WARWICK BUSINESS SCHOOL
THE UNIVERSITY OF WARWICK

Masters Programmes

Dissertation Cover Sheet

Degree Course: MSc Mathematical Finance

Student ID Number:

Title: Robustness of Sequence Models in Deep Learning

Dissertation Code: ST9150

Submission Deadline: 8th September 2020 20:00 (BST)

Word Count: 9873

Number of Pages: 59

"I declare that this work is entirely my own in accordance with the University's Regulation 11 and the WBS guidelines on plagiarism and collusion. All external references and sources are clearly acknowledged and identified within the contents.

No substantial part(s) of the work submitted here has also been submitted by me in other assessments for accredited courses of study, and I acknowledge that if this has been done it may result in me being reported for self-plagiarism and an appropriate reduction in marks may be made when marking this piece of work."

Warwick Business School
The University of Warwick
Coventry CV4 7AL
United Kingdom

wbs.ac.uk
+44 (0)24 7652 4306
enquiries@wbs.ac.uk

Robustness of Sequence Models in Deep Learning

A dissertation presented for the degree of MSc Mathematical Finance

Supervisor: Dr Martin Lotz

Denis O'Driscoll

Warwick Business School

University of Warwick

United Kingdom

September 8, 2021

I declare that this work is entirely my own in accordance with the University's Regulation 11 and the WBS guidelines on plagiarism and collusion. All external references and sources are clearly acknowledged and identified within the contents.

No substantial part(s) of the work submitted here has also been submitted by me in other assessments for accredited courses of study, and I acknowledge that if this has been done it may result in me being reported for self-plagiarism and an appropriate reduction in marks may be made when marking this piece of work.

Abstract

The robustness of deep learning methods for finance is an exciting area of research currently in its infancy as the adoption of deep learning becomes more widespread. This study aims to investigate the robustness of a recurrent neural network with Long Short-Term Memory (LSTM) architecture that makes feasible predictions for daily Bitcoin closing prices. This is achieved by first replicating and later modifying an LSTM architecture documented in the relevant literature and then focusing on the robustness of the final network created. The robustness of the network is examined by training the network with training data that has been perturbed by Gaussian noise and investigating the effect this has on out-of-sample predictions. Additional robustness tests are conducted by examining how adding Gaussian noise layers and noisy dense layers to the network affect training accuracy and out-of-sample predictions. This work found that the LSTM network was especially robust to random perturbations within the data, but prediction error, measured by root-mean-square error, increased when Gaussian noise and noisy dense layers were added to the model. Comparing the robustness of the LSTM with an ARIMA model to random noise within the training data, it was also found that the ARIMA model was especially susceptible to noise within the data compared to the LSTM. These results suggest that deep learning methods are more robust than traditional linear methods but are not infallible and are still susceptible to perturbations within the model, often tough to recognise due to the black-box nature of these models. On this basis, the robustness of the deep learning network with LSTM architecture is impressive. However, one should still always test their network in a variety of different ways as every network behaves differently.

Acknowledgements

I would like to express my sincere thanks to my supervisor Dr Martin Lotz for his time and advice while assisting me with the dissertation over the last few months. I would also like to thank the faculty of WBS and lecturers of the MSMF for their effort and assistance in delivering and managing the course during these difficult times. I would also like to thank my parents for their support throughout the year.

Contents

List of Figures

List of Tables

1 Introduction

In the world today, many artificial intelligence (AI) systems find their basis in machine learning. Machine learning describes 'the capacity of systems to learn from problem-specific training data to automate the process of analytical model building and solve associated tasks.' (Janiesch et al., 2021, p. 1). Deep learning is a branch of machine learning that utilises artificial neural networks (ANNs) as the framework used to solve a particular problem. Although often used interchangeably to describe similar concepts, we will distinguish between AI, machine learning and deep learning for the sake of clarity. AI generally describes the creation of processes to replicate or outperform human actions or performance in carrying out tasks or making decisions, often in complicated situations. To distinguish between machine learning, shallow machine learning, and deep learning, the aforementioned authors created the Venn diagram seen in Figure 1. Deep neural networks are more complex networks, with multiple hidden layers and advanced infrastructures within these layers than basic one-layer neural networks. However, the authors admit that this distinction is not ubiquitous in the literature, hence the dashed line in the diagram. For this study, we will use this distinction. Sequence models in the context of deep learning and this work are simply models that generate a sequence of outputs from a sequence of inputs.

