categorical Entropy**	**Train Time (Minutes)**	**Accuracy**	**Sparse Categorical Entropy**	**Train Time (Minutes)**
Test 1	0.539	0.6902	1.6	0.553	0.6865	2.0
Test 2	0.538	0.6903	1.4	0.547	0.6863	2.3
Test 3	0.538	0.6900	1.6	0.550	0.6856	2.3
Test 4	0.530	0.6915	1.4	0.545	0.6872	2.0
Test 5	0.507	0.6926	1.3	0.542	0.6878	2.0
Average	**0.530**	**0.6909**	**1.4**	**0.547**	**0.6867**	**2.1**

*Table 4.3 - Comparison of validation metrics for **bidirectional** LSTM and GRU architectures for classification model*

	LSTM (Bidirectional)	GRU (Bidirectional)

	Accuracy	Sparse Categorical Entropy	Train Time (Minutes)	Accuracy	Sparse Categorical Entropy	Train Time (Minutes)
Test 1	0.518	0.6919	2.9	0.547	0.6866	3.7
Test 2	0.542	0.6899	2.6	0.549	0.6866	3.7
Test 3	0.529	0.6913	2.6	0.540	0.6887	3.7
Test 4	0.538	0.6913	3.5	0.543	0.6877	3.8
Test 5	0.525	0.6909	2.6	0.546	0.6886	4.4
Average	**0.530**	**0.6911**	**2.8**	**0.545**	**0.6877**	**3.9**

Results indicated that the use of bidirectional RNN variants did not provide any statistical benefit over unidirectional RNNs for this use case. Additionally, unidirectional RNNs trained significantly faster, since unidirectional RNNs only require a forward pass of the sequences, while bidirectional RNNs require both a forward and backward pass of sequences [26]. This ruled out the use of bidirectional architectures for our final model.

Additionally, both GRU architectures outperformed LSTM architectures. This contradicted initial expectations since literature suggested LSTM may outperform GRU architectures when a larger sequence length was used [21] (a sequence length of 200 was used in this case). For this use case, not only is GRU more efficient (in terms of memory), it is also more accurate.

It is interesting to see consistently higher training times for GRU when compared to LSTM, despite GRU's simpler memory-cell structure. This is likely simply due to GRU overfitting at later epochs, thus early-stopping halts training after a longer period.

4.2.3 Indicators versus no Indicators

We expected the addition of indicators to increase model accuracy just as Hsu et al hypothesised [12]. The idea behind this prediction was that the indicators could have smoothed the noisy price action, thus leading to better generalisation capabilities. When looking at Table 4.4 we can see that this is not the case.

Table 4.4 – Validation metrics for indicator experimentation using classification model

	Indicators			No Indicators		
	Accuracy	Sparse Categorical Entropy	Train Time (Minutes)	Accuracy	Sparse Categorical Entropy	Train Time (Minutes)
Test 1	0.544	0.6872	2.0	0.547	0.6864	2.2
Test 2	0.550	0.6864	2.3	0.540	0.6876	2.2
Test 3	0.547	0.6866	2.1	0.549	0.6864	2.0
Test 4	0.549	0.6863	2.0	0.556	0.6852	2.0
Test 5	0.554	0.6858	2.1	0.546	0.6875	2.1
Average	**0.549**	**0.6865**	**2.1**	**0.547**	**0.6866**	**2.1**

From these results we can conclude that the addition of indicators does not significantly increase the predictive capabilities of the model – the addition of indicators only increased accuracy by 0.2% on average. It is reasonable to interpret this minor increase in accuracy as the result of chance; running the tests again we may find that no indicators have a higher overall accuracy.

The addition of indicators adds an additional 15 values to each data point in the sequences; therefore, this approach causes a drastic increase in the use of VRAM, therefore we will omit the use of indicators for the final model.

Despite Hsu et al's hypotheses, they similarly concluded that indicators did not significantly increase accuracy, though this contradicted the findings of Demir et al [13], who discovered significant increases in accuracy by using technical indicators to predict electricity price. There is potential that this could be the result of our indicator selection method. Demir et al approached this by handpicking indicators that highlighted oscillations or trends in prices (rather than attempting to maximise variation in data, much like our approach).

4.2.4 Effect of Sequence Length

A higher sequence length may provide the model with more data to work with, therefore, should the additional data also be important, we would expect an increase in accuracy. In contrast, a sequence length that is too long may end up being counterproductive since LSTM or GRU memory-cells will have some difficulty remembering information too far away from the most recent data point [44].

Results for variation of sequence length can be seen in the table below (Table 4.5):

Table 4.5 - Validation metrics for classification models with varying sequence lengths

Length	Accuracy	Sparse Categorical Entropy	Train Time (Minutes)
50	0.545691	0.687607	0.640377
100	0.548006	0.686467	1.091628
150	0.547626	0.686195	1.553121
200	0.540233	0.688038	2.007162
250	0.540924	0.687546	2.269982
300	0.556876	0.684797	3.354287
350	0.543036	0.687603	3.662318
400	0.550387	0.686008	4.395992

The results suggest that longer sequence length don't necessarily always lead to increases in accuracy. There is an optimal sequence length dependant on architecture and the dataset. In this case, all the tested sequence lengths result in similar model accuracies (when forecasting 10 minutes into the future). We can assume the slight insignificant changes in accuracy were due to the slightly random nature of training a model; starting neuron connection weights and connections that are cut off as a result of dropout vary in the training of each model [45].

Of course, training times and VRAM usage for those with longer sequence lengths is significantly higher. Because of this, a sequence length of 100 was chosen for the final model (100 was picked over 50 to ensure enough data points were available for possible unforeseen contingencies).

4.2.5 Effect of Forecast Period

Table 4.6 - Validation metrics for classification models with varying forecast periods

Forecast Period	Accuracy	Sparse Categorical Entropy	Train Time (Minutes)
1	0.533769	0.690035	2.644503

5	0.549016	0.686612	2.193867
10	0.542076	0.688374	2.009085
20	0.533822	0.690954	1.708920
30	0.529190	0.691260	1.563716
50	0.509222	0.693329	1.661368

4.2.6 Hyperparameter Optimisation using Genetic Algorithm

Based on prior results, a unidirectional GRU architecture, a sequence length of 100, and a forecast period of 10, and no indicators were used for the final classification model. The base model hyperparameters used during prior tests may not necessarily be optimal, therefore we optimised the hyperparameters using Genetic Algorithm. The fitnesses of the models over generations can be seen below (Figure 4.4).