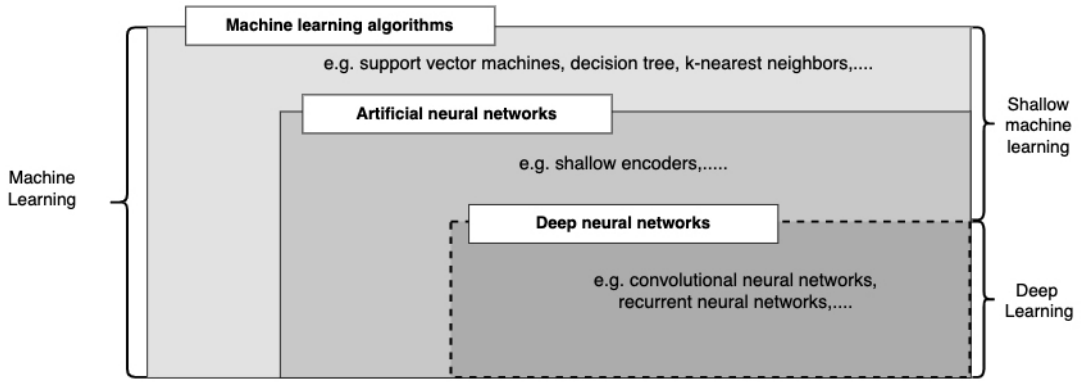

Figure 1: Machine Learning Distinctions (Janiesch et al., 2021).

Self-driving cars, computer vision, and natural language processing (NLP) are some of the most well-known deep learning applications today. To the end-user, the implementation and functionality of these deep learning technologies are often a mystery. Therefore, a team of data scientists and software engineers who create deep learning systems such as that in a self-driving car are responsible for ensuring that these systems are accurate, robust, and avoid unfavourable outcomes, e.g. an autonomous vehicle crashing, in practice where many random external factors and data are affecting model behaviour. The same careful consideration of random perturbations

in the underlying data and model behaviour must also be implemented in financial applications of deep learning. If important financial decisions risking vast amounts of capital are made based on deep learning models, these decisions and their success rely heavily on the robustness of the model and the data used to create it.

The main goal of this work is to explore how robust a deep learning network, specifically a recurrent neural network (RNN) with Long Short-Term Memory (LSTM) architecture, is in a financial prediction setting. Numerous RNNs will be created with varying infrastructures tasked with predicting Bitcoin (BTC) closing prices for the next five days in the future from the previous five-day closing prices and volume traded. The initial LSTM created in this work is similar to that found in Mehtab et al. (2020), which is subsequently modified for our purposes. The models will be trained using training data created from 1247 days' worth of data from 01/01/2018 to 31/05/2021. Different models will be experimented with to create a model that makes feasible predictions where this final model will then be scrutinised to address the primary goal of this work.

To test the robustness of our proposed LSTM network, we will investigate how sensitive the model created is to small perturbations or adversarial attacks in the data used for training the models that may be undetectable by humans and also how perturbations within the model affect model prediction accuracy. The term adversarial attack stems from the term adversarial example used in the context of neural networks by Szegedy et al. (2014). Adversarial examples were defined as imperceptible perturbations added to a test image to unfavourably change the prediction made by a neural network for an object recognition task. These adversarial examples could then be generated to "attack" a neural network, leading to the network making unexpected and incorrect predictions. To investigate the robustness of the models created as part of this work, we will generate adversarial examples and perturbations by adding Gaussian noise to the data used to train the network and by adding different kinds of noisy layers to the network. The effects of these perturbations on the training period and post-training prediction accuracy of the model on out-of-sample data will be analysed and reviewed. We will also compare the accuracy and robustness of the RNN created in this work to that of a typical ARIMA model tasked with predicting BTC closing prices.

The literature review for this work immediately follows the introduction. The literature review outlines applications of neural networks, the efficient market hypothesis in the context of Bitcoin efficiency, and current studies focused on the robustness of deep learning methods in varying contexts. Chapter 3 describes the data used in the study along with the data windowing process used when building the models outlined in this work. Chapters 4 and 5 give brief introductions to neural networks and recurrent neural networks. Chapter 5 also describes the LSTM architecture

used in this study. Chapter 6 outlines the model building process carried out as part of this study to create a final model that we will use to test the robustness of a deep learning architecture. Chapter 7 describes the robustness tests carried out in this work, with the results of these tests found in Chapter 8. The final chapter includes our conclusion, criticisms of the study, future directions, and implications of this study.

2 Literature Review

Deep learning has been effectively used to achieve impressive results for applications in science, technology and finance. Neural networks have been implemented to predict organic chemistry reactions (Wei et al., 2016), estimate soil erosion (Kim and Gilley, 2008), drive autonomous vehicles (Kocic et al., 2019), carry out pattern recognition (Yun et al., 2018), and for stochastic control in finance (Germain et al., 2021). Google's AI subsidiary, DeepMind, recently gained attention from mainstream media for creating AlphaFold, which utilises deep learning to solve a 50-year-old problem within biology. AlphaFold uses neural networks to predict protein structures with a degree of accuracy far and above other computational methods applied to the same problem (Senior et al., 2020). These are only a select few examples of the applications of deep learning that do not even scratch the surface of volumes of work being produced in the field.

Neural networks are a popular choice with machine learning practitioners for many reasons. Non-linear interactions within the training data are accounted for, which can increase in-sample fit compared to traditional models. Input data can be expanded to include potential predictors of relevance to the prediction problem (Heaton et al., 2018). Further, the advancement of computational power, the introduction of the backpropagation algorithm by Rumelhart et al. (1986), and libraries such as TensorFlow, Scikit-learn and PyTorch for programming languages such as Python have allowed non-expert programmers to implement deep learning methods efficiently with relative ease. However, practitioners should still be extremely familiar with the functionality, shortfalls, and mathematical framework of the networks they are building, which may not always be the case.

As powerful as they may be in a wide variety of contexts, neural networks are by no means perfect and have their limitations. Neural networks, especially those that are deep, can be extremely computationally expensive (Thompson et al., 2020). AlphaFold, mentioned previously, required vast amounts of data during a model training process that took weeks despite using extreme supercomputing power. Another flaw of deep learning models is their often described "black-box" nature. Fan et al. (2021, p. 1) remark that 'deep learning works as a black-box model in the sense that although deep learning performs quite well in practice, it is difficult to explain its underlying mechanism and behaviours'. This is due to the fact that when we train a deep network, it may have hundreds if not thousands of parameters used to make predictions. These parameters are often hard to interpret with little meaning versus traditional linear methods such as ARIMA models for time series modelling. ARIMA models typically have fewer parameters which also have clear definitions and properties within the model. It can also be hard to discern why a neural network you create performs better than another similar network when making predictions without expert knowledge of the network's mathematical underpinning.

The efficient-market hypothesis (EMH) states that that the current price of an asset reflects all the information available regarding that asset that may affect its price. The EMH is typically divided into three forms: strong, semi-strong and weak. The weak EMH asserts that historical prices do not affect future prices or that historical prices of an asset cannot be used in such a way to generate profits by predicting future prices. One of the first studies on the efficiency of Bitcoin concluded that the Bitcoin market was not weakly efficient over their entire sample, but when splitting the sample between 2010-2013 and 2013-2016, found that in the later period, the market was becoming slightly more efficient (Urquhart, 2016). The authors suggest that this could be due to the interest and knowledge of Bitcoin increasing among investors in the latter time frame. Researchers have been interested in the efficiency of Bitcoin and cryptocurrency markets, often coming to contradictory conclusions. Cheah et al. (2018), Hu et al. (2019), and Jiang et al. (2018) examine Bitcoin and cryptocurrency markets, concluding that they exhibit market inefficiency through varying experiments and methods. However, Sensoy (2019) asserts that BTCEUR and BTCUSD markets have become more informationally efficient since 2016, with a strong positive relationship between liquidity and high-frequency Bitcoin efficiency. A study in response to Urquhart (2016) showed that 'an odd integer power of Bitcoin returns are largely weakly efficient over the full period as well as over the two sub-sample periods.' (Nadarajah and Chu, 2017, p.3). Since the literature does not fully agree on the inefficiency of Bitcoin and cryptocurrencies in general, this motivated the experiments carried out in this work in which a model tasked with predicting cryptocurrency prices from historical prices and volumes traded is created and scrutinised.