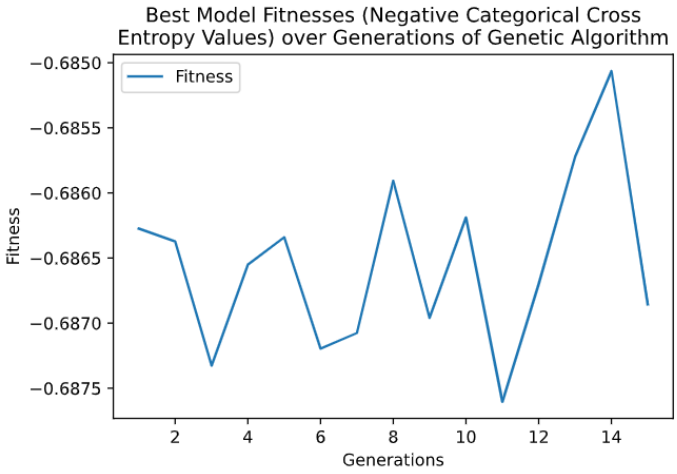

Figure 4.4 - Best classification model finesses over generations of Genetic Algorithm

A population size of 10, a crossover rate of 0.9, a mutation rate of 0.2 and elitism (with 2 elite) was used. The Genetic Algorithm was run over 15 generations. The metric used for fitness was (negative) Categorical Cross Entropy. Best model fitnesses over generations lacked stability, though is again likely due to the random nature of training Artificial Neural Networks. The most accurate model was discovered in generation 14. This model's hyperparameters was subsequently used in the final model. The hyperparameters corresponding to the most accurate model were (Table 4.7):

Table 4.7 - Final model hyperparameters

Hyperparameter	Value
Hidden Layers	2
Neurons per Layer	60
Batch Size	1534
Dropout	0.471
Initial Learn Rate	0.00373

Dropout and initial learn rate are shown to 3 significant figures.

The near-optimal hyperparameters discovered are similar to those used for our base hyperparameters which were based on recommendations from literature. The only surprising result is the abnormally high dropout rate present in final hyperparameters (0.471). Similar high dropout rates were present throughout the majority of models with the highest fitnesses over generations; we can assume this is no mistake. Cheng et al recommended dropout rates in the region of 0.2 or 0.3 [43], though this may vary depending on the complexity of the problem. Due to high levels of noise and the sheer complexity of financial markets, a higher dropout rate was likely more optimal. It prevented the model from overfitting. This finding will prove useful when manually optimising models in future work.

4.2.7 Final Classification Model Results

As previously stated, final model hyperparameters can be seen in Table 4.7. These were used when training the final model. While the model hyperparameters are identical to those discovered using Genetic Algorithm, resultant validation metrics can still differ slightly due to different initial weights and the random nature of dropout. Overall, the training process of the final model was fairly stable; both the accuracy (Figure 4.5) and loss (Figure 4.6) did not diverge. In other words, there were little signs of overtraining.

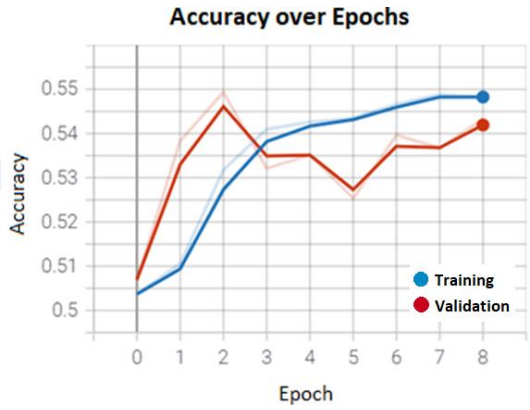

Figure 4.5 - Validation and training accuracy over epochs for final classification model

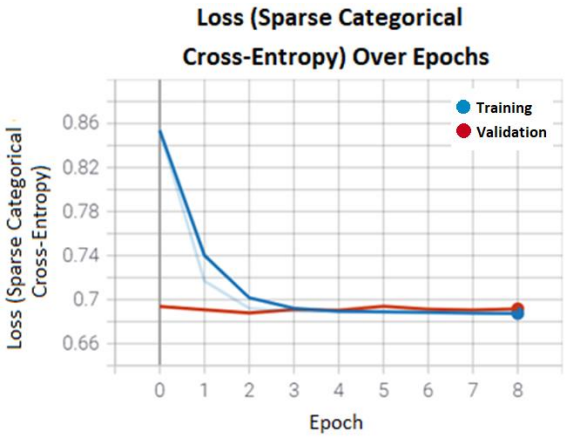

Figure 4.6 - Validation and training loss over epochs for final classification model

Early stopping reverted the model weights to those present at epoch 2 where validation metrics were at their best. At this point, validation accuracy was around 0.55 (55%) while categorical cross-entropy was around 0.683.

We later tested the model on unseen test data. The data was unshuffled and from a future period of time (in relation to training and validation data). The model achieved a classification accuracy of 54.8% and a categorical cross-entropy of 0.689. While these accuracy metrics are fairly impressive (anything consistently over 50% is impressive in financial markets) we can further increase accuracy by selecting only the top 20% of predictions; the predictions the model is most certain are correct. As we stated before, the classification model has two outputs. We can think of output one as the model's confidence that future price will decrease and output two as the model's confidence that future price will increase. When the difference between these two outputs is highest, the model is most certain about that particular prediction/classification.

When tested on only the top 20% of predictions, the accuracy metrics drastically improved, increasing the classification accuracy to 59.2% and decreasing the categorical cross-entropy to 0.682.

Next, the model and test data predictions were used in the created trading simulator. The model saw a steady and stable increase in account balance of 38.94% over 565 trades (Figure 4.7).

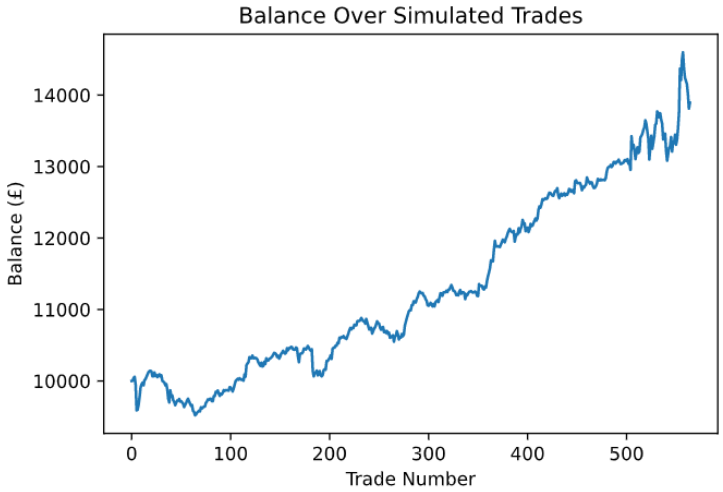

Figure 4.7 - Account balance over simulated trades for final classification model

Additionally, 60.11% of these trades were winning trades. On average, each winning trade adds 0.37% to the account, while each losing trade would take 0.36% from the account, thus, the expected profit per trade is 0.079% (2sf):

$$(win\ probability * avg\ win) - (loss\ probability * avg\ loss) = expected\ profit$$

$$\downarrow$$

$$(0.6011 * 0.37) - \big((1 - 0.6011) * 0.36\big) = 0.078803\ \%$$

We would have expected accuracy to decrease over time since markets are dynamic, and are constantly adapting. Profitable trading strategies cease to last forever [27], however the trading strategy this model adopted survived the period of time it was simulated over.

Looking at confusion matrices for the model's predictions give us a little more insight into the strategies and biases of the model:

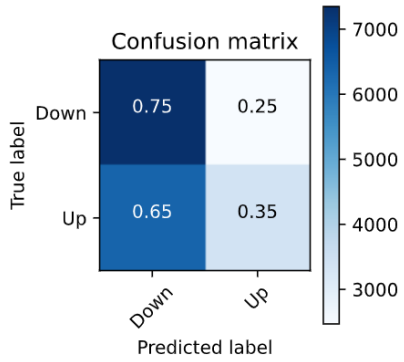

Figure 4.8 - Confusion matrix for final classification model's predictions on test data

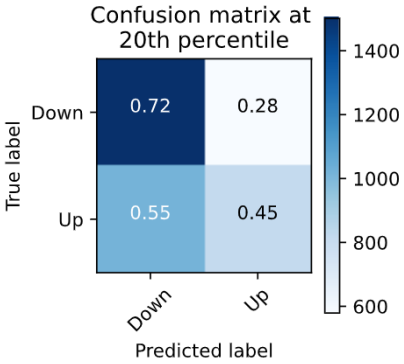

Figure 4.9 - Confusion matrix for final classification model's predictions on test data (best 20% of predictions)

Figure 4.8 is a confusion matrix that comprises all predictions while Figure 4.9 is a confusion matrix that includes only the top 20% of predictions. Both matrices display similar results, showing the model having a bias towards predicting prices will decrease. This is intriguing since, over the period represented in the dataset, the price of Bitcoin was generally increasing in a steady uptrend. Perhaps downward movements are more predictable for this trading vehicle. Despite the lengthy uptrend, the majority of predictions are correct.

4.3 Significance of Findings

Our resultant model managed to achieve an overall accuracy of 60.11% when used in a simulated trading scenario, which translated into a positive overall expectancy (for profits). An accuracy of 60.11% rivals and often outperforms the predictive accuracy of findings in existing literature (especially for forecasting bitcoin price). McNally et all achieved a final classification accuracy of 52% [8] while Madan et al predicted future price direction of Bitcoin with 50-55% accuracy for 10-minute intervals [9]. Additionally, as markets continue to adapt, the usage of AI in financial markets will increase (following current trends) [46] hereby it is to be expected that drawing profits using machine learning is increasing in difficulty. Older models such as those seen in existing literature may result in

poor results when used in today's trading environment. The fact that our model managed to result in consistent profitability despite the increasing domination of AI in financial markets is impressive.

Our usage of a simulated trading environment is completely original. This is nowhere to be seen in the existing literature. This is quite unfortunate since displaying the potential effectiveness of the model in a real-life scenario is a simple yet effective method for communicating the models' capabilities. It additionally helps with finding potential loopholes in methodology and results. For example, it is likely that a large number of models present in literature simple achieve high levels of accuracy since the same prediction is continually predicted (for example when a trend is identified, the model may predict an increase in price 10 times in a row). In a real-life scenario, you can only trade that single trend once (you can only spend money once), therefore when traded, accuracy would be less than claimed.

Whilst computationally expensive, hyperparameters and therefore overall model performance was optimised using Genetic Algorithm. This is rarely performed in the literature for forecasting financial markets, possible due to the sheer amount of computation time required. Despite the drawbacks, hyperparameter optimisation is highly beneficial and can lead to surprising results, such as our surprisingly high dropout rate in our most optimal models.

In agreeance with a number of existing studies, we found that a lower forecast period does not necessarily lead to higher classification accuracy. As an example, Madan et al found dramatically high accuracy when predicting daily price movements over 10-minute price movements. This is likely due to the high amount of noise found in financial markets at lower timeframes.

4.4 Challenges

There were several challenges that this study faces. Identifying these challenges can help us overcome these challenges in future work.

Firstly, as a precaution, it is entirely possible that some of the findings cannot be generalised to all situations, only to the base parameters chosen. For example, higher sequence lengths did not elicit significant changes in the predictive accuracy of the model. Despite this, higher sequence lengths may have worked better with LSTM just as Gao et al [21] stated, though GRU was used in the base parameters for our experimentation. Further generalisation of findings may require additional experimentation.

Our results showed us that indicators did not lead to improvements in overall accuracy. The literature is in fierce debate about the predictive ability of technical indicators in financial markets. While we concluded here that indicators did not lead to increases in accuracy, there is potential that a different indicator selection method could have led us to a different conclusion. For example, Demir et al [13] approached this by handpicking indicators that highlighted oscillations or trends in prices and found noteworthy improvements in accuracy as a result.

Throughout this study, predictions were made using 1-minute price data for Bitcoin. We achieved respectable accuracy, though it is possible that accuracy could be further increased by using price data from high timeframes (such as daily price data). This is precisely what Madan et al found [9]. There are drawbacks and challenges to using this approach, however. Firstly, using lower timeframes allows

the use of larger datasets; there is more data when recorded every minute as opposed to every day. Additionally, the Adaptive Markets Hypothesis [27] may mean that daily price data from far in the past may have little relevance to today's price movements. The benefit gained in predictive accuracy may outweigh the drawbacks of this approach, however.

While we can see that our implementation of Genetic Algorithm led to an increase in final model accuracy, there is potential that providing the algorithm with more hyperparameters and additional model customisability could have led to a further improved model. For example, the algorithm may have discovered a vastly different set of hyperparameters if each layer could have a different number of neurons (rather than each layer having the same number of neurons). This does further complicate implementation, however.

Lastly, the results found using our final regression model were overshadowed by the overwhelmingly positive results found for the final classification model. The regression model tended towards predicting mean prices, consequently leading to a lack of directional accuracy. A regression problem has a much larger scope for error than a two-class classification problem, therefore a much more complex architecture could be required to more accurately predict prices.

4.5 Summary

There were many surprising discoveries elicited from our experimentation. Firstly, and most importantly, the use of a classification model reduced the scope for error, and thus led to an increase in accuracy when predicting future market direction.

When using base hyperparameters picked using recommendations from literature, for this specific problem, bidirectional RNN architectures did not perform significantly better than unidirectional variants, thus it did not make sense to utilise them in the final model (due to less optimal training times). Additionally, GRU architectures performed better than LSTM architectures, despite using a reasonably large sequence length (for our base parameters) of 200.

The addition of technical indicators in our dataset did not increase the accuracy of the model. The increased dataset dimensions when using technical indicators increased VRAM usage therefore technical indicators were not used in the final model.

The lower forecast periods intriguingly did not result in the highest accuracies. We can infer that lower forecast periods may be difficult to predict because the price of financial markets is generally quite noisy on lower time scales.

Hyperparameter optimisation provided some unexpected results. While most hyperparameters were close to our predicted optimal hyperparameters (recommended from literature), the dropout rate was fairly large (0.471), while the literature recommended 0.2 to 0.3 [43]. This may have been due to the high complexity of the problem.