Recurrent neural networks are specifically designed for data that is structured sequentially, e.g. time series data and speech data. RNNs are neural networks that have memory, which is the key differentiator between them and simple feedforward neural networks. A common choice of RNN is the Long Short-Term Memory (LSTM) network. Hochreiter and Schmidhuber (1997) introduced LSTM as a method of addressing the vanishing or exploding gradient problem that can plague some neural networks and found that LSTMs learn faster than other methods such as backpropagation. RNNs are often more flexible and suited to modelling and forecasting time series compared to standard linear methods (Petneházi, 2019). As a result, RNNs and LSTM networks are frequently used within a financial setting as they are well suited for financial time series data. Chen et al. (2021) utilised LSTM within an RNN to estimate macroeconomic state processes from time series data as part of a larger asset pricing network for individual stock returns.

Although there are plenty of examples of applications of deep neural networks and their often impressive results, the amount of literature focused on the robustness of these networks is

light in comparison. As mentioned previously, one of the earliest studies formalising the use of adversarial examples to test the robustness of deep learning networks was carried out by Szegedy et al. (2014). In this work, the authors apply an 'imperceptible non-random perturbation to a test image', which can unfavourably change the prediction output of a deep neural network built for object recognition. These adversarial examples are found by maximising the network's prediction error while remaining undetectable to humans. Furthermore, the authors found that an adversarial example that caused an incorrect prediction for a specific neural network was also difficult for a similar but different network to correctly predict, even if the different networks had different hyperparameters or were trained using different subsets of the data. The authors suggest that the weakness of the networks to these examples may come from the fact that these adversarial examples occur with such low probability that they are rarely observed during the model training or testing phase. Göpfert et al. (2020) also investigate adversarial examples within a convolutional neural network (CNN), another deep learning framework. The authors created the Entropy-based Iterative Method (EbIM), which creates adversarial attacks for a CNN that are also extremely difficult to detect by humans even if the magnitude of the perturbation is large. The authors conclude that deep learning adversarial attacks would remain a potential risk requiring further investigation. Bastani et al. (2017) propose some metrics to quantify robustness while also admitting that existing approaches that attempt to improve robustness can result in overfitting to the adversarial examples one can generate to test their network's vulnerabilities.

Ko et al. (2019) propose the Propagated output Quantified Robustness for RNNs (POPQORN) algorithm to quantify the robustness of RNNs such as LSTMs and gated recurrent units (GRUs). One experiment in this work focuses on the robustness in an NLP context. The POPQORN algorithm could successfully compute certified lower bounds and guarantee lower bounds on the minimum distortion caused by adversarial inputs. Cheng et al. (2020) recognise that most of the work focused on adversarial attacks focuses on image classification while also admitting that the adversarial attack problem is more challenging for sequence-to-sequence models in a deep learning context. The difficulty arises due to the potentially infinite number of output sequences in settings such as predicting the next word in a sentence given the previous words. The authors create an adversarial attack framework called 'Seq2Sick', which aims to be similar to the original input sequence but successfully produces unexpected and incorrect output sequences. The authors conclude that sequence-to-sequence models are more robust to adversarial attacks than deep neural networks for image classification.

With the majority of the investigation of adversarial attacks and the robustness of deep learning networks being applied to image classification and NLP, less work has focused on adversarial

attacks and robustness for deep learning sequence models with financial applications. This is because this is generally an extremely new area of interest. Secondly, practitioners who can create an accurate, state-of-the-art deep neural network for financial time series prediction that is also extremely robust and immune to adversarial attacks would be foolish not to keep their models a secret as these models could be highly lucrative. An attempt to explore adversarial attacks in a high-frequency trading setting is made by Goldblum et al. (2020). Initially, the authors attempt to predict stock prices ten seconds into the future from size weighted average prices using order book data from the previous minute. The authors propose different potential ways of creating adversarial attacks. The most feasible is an attack that 'works on a large number of historical training snippets with the hope that it transfers to unseen testing snippets.' The authors found that small perturbations that were small in size relative to the order book could trick their model quite regularly. They also found that the adversarial patterns they created that fooled one model often fooled others which is a phenomenon we have mentioned previously, although in a different setting. However, this work is not perfect, with several reviewers highlighting issues with the authors' algorithms and failures to describe the bounds on the size of the perturbations concretely.