Lastly, our final classification model (which used the near-optimal hyperparameters from Genetic Algorithm) managed to achieve a classification accuracy of 59.2% (when using the top 20% of predictions), increase the balance by 38.94% over 565 trades in our trading simulation (trading the correct direction 60.11% of the time), and achieved an expected profit of 0.079% of the account balance per trade. These results are astounding, especially given the prominence of automated trading for Bitcoin in today's climate.

Chapter 5
Conclusions and Future Work

5.1 Conclusions

The main aim of the study was to discover whether Recurrent Neural Networks can be trained to predict future price direction using price and volume data alone. This was explored by creating two final RNN models that predicted future Bitcoin prices and prices directions, a regression model and a classification model. The final classification model performed significantly better than the regression model, likely due to the reduced scope for error. The classification model managed to achieve a classification accuracy of 54.8%. This accuracy increased to 59.2% when only the top 20% of predictions were selected. This accuracy rivals and is superior in most cases to existing literature. McNally et all achieved a final classification accuracy of 52% on Bitcoin [8] while Madan et al predicted future price direction of Bitcoin with 50-55% accuracy for 10-minute intervals [9].

When placed into a trading simulator, the classification model managed to increase the starting balance by 38.94% over 565 simulated trades. 60.11% of trades were correct, and the expected profit per trade was 0.079% of the current account balance. Our confusion matrix suggested that the model was biased towards predicting future price would decrease, despite Bitcoins overall uptrend between 2017 and mid-2020 (the approximate time period of our dataset). Regardless, simulated balance increased in a stable fashion.

A secondary aim involved discovering how changes to the dataset and model architecture affected overall accuracy when predicting future price direction. This was tested by using a base set of dataset and model parameters, and modifying these parameters individually while measuring validation accuracy metrics after training. We discovered that GRU architecture performed better than LSTM architecture despite the high sequence length of 200, contrary to the suppositions of Gao et al [21], who stated LSTM may be more accurate at higher sequence lengths. We also discovered that bidirectional variants did not significantly increase accuracy, despite the favour of bidirectional variants literature. Additionally, we concluded the addition of technical indicators in the dataset and drastically increasing the sequence length provided no noticeable benefits. Lastly, to our surprise, a shorter forecast period did not necessarily result in higher accuracy, likely due to the larger amounts of noise and unpredictability in markets at lower timeframes.

The final secondary aim includes the discovery of optimal hyperparameters. We approached this by using Genetic Algorithm to progressively discover near-optimal hyperparameters for our regression and classification model. The majority of resulting hyperparameters turned out to be near those recommended in literature, though dropout was exceptionally high (0.471) likely due to the complexity of the problem

5.2 Future Work

Though our study provided engaging results and achieved significant accuracy, it can still further be extended to improve accuracy and extend or verify findings.

Firstly, a large portion of manual traders attempt to recognise common and popular patterns in market price data in order to predict future prices direction and market sentiment [47]. In future work, we could incorporate pattern recognition algorithms such as Dynamic Time Warping [48]. Data related to the patterns extracted using the pattern recognition algorithms could be added to the dataset.

Additionally, while price and volume history tells a story about the markets, and may drive a portion of the future movement, markets are also heavily influenced by news and world events. Oncharoen et al [49] found that the predictive accuracy of their machine learning models was improved by combining both technical (price) aspects and fundamental (news) aspects. Implementing this multidisciplinary approach into future work will likely result in superior model capabilities.

While this study focussed on the use of Recurrent Neural Network architectures to predict time series data, additional (recent) approaches for predicting time series data exist and rival their predictive capabilities. A number of studies merit the use of Convolutional Neural Networks (CNNs) for predicting time-series data such as market price prediction. Selvin et al [50] discovered that a Convolutional Neural Network architecture resulted in higher predictive accuracy than LSTM architecture for predicting future stock price (regression). While CNNs may result in increases in accuracy, their increased number of hyperparameters may increase the complexity and difficulty of finding optimal hyperparameters, though the benefits may outweigh the challenges.

While our technical indicator selection method had merit (increasing variance of data by reducing correlations between indicators) it is possible that the use of other methods could have led to different results. Demir et al [13] approached selection by handpicking indicators that highlighted oscillations or trends and found noteworthy improvements in accuracy as a result.

Lastly, our dataset includes short-term one-minute price and volume data, which limited us to mainly predicting short term price movements. Studies have often found greater difficulty in predicting shorter time fluctuations in price using machine learning approaches [9]. Collecting longer-term data (such as hourly or daily data) and attempting to forecast days or weeks into the future could result in higher model accuracy.

Chapter 6
Reflection

Before the creation of this project, my machine learning knowledge was limited. My interest in the field made me eager to learn more; this project was the perfect opportunity. At this present day, my knowledge in the field has drastically improved. I not only understand how Recurrent Neural Networks work, but I am also able to use that knowledge to practically implement models that can be used to forecast time-series data. Moreover, through learning TensorFlow, the machine learning framework provided by Google, I can extend this knowledge to a variety of other machine learning algorithms, such as Convolutional Neural Networks or Random Forest.

Additionally, my ability to navigate literature has drastically improved. This project has also taught me the value of checking existing literature before undergoing a large project. Existing discoveries from state-of-the-art literature greatly influenced the direction of this project and helped me avoid arriving at dead-ends.

The experimental aspect of this project also taught me to value self-discovery. In other words, not all results from the literature are axiomatic, and results may not generalise to your own application, therefore you may find different results.

Throughout this project, there were a vast number of difficulties faced. One was already detailed; learning a new and seemingly complex machine learning framework. Another challenge was obtaining impressive accuracy for the final model. Initially, a regression model was used, though this did not generate the results desired. These results were later overshadowed by the results elicited by the adoption of a classification model.

Another major pain point in this project was training times. Generally, in software development, the development process is relatively quick; you can add code, recompile, and run the software to test the effect of new code. In machine learning, you may have to wait for the model to finish training before you notice that something isn't quite right, or that an optimisation had no effect (or even a negative effect). This was especially true when using Genetic Algorithm to optimise model hyperparameters since Genetic Algorithm required numerous models to be trained (150 to be precise, 10 per population and 15 generations). On the first couple of runs, the implementation of Genetic Algorithm was not quite right (and elitism was not implemented). This was not evident until the results were generated, and the code was analysed. The algorithm later had to be rerun.

The project initiation document was titled "Time Series Modelling using Machine Learning". While technically nothing has changed, and this is exactly what my project has broadly detailed, the project and project title became more focused as time progressed. Initially, I had planned to forecast future market price on stock data, and the dataset was originally gathered manually. Despite this, difficulties throughout the project lead to change from stock price data to cryptocurrency data. This was due to the wide and free availability of cryptocurrency price data. Additionally, the model had difficulties in training on the stock dataset gathered since the stock data had "gap ups" and "gap downs". These are essentially periods where there is no data or trading activity since the stock market trading is not open 24/7 (while cryptocurrency trading is).

Some of the initial difficulties in this project stemmed from attempts to traverse such a complex field alone, therefore, if I could do this project a second time, I would have first looked at the approaches used in existing literature. Doing this first would have saved precious time (rather than creating a model alone, finding unsatisfactory results, then looking to literature).

Overall, this project has taught me invaluable skills in an expanding field (machine learning) that is currently in its infancy. These skills are becoming increasingly desirable among employers and innovators. I will continue to learn and refine my machine learning knowledge in the future such that I can innovate beyond the scope of this project.

Git Repository (Full Code)

Full code can be found in the git repository hosted on CSGitLab (the University of Reading's privately hosted GitLab). This includes the dataset used.

This can be found at:

https://csgitlab.reading.ac.uk/ann-price-prediction-final-year-project/bi-lstm-gru-bitcoin-forecasting

References

[1] C. D. K. II and J. A. Dahlquist, *Technical Analysis: The Complete Resource for Financial Market Technicians*. FT Press, 2010, pp. 3–4.

[2] C.-H. Park and S. H. Irwin, 'What Do We Know About the Profitability of Technical Analysis?', *J. Econ. Surv.*, vol. 21, no. 4, pp. 786–826, 2007, doi: https://doi.org/10.1111/j.1467-6419.2007.00519.x.

[3] B. Huang, Y. Huan, L. D. Xu, L. Zheng, and Z. Zou, 'Automated trading systems statistical and machine learning methods and hardware implementation: a survey', *Enterp. Inf. Syst.*, vol. 13, no. 1, pp. 132–144, Jan. 2019, doi: 10.1080/17517575.2018.1493145.

[4] A. Shewalkar, D. Nyavanandi, and S. Ludwig, 'Performance Evaluation of Deep Neural Networks Applied to Speech Recognition: RNN, LSTM and GRU', *J. Artif. Intell. Soft Comput. Res.*, vol. 9, pp. 235–245, Oct. 2019, doi: 10.2478/jaiscr-2019-0006.

[5] Y. Bengio, 'Gradient-Based Optimization of Hyperparameters', *Neural Comput.*, vol. 12, no. 8, pp. 1889–1900, Aug. 2000, doi: 10.1162/089976600300015187.

[6] J. Smit, 'Binance Full History', *Kaggle*, Feb. 01, 2021. https://kaggle.com/jorijnsmit/binance-full-history (accessed Apr. 05, 2021).

[7] J. Brownlee, *Machine Learning Mastery With Python: Understand Your Data, Create Accurate Models, and Work Projects End-to-End*. Machine Learning Mastery, 2016, p. 16.

[8] S. McNally, J. Roche, and S. Caton, 'Predicting the Price of Bitcoin Using Machine Learning', in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Mar. 2018, pp. 339–343, doi: 10.1109/PDP2018.2018.00060.

[9] I. Madan, S. Saluja, and A. Zhao, 'Automated Bitcoin Trading via Machine Learning Algorithms', p. 5, 2015.

[10] A. Greaves and B. Au, 'Using the Bitcoin Transaction Graph to Predict the Price of Bitcoin', 2015. /paper/Using-the-Bitcoin-Transaction-Graph-to-Predict-the-Greaves-Au/a0ce864663c100582805ffa88918910da89add47 (accessed Apr. 28, 2021).

[11] Y. Bodyanskiy and S. Popov, 'Neural network approach to forecasting of quasiperiodic financial time series', *Eur. J. Oper. Res.*, vol. 175, no. 3, pp. 1357–1366, Dec. 2006, doi: 10.1016/j.ejor.2005.02.012.

[12] M.-W. Hsu, S. Lessmann, M.-C. Sung, T. Ma, and J. E. V. Johnson, 'Bridging the divide in financial market forecasting: machine learners vs. financial economists', *Expert Syst. Appl.*, vol. 61, pp. 215–234, Nov. 2016, doi: 10.1016/j.eswa.2016.05.033.

[13] S. Demir, K. Mincev, K. Kok, and N. G. Paterakis, 'Introducing Technical Indicators to Electricity Price Forecasting: A Feature Engineering Study for Linear, Ensemble, and Deep Machine Learning Models', *Appl. Sci.*, vol. 10, no. 1, Art. no. 1, Jan. 2020, doi: 10.3390/app10010255.

[14] S. Boonprong, C. Cao, W. Chen, X. Ni, M. Xu, and B. K. Acharya, 'The Classification of Noise-Afflicted Remotely Sensed Data Using Three Machine-Learning Techniques: Effect of Different Levels and Types of Noise on Accuracy', *ISPRS Int. J. Geo-Inf.*, vol. 7, no. 7, Art. no. 7, Jul. 2018, doi: 10.3390/ijgi7070274.

[15] S. Badillo *et al.*, 'An Introduction to Machine Learning', *Clin. Pharmacol. Ther.*, vol. 107, Mar. 2020, doi: 10.1002/cpt.1796.

[16] Z. Dai, H. Zhu, and J. Kang, 'New technical indicators and stock returns predictability', *Int. Rev. Econ. Finance*, vol. 71, pp. 127–142, Jan. 2021, doi: 10.1016/j.iref.2020.09.006.

[17] L. Yin and Q. Yang, 'Predicting the oil prices: Do technical indicators help?', *Energy Econ.*, vol. 56, pp. 338–350, May 2016, doi: 10.1016/j.eneco.2016.03.017.

[18] B. G. Malkiel and E. F. Fama, 'Efficient Capital Markets: A Review of Theory and Empirical Work*', *J. Finance*, vol. 25, no. 2, pp. 383–417, 1970, doi: https://doi.org/10.1111/j.1540-6261.1970.tb00518.x.

[19] D. A. Lesmond, M. J. Schill, and C. Zhou, 'The illusory nature of momentum profits', *J. Financ. Econ.*, vol. 71, no. 2, pp. 349–380, Feb. 2004, doi: 10.1016/S0304-405X(03)00206-X.

[20] K. A. Althelaya, E. M. El-Alfy, and S. Mohammed, 'Stock Market Forecast Using Multivariate Analysis with Bidirectional and Stacked (LSTM, GRU)', in *2018 21st Saudi Computer Society National Computer Conference (NCC)*, Apr. 2018, pp. 1–7, doi: 10.1109/NCG.2018.8593076.

[21] S. Gao *et al.*, 'Short-term runoff prediction with GRU and LSTM networks without requiring time step optimization during sample generation', *J. Hydrol.*, vol. 589, p. 125188, Oct. 2020, doi: 10.1016/j.jhydrol.2020.125188.

[22] S. Yang, X. Yu, and Y. Zhou, 'LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example', in *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*, Jun. 2020, pp. 98–101, doi: 10.1109/IWECAI50956.2020.00027.

[23] G.-B. Zhou, J. Wu, C.-L. Zhang, and Z.-H. Zhou, 'Minimal gated unit for recurrent neural networks', *Int. J. Autom. Comput.*, vol. 13, no. 3, pp. 226–234, Jun. 2016, doi: 10.1007/s11633-016-1006-2.