3 Data

3.1 Data Description

Cryptocurrency data used in this research contains daily Bitcoin (BTC) close prices in United States Dollar (USD) and volume data in BTC from the Binance exchange. The first data point is from 01/01/2018, with the last data point from 31/05/2021, resulting in 1247 days' worth of data. This cryptocurrency was chosen as it is the most popular cryptocurrency in terms of volume traded, and Binance being the world's largest cryptocurrency exchange in the world in terms of volume traded. The data was obtained from `www.cryptodatadownload.com`, which offers vast amounts of cryptocurrency data from the world's largest exchanges free of charge for personal or educational use. Plots of the BTC closing prices in USD and volume traded in BTC for the entire data set are seen below in Figure 2.

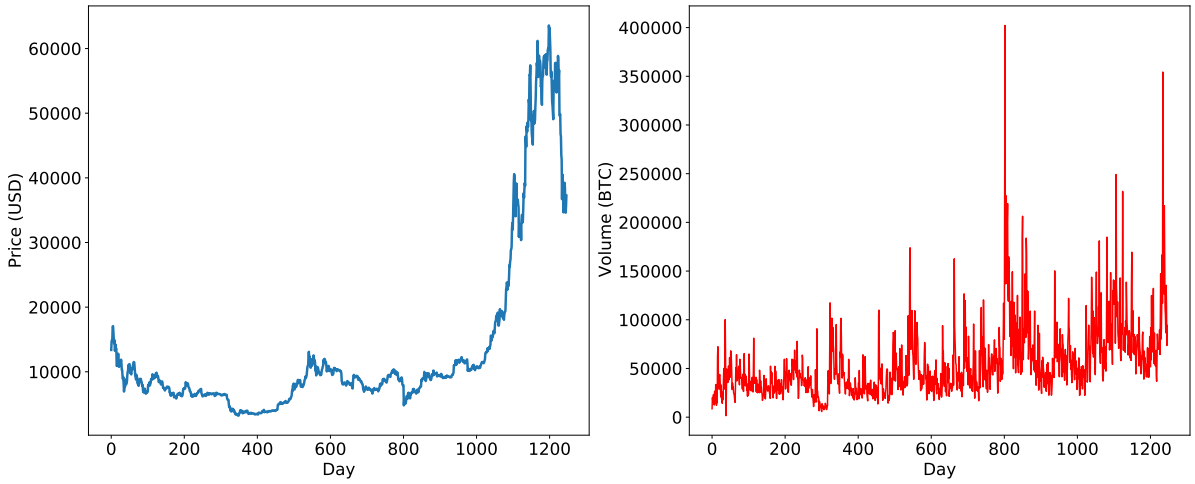

Figure 2: **Left**: BTC Close Prices (USD). **Right**: Volume Traded (BTC).

We focus on cryptocurrency in this work due to its potential market inefficiency mentioned previously. As a result, the neural networks created in this work may find long-term patterns within the data to predict future prices with an acceptable degree of accuracy. We also chose to focus on cryptocurrency as it is a rapidly expanding area of finance, with many previously sceptical investors flocking to the new asset class in recent years. Additionally, cryptocurrency markets are notorious for exhibiting extreme levels of volatility where in the past, prices fell by 12% after an unexpected tweet from an influential billionaire was published. Deep learning methods may recognise pricing patterns in the data despite the volatility, better than traditional linear models.