[24] L. Qi, M. Khushi, and J. Poon, 'Event-Driven LSTM For Forex Price Prediction', *ArXiv210201499 Q-Fin*, Jan. 2021, Accessed: Apr. 06, 2021. [Online]. Available: http://arxiv.org/abs/2102.01499.

[25] Z. Cui, R. Ke, Z. Pu, and Y. Wang, 'Stacked bidirectional and unidirectional LSTM recurrent neural network for forecasting network-wide traffic state with missing values', *Transp. Res. Part C Emerg. Technol.*, vol. 118, p. 102674, Sep. 2020, doi: 10.1016/j.trc.2020.102674.

[26] M. Schuster and K. K. Paliwal, 'Bidirectional recurrent neural networks', *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997, doi: 10.1109/78.650093.

[27] A. W. Lo, 'Adaptive Markets and the New World Order (corrected May 2012)', *Financ. Anal. J.*, vol. 68, no. 2, pp. 18–29, Mar. 2012, doi: 10.2469/faj.v68.n2.6.

[28] J. Sola and J. Sevilla, 'Importance of input data normalization for the application of neural networks to complex industrial problems', *IEEE Trans. Nucl. Sci.*, vol. 44, no. 3, pp. 1464–1468, Jun. 1997, doi: 10.1109/23.589532.

[29] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, 'Speeding Up Distributed Machine Learning Using Codes', *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, Mar. 2018, doi: 10.1109/TIT.2017.2736066.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, 'Dropout: a simple way to prevent neural networks from overfitting', *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[31] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, 'How Does Batch Normalization Help Optimization?', *ArXiv180511604 Cs Stat*, Apr. 2019, Accessed: Apr. 14, 2021. [Online]. Available: http://arxiv.org/abs/1805.11604.

[32] R. Tutunov, M. Li, A. I. Cowen-Rivers, J. Wang, and H. Bou-Ammar, 'Compositional ADAM: An Adaptive Compositional Solver', *ArXiv200203755 Cs Math Stat*, Apr. 2020, Accessed: Apr. 14, 2021. [Online]. Available: http://arxiv.org/abs/2002.03755.

[33] T. K. Leen, T. G. Dietterich, and V. Tresp, *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*. MIT Press, 2001, pp. 402–403.

[34] S. N. Sivanandam and S. N. Deepa, *Genetic Algorithms*. Berlin, Heidelberg: Springer, 2008, pp. 16–19.

[35] W. Spears and K. De Jong, *On the Virtues of Parametrized Uniform Crossover*. 1991.

[36] Chang Wook Ahn and R. S. Ramakrishna, 'Elitism-based compact genetic algorithms', *IEEE Trans. Evol. Comput.*, vol. 7, no. 4, pp. 367–385, Aug. 2003, doi: 10.1109/TEVC.2003.814633.

[37] M. Laumanns, E. Zitzler, and L. Thiele, 'On The Effects of Archiving, Elitism, And Density Based Selection in Evolutionary Multi-Objective Optimization', in *In*, 2001, pp. 181–196, doi: https://doi.org/10.1007/3-540-44719-9_13.

[38] Z. H. Khan, T. S. Alin, and A. Hussain, 'Price Prediction of Share Market using Artificial Neural Network (ANN)', *International Journal of Computer Applications*, vol. 22, no. 2, 2011, doi: 10.5120/2552-3497.

[39] D. G. Mayer and D. G. Butler, 'Statistical validation', *Ecol. Model.*, vol. 68, no. 1, pp. 21–32, Jul. 1993, doi: 10.1016/0304-3800(93)90105-2.

[40] A. J. Thomas, M. Petridis, S. Walters, S. M. Gheytassi, and R. E. Morgan, 'Two Hidden Layers are Usually Better than One', 2017, doi: 10.1007/978-3-319-65172-9_24.

[41] D. Stathakis, 'How many hidden layers and nodes?', *Int. J. Remote Sens.*, vol. 30, no. 8, pp. 2133–2147, Apr. 2009, doi: 10.1080/01431160802549278.

[42] J. Heaton, 'The Number of Hidden Layers', *Heaton Research*, Jun. 01, 2017. https://www.heatonresearch.com/2017/06/01/hidden-layers.html (accessed Apr. 11, 2021).

[43] G. Cheng, V. Peddinti, D. Povey, V. Manohar, S. Khudanpur, and Y. Yan, *An Exploration of Dropout with LSTMs*. 2017, p. 1590.

[44] G. Rao, W. Huang, Z. Feng, and Q. Cong, 'LSTM with sentence representations for document-level sentiment classification', *Neurocomputing*, vol. 308, pp. 49–57, Sep. 2018, doi: 10.1016/j.neucom.2018.04.045.

[45] S. Rahnamayan, H. R. Tizhoosh, and M. M. A. Salama, 'Opposition versus randomness in soft computing techniques', *Appl. Soft Comput.*, vol. 8, no. 2, pp. 906–918, Mar. 2008, doi: 10.1016/j.asoc.2007.07.010.

[46] H. Ghoddusi, G. G. Creamer, and N. Rafizadeh, 'Machine learning in energy economics and finance: A review', *Energy Econ.*, vol. 81, pp. 709–727, Jun. 2019, doi: 10.1016/j.eneco.2019.05.006.

[47] J.-L. Wang and S.-H. Chan, 'Stock market trading rule discovery using pattern recognition and technical analysis', *Expert Syst. Appl.*, vol. 33, no. 2, pp. 304–315, Aug. 2007, doi: 10.1016/j.eswa.2006.05.002.

[48] D. J. Berndt and J. Clifford, 'Using dynamic time warping to find patterns in time series', in *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, Seattle, WA, Jul. 1994, pp. 359–370, Accessed: Apr. 27, 2021. [Online].

[49] P. Oncharoen and P. Vateekul, 'Deep Learning for Stock Market Prediction Using Event Embedding and Technical Indicators', in *2018 5th International Conference on Advanced Informatics: Concept Theory and Applications (ICAICTA)*, Aug. 2018, pp. 19–24, doi: 10.1109/ICAICTA.2018.8541310.

[50] S. Selvin, R. Vinayakumar, E. A. Gopalakrishnan, V. K. Menon, and K. P. Soman, 'Stock price prediction using LSTM, RNN and CNN-sliding window model', in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sep. 2017, pp. 1643–1647, doi: 10.1109/ICACCI.2017.8126078.

Appendices

Appendix Chapter 1 – Regression Model Results

1.1 RNN Architecture

	LSTM			GRU		
	R Square	MAE	Train Time (Minutes)	R Square	MAE	Train Time (Minutes)
Test 1	-0.0325	0.001040	6.0	-0.0335	0.001042	4.1
Test 2	-111.7242	0.009462	1.3	-0.0313	0.001041	5.8
Test 3	-0.0311	0.001040	5.3	-0.0305	0.001039	4.8
Test 4	-0.0325	0.001039	6.1	-0.0315	0.001041	6.2
Test 5	-0.0319	0.001040	5.3	-0.0308	0.001039	6.0
Average	-0.0320	0.001040	5.7	-0.0315	0.001040	5.4

	LSTM (Bidirectional)			GRU (Bidirectional)		
	R Square	MAE	Train Time (Minutes)	R Square	MAE	Train Time (Minutes)
Test 1	-0.0340	0.001042	10.7	-0.0329	0.001043	13.0
Test 2	-0.0338	0.001043	13.7	-0.0315	0.001041	12.6
Test 3	-0.0365	0.001045	10.4	-0.0328	0.001041	10.7
Test 4	-0.0370	0.001046	11.7	-0.0331	0.001042	12.9
Test 5	-0.0331	0.001040	12.0	-0.0356	0.001044	9.8
Average	-0.0349	0.001043	11.7	-0.0332	0.001042	11.8

1.2 Indicators versus no Indicators

	Indicators			No Indicators		
	R Square	MAE	Train Time (Minutes)	R Square	MAE	Train Time (Minutes)
Test 1	-0.0351	0.001061	10.1	-0.0354	0.001049	10.7
Test 2	-0.0352	0.001062	13.0	-0.0338	0.001050	14.9
Test 3	-0.0347	0.001060	12.3	-0.0362	0.001051	9.8
Test 4	-0.0351	0.001062	13.0	-0.0377	0.001053	12.0
Test 5	-0.0351	0.001060	12.0	-0.0356	0.001051	11.7
Average	**-0.0350**	**0.001061**	**12.1**	**-0.0357**	**0.001051**	**11.8**

1.3 Effect of Sequence Length

	R Square	MAE	Train Time (Minutes)
Length 50	-0.03413	0.001046	3.458199
Length 100	-0.03441	0.001059	6.780635
Length 150	-0.03575	0.001053	10.6208
Length 200	-0.03896	0.001047	12.71089
Length 250	-0.03417	0.001059	15.34224
Length 300	-0.03379	0.001046	16.98044
Length 350	**-inf**	0.001049	19.14106
Length 400	-0.03482	0.001045	23.0904

1.4 Effect of Forecast Period

	R Square	MAE	Train Time (Minutes)
Forward 1	-0.03692	0.001051	11.6846
Forward 5	-0.03401	0.002374	9.431959
Forward 10	-16.5155	0.013472	2.639607
Forward 20	-0.03187	0.004631	10.41841
Forward 30	-0.02834	0.005606	10.42984

Appendix Chapter 2 – Code Snippets

2.1 Trading Simulation Code (Regression Model)

```
def do_reg_simulation(predictions: np.ndarray, test_y: np.ndarray,
forecast_period: int, percentile = 100.0):
    balance = 10000.0
    balances = [balance]
    wins, losses = [], []
    spread = 0.000122 # As fraction of price
    leverage = 2.0
    upper = np.percentile(predictions, 100.0 - percentile / 2)
    lower = np.percentile(predictions, percentile / 2)
    last_trade_index = -np.inf

    print(f"\n\nStart Balance: {balance}")
    for index, prediction in enumerate(predictions):
        prediction = prediction[0]
        actual = test_y[index]
        if prediction > upper or prediction < lower:
            spread_cost = leverage * balance * spread
            should_buy = prediction > 0 # Buy if positive, sell if negative
            can_trade = index > last_trade_index + forecast_period # Cant
trade if already in a trade
            if not can_trade:
                continue
            if should_buy:
                new_balance = balance * (1 + (leverage * actual)) -
spread_cost
            else: # We are selling
                new_balance = balance * (1 - (leverage * actual)) -
spread_cost
            if new_balance > balance:
                wins.append(abs(leverage * actual))
            else:
                losses.append(abs(leverage * actual))
            last_trade_index = index
            balance = new_balance
            balances.append(balance)

    print(f"Final Balance: {balance: .2f}")
    print("Showing plot for final balance:")
    print(f"{len(balances)} trades executed")
```

```
    print(f"{len(predictions)} total predictions")
    print(f"Percentage wins: {len(wins) / (len(balances) - 1) * 100: .2f}%")
    print(f"Average win % of acc: {np.average(wins): .4f}")
    print(f"Average loss % of acc: {np.average(losses): .4f}")
    print(f"Wins: {len(wins)} --- Losses: {len(losses)} ---
{len(wins)/(len(wins) + len(losses)) * 100: .2f}% wins")
    print(f"% Predictions < 0: {len(predictions[predictions < 0]) /
len(predictions) * 100: .4f}%")

    plt.plot(balances)
    plt.xlabel("Trade Number")
    plt.ylabel("Balance (£)")
    plt.title("Balance Over Simulated Trades")
    plt.draw()
```

2.2 Trading Simulation Code (Classification Model)

```
def do_simulation(predictions: np.ndarray, test_y: np.ndarray, percentages:
np.ndarray, forecast_period = 0, percentile = 0.0):
    balance = 10000.0
    balances = [balance]
    wins, losses = [], []
    spread = 0.000122 # As fraction of price
    leverage = 2.0
    last_trade_index = -np.inf

    difs = [abs(x - y) for x, y in predictions]
    min_dif = np.percentile(difs, 100.0 - percentile)

    print(f"\n\nStart Balance: {balance}")
    for index, confidences in enumerate(predictions):
        prediction = np.argmax(confidences)
        confidence = abs(confidences[prediction])
        actual = test_y[index]
        percent = percentages[index]
        dif = abs(confidences[0] - confidences[1])
        can_trade = index > last_trade_index + forecast_period # Cant trade if
already in a trade
        if dif >= min_dif and can_trade:
            spread_cost = leverage * balance * spread
            new_balance = 0
            if prediction == actual: ## Correct direction
                new_balance = balance + (balance * abs(percent) * leverage) -
spread_cost
                wins.append(abs(percent) * leverage)
            else: ## Incorrect direction
                new_balance = balance - (balance * abs(percent) * leverage) -
spread_cost
                losses.append(abs(percent) * leverage)
            last_trade_index = index
            balance = new_balance
            balances.append(balance)
```

```
    print(f"Final Balance: {balance: .2f}")
    print("Showing plot for final balance:")
    print(f"{len(balances)} trades executed")
    print(f"{len(predictions)} total predictions")
    print(f"Percentage wins: {len(wins) / (len(balances) - 1) * 100: .2f}%")
    print(f"Average win % of acc: {np.average(wins) * 100: .2f}%")
    print(f"Average loss % of acc: {np.average(losses) * 100: .2f}%")
    print(f"Wins: {len(wins)} --- Losses: {len(losses)} ---
{len(wins)/(len(wins) + len(losses)) * 100: .2f}% wins")

    plt.plot(balances)
    plt.xlabel("Trade Number")
    plt.ylabel("Balance (£)")
    plt.title("Balance Over Simulated Trades")
    plt.draw()
```