3.2 Data Processing and Windowing

First, we will split the data into three distinct blocks. We will take the first 90% of the data beginning 01/01/2018 and use this as our training dataset. The remaining 10% will be divided further, with two-thirds of this data being our validation dataset and the remaining third being our test dataset. We will then scale the data using the MinMax scaler with a range of zero and one. Further details of the scaling process can be found in Appendix A. After scaling, we window the data in each of the three blocks into arrays of consecutive five-day inputs, where we use five-day closing prices and volumes to predict the next five-day closing prices of the asset, which are then compared to the five-day actual closing prices, referred to here as the outputs. A visual representation of the data windowing for the first ten days of the training dataset is seen in Figure 3, where each orange block represents the closing price and volume traded at time t, and each blue block represents the closing price at time t. This is then repeated for the validation and test data. The training data will be used to fit and train the network. The validation set is not involved in fitting the model, but it allows us to see how well our model can generalise during the training process. Finally, the test dataset, also referred to as the unseen data, will not be used during the training of the model and will be used to make predictions based on the trained model's parameters and to evaluate how well our trained model can make predictions on out-of-sample data.

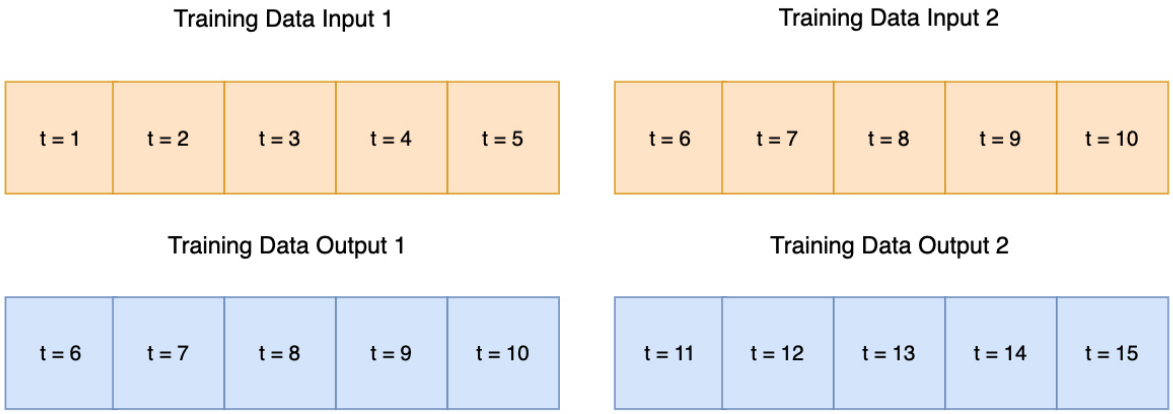

Figure 3: Data Windowing Process for Training Data.

4 Artificial Neural Networks

4.1 Background

Neural networks were originally proposed by McCulloch and Pitts (1943). Frank Rosenblatt later created the perceptron to advance work in pattern recognition (Rosenblatt, 1958). Although the initial interest and mathematical modelling pursuits of neural networks were from a biological standpoint, ANNs are now used in many fields unrelated to biology. Multilayer perceptrons (MLPs) (Rumelhart et al., 1986), also referred to as feedforward networks, are viewed as the most basic and common form of ANN.

At the simplest level, MLPs are a network of nodes, also called neurons or units, that are connected via weights. A typical network consists of an input layer, multiple hidden layers, and an output layer. By providing an input to some or all of the nodes, the network is activated. This activation is then spread through the layers via the weighted connections to the output layer. This process is known as the forward pass of the network. Neural networks consist of L layers. The variables at each layer are functions of those in the previous layer and are used to construct features for the variables in the next layer. The weighted connections within the model, often called weights, are initially unknown, and our goal is to find these weights that make the model a good fit to the training data. This is achieved by measuring the fit to the training data with respect to a loss function. Figure 4 is a simple example of a neural network with an input later containing three input neurons, one hidden layer with two neurons and an output layer with a single output neuron. The weighted connections are represented by the black arrows. The hidden layer in Figure 4 is also referred to as a dense layer as both neurons are connected to every neuron in the input layer. The following brief mathematical description of neural networks follows Bishop (2006) and Franklin (2005), which can be referred to for more detail.

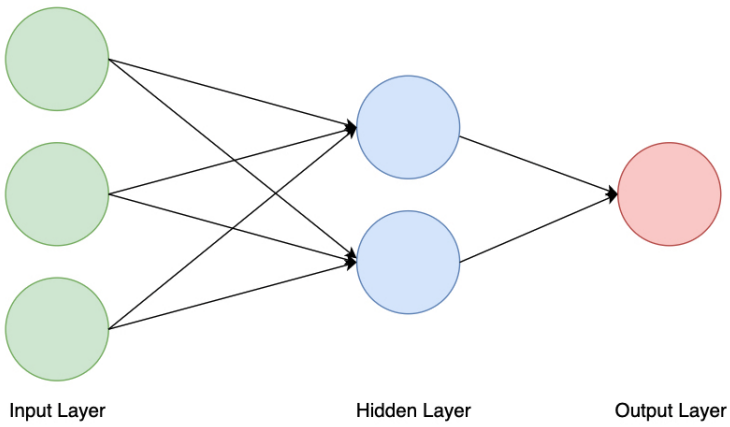

Figure 4: A Simple Neural Network.

4.2 Activation Functions

The input layer consists of the input vector $\mathbf{X} = x^{(1)},...,x^{(p)}$ where we have p variables or features we are interested in using for prediction, with n data points for each $x^{(i)}$. Let there be m_l variables in each layer with $m_0 = m$. Let $A_l^{(k)}$ be the kth variable in the lth layer where each variable A is known as an activation. Activation functions, σ_l, are used in neural networks to transform the input of a neural network, expressed as a linear combination of weights and bias, to an output in feature space. Each neuron in the first hidden layer calculates a weighted sum of the network inputs $\mathbf{X}$. Hence $A_0^{(k)} = x^{(k)}$. Then at all layers

$$Z_l^{(k)} = \sigma_l(A_l^{(k)}), \tag{1}$$

$$Z_l = (Z_l^1, ..., Z_l^{m_l}). \tag{2}$$

Then at the lth layer where $1 \leq l \leq L$

$$A_l^{(k)} = Z_{l-1}^T \beta_l^k + \beta_l^{(k,0)} \tag{3}$$

where β_l^k is a vector of weights of length m_l and $\beta_l^{(k,0)}$ is the bias.

The process of calculating the summation and activation can then be repeated for each of the hidden layers. A popular activation function in the context of deep learning is rectified linear unit (ReLU) (Nair and Hinton, 2010). The ReLU function is a simple yet powerful function in the context of neural networks. The definition of the ReLU function is seen below.

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Since its introduction, the ReLU activation function has proven to be one of the most popular and powerful activation functions used today (Nwankpa et al., 2018). Compared to other popular activation functions such as the sigmoid and tanh functions, the ReLU can offer improved performance, especially when using gradient descent learning methods. A study by Zeiler et al. (2013) investigated the performance of the ReLU in speech processing for a deep neural network concluded that ReLU activation units 'are easier to optimise, converge faster, generalise better, and are faster to compute'.

An extension to the ReLU is the Leaky ReLU proposed by Maas et al. (2013). The Leaky ReLU

is defined as follows for $0 \leq \alpha \leq 1$

$$f(\alpha, x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise.} \end{cases} \tag{5}$$

The standard ReLU activation function can suffer from an issue known as "dying ReLU" since it is zero for all negative values. Neurons that always output zero are called "dead", leading to the updating weights not being activated during the training process, as a dead neuron leads to zero activation (Nwankpa et al., 2018). Maas et al. (2013) fix alpha to a small value such as 0.01, which prevents the dying neuron problem by preventing the gradient from being zero at any time during the training process.

Most importantly for most activation functions is that they are almost differentiable everywhere, which allows the network to be trained using gradient descent. This also allows us to use one activation function for neurons in one layer and use a different activation function in another layer if we desire.

4.3 Output Layers

The output vector $\mathbf{Y}$ of an MLP is obtained by the activation of the neurons in the output layer. Typically, there is one output variable, which would be similar to the single Z variable in that layer. Hence for a network with L hidden layers

$$Y = \sigma_L(Z_{L-1}^T \beta_L^k + \beta_L^0), \tag{6}$$

where σ_L is the activation function for the Lth layer.