2.3 Full Model Class Code

```
from datetime import datetime
import time
import tensorflow as tf
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Dropout, BatchNormalization,
LSTM, GRU, CuDNNLSTM, CuDNNGRU, Bidirectional
from tensorflow.python.keras.callbacks import TensorBoard, EarlyStopping
from tensorflow.python.client import device_lib
import numpy as np
import os

from app.parameters import Architecture
from .RSquaredMetric import RSquaredMetric, RSquaredMetricNeg


class Model():
    def __init__(self, train_x, train_y,
        validation_x , validation_y, seq_info:str,
        *,
        max_epochs = 100, batch_size = 1024, hidden_layers = 2,
        neurons_per_layer = 64, architecture = Architecture.LSTM.value,
        dropout = 0.1, is_bidirectional = False, initial_learn_rate = 0.001,
        early_stop_patience = 6, is_classification=False):

        ## Param member vars
        self.max_epochs = max_epochs
        self.batch_size = batch_size
        self.hidden_layers = hidden_layers
        self.neurons_per_layer = neurons_per_layer
        self.architecture = architecture
        self.dropout = dropout
        self.is_bidirectional = is_bidirectional
        self.initial_learn_rate = initial_learn_rate
```

```
        self.seq_info = seq_info
        self.is_classification = is_classification
        self.early_stop_patience = early_stop_patience
        self.train_time = 0

        self.train_x = train_x
        self.train_y = train_y
        self.validation_x = validation_x
        self.validation_y = validation_y

        ## Other member vars
        self.model = Sequential()
        self.training_history = None
        self.score: dict = {}

        self._create_model()


    ### PUBLIC FUNCTIONS

    def get_model(self):
        return self.model

    def train(self):
        start = time.time()
        early_stop = EarlyStopping(monitor='val_loss',
patience=self.early_stop_patience, restore_best_weights=True)
        tensorboard =
TensorBoard(log_dir=f"{os.environ['WORKSPACE']}/logs/{self.seq_info}__{self.ge
t_model_info_str()}__{datetime.now().timestamp()}")

        # Train model
        self.training_history = self.model.fit(
            self.train_x, self.train_y,
            batch_size=self.batch_size,
            epochs=self.max_epochs,
            validation_data=(self.validation_x, self.validation_y),
            callbacks=[tensorboard, early_stop],
            shuffle=True
        )

        # Score model
        self.score = self.model.evaluate(self.validation_x, self.validation_y,
verbose=0)
        self.score = {out: self.score[i] for i, out in
enumerate(self.model.metrics_names)}
        print('Scores:', self.score)
        end = time.time()
        self.train_time = end - start


    def save_model(self):
        self._save_model_config()
        self._save_model_weights()

    def get_model_info_str(self):
```

```
        return f"{'Bi' if self.is_bidirectional else
''}{self.architecture.__name__}-HidLayers{self.hidden_layers}-
Neurons{self.neurons_per_layer}-Bat{self.batch_size}-Drop{self.dropout}"

    ### PRIVATE FUNCTIONS

    def _create_model(self):
        """
        Creates and compiles the model
        """
        self._use_gpu_if_available()

        ##### Create the model ####
        self.model = Sequential()

        if self.is_bidirectional:

self.model.add(Bidirectional(self.architecture(self.neurons_per_layer,
input_shape=(self.train_x.shape[1:]), return_sequences=True)))
        else:
            self.model.add(self.architecture(self.neurons_per_layer,
input_shape=(self.train_x.shape[1:]), return_sequences=True))
        self.model.add(Dropout(self.dropout))
        self.model.add(BatchNormalization())

        for i in range(self.hidden_layers):
            return_sequences = i != self.hidden_layers - 1 # False on last
iter
            if self.is_bidirectional:

self.model.add(Bidirectional(self.architecture(self.neurons_per_layer,
return_sequences=return_sequences)))
            else:
                self.model.add(self.architecture(self.neurons_per_layer,
return_sequences=return_sequences))
            self.model.add(Dropout(self.dropout))
            self.model.add(BatchNormalization())

        if self.is_classification:
            self.model.add(Dense(2, activation="sigmoid"))
        else:
            self.model.add(Dense(1))

        adam = tf.keras.optimizers.Adam(learning_rate=self.initial_learn_rate)


        if self.is_classification:
            self.model.compile(
                loss="sparse_categorical_crossentropy",
                optimizer=adam,
                metrics=["sparse_categorical_crossentropy", "accuracy"]
            )
        else:
            self.model.compile(
                loss=RSquaredMetricNeg,
                optimizer=adam,
```

```
                metrics=["mae", RSquaredMetric]
            )


    def _use_gpu_if_available(self):
        ## Utilise GPU if GPU is available
        local_devices = device_lib.list_local_devices()
        gpus = [x.name for x in local_devices if x.device_type == 'GPU']
        if len(gpus) != 0:
            if self.architecture == GRU:
                self.architecture = CuDNNGRU
            elif self.architecture == LSTM:
                self.architecture = CuDNNLSTM


    def _save_model_weights(self):
        file_path = ""
        if self.is_classification:
            file_path =
f"{os.environ['WORKSPACE']}/models/final/{self.seq_info}__{self.get_model_info
_str()}__{self.max_epochs}-
{self.score['sparse_categorical_crossentropy']:.3f}.h5"
        else:
            file_path =
f"{os.environ['WORKSPACE']}/models/final/{self.seq_info}__{self.get_model_info
_str()}__{self.max_epochs}-{self.score['RSquaredMetric']:.3f}.h5"
        self.model.save_weights(file_path)
        print(f"Saved model weights to: {file_path}")

    def _save_model_config(self):
        json_config = self.model.to_json()
        file_path =
f'{os.environ["WORKSPACE"]}/model_config/{self.get_model_info_str()}.json'
        with open(file_path, "w+") as file:
            file.write(json_config)
        print(f"Saved model config to: {file_path}")
```

2.4 Indicator Correlation Reduction

```
def reduce_correlation_matrix(correlations: pd.DataFrame, reduction_size:
int):
    best_indicators: List[str] = []

    correlations = correlations.abs()
    correlations_original = correlations.copy()

    cor = correlations.to_numpy()
    row_sums = np.sum(cor, axis=1)
    min_row = np.argmin(row_sums)
    best_indicators.append(correlations.columns[min_row])

    correlations.drop(correlations.index[min_row], axis="index", inplace=True)
```

```
    correlations.drop(correlations.columns[min_row], axis="columns",
inplace=True)

    while len(best_indicators) < reduction_size:
        row_sums = []
        for index, row in correlations.iterrows():
            row_sums.append(correlations_original.loc[index,
best_indicators].sum())
        min_row = np.argmin(row_sums)
        ind = correlations.columns[min_row]
        if ind not in best_indicators:
            best_indicators.append(ind)
            correlations.drop(correlations.index[min_row], axis="index",
inplace=True)
            correlations.drop(correlations.columns[min_row], axis="columns",
inplace=True)

    ret = correlations_original.loc[best_indicators, best_indicators]
    return ret
```