4.4 Loss and Risk

Let $\mathbf{X}$ and $\mathbf{Y}$ be vectors that have outcomes in the spaces $\mathcal{X}$ and $\mathcal{Y}$ respectively. Let $f : \mathcal{X} \to \mathcal{Y}$ be a function from $\mathcal{X}$ to $\mathcal{Y}$. Our aim is to find an estimate $\hat{f} : \mathcal{X} \to \mathcal{Y}$. That is, given some predictors $\mathbf{X} \in \mathcal{X}$ we want to make accurate predictions of $\mathbf{Y} \in \mathcal{Y}$ by using $\hat{f}(\mathbf{X})$, we may also refer to $\hat{f}(\mathbf{X})$ as $\hat{Y}$. A loss function is any function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ used to measure the difference between values of $\mathbf{Y}$ and predictions $f(\mathbf{X})$. Loss functions are used to optimise the parameters in a neural network. The risk, $R : \mathcal{F} \to \mathbb{R}_{\geq 0}$, associated with the loss function $\mathcal{L}$ and function $f : \mathcal{X} \to \mathcal{Y}$ is given by

$$R(f) = \mathbb{E}_{\mathbf{X},\mathbf{Y}}[\mathcal{L}(Y, f(X))]. \tag{7}$$

Our training data $\mathcal{T} = (X_i, Y_i)_{i=1}^n$ is a sample from the joint distribution of $\mathbf{X}$ and $\mathbf{Y}$. Thus the empirical risk $\hat{R}$ is an estimate of R based on the sample $\mathcal{T}$.

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(Y_i, f(X_i)). \tag{8}$$

Risk gives us the average loss over the joint distribution of $\mathbf{X}$ and $\mathbf{Y}$. R is a deterministic function, but $\hat{R}$ is random since it depends on the random sample used to train the model. From the law of large numbers, for any f as $n \to \infty$,

$$\hat{R}(f) \to R(f). \tag{9}$$

In practice we want to find the f that minimises the empirical risk i.e.

$$\hat{f} = \underset{f \in \hat{\mathcal{F}}}{\operatorname{argmin}} \hat{R}(f). \tag{10}$$

By minimising the empirical risk we obtain a function $\hat{f}$ that maximises the accuracy of our predictions $\hat{f}(\mathbf{X})$ compared to the true values $\mathbf{Y}$ for the training data $\mathcal{T}$. In tasks such as our task of creating a model to predict cryptocurrency prices based on a model trained on previous prices we use a mean squared error (MSE) loss function.

$$\mathcal{L}(\mathbf{Y}, f(\mathbf{X}, \theta)) = \frac{1}{2} \sum_{n=1}^{N} ||f(\mathbf{X}_n, \theta) - \mathbf{Y}||^2 \tag{11}$$

where f is the output of the network, $\mathbf{X}$ are the input vectors, θ is a vector of the neural network's weights, and $\mathbf{Y}$ is the corresponding target vector.

4.5 Gradient Descent and Backpropagation

When our loss function $\mathcal{L}$ is differentiable almost everywhere, we use gradient descent to minimise the empirical risk, which involves backpropagation which is an implementation of the chain rule. The backpropagation algorithm allows for ANNs and MLPs to be trained using gradient descent. Gradient descent involves finding the derivative of the loss function with respect to the network weights and then altering the weights in the direction of the negative slope to minimise the loss function. By minimising the loss function, we are minimising the mean squared error of our predictions using our training data and the true values of what we are trying to predict. Backpropagation refers to how we compute the gradient while gradient descent performs the learning using this gradient. We will not concern ourselves with the minute details of gradient descent and backpropagation and its mathematical implementation for a general

loss function as it is not the focus of this work. An in-depth review of backpropagation and gradient descent can be found in Bishop (2006) and Staudemeyer and Morris (2019, p.7-11).

5 Recurrent Neural Networks

5.1 Background

Recurrent neural networks are extremely useful for processing time series or sequences of data where the order of the data is important. RNNs are powerful models due to their ability to scale to much longer sequences than regular neural networks (Guida, 2019). RNNs have a "memory" in the sense that the inputs and outputs are influenced by the current input. A key aspect of RNNs is that parameters are shared across each layer as well as sharing the same weight parameter within each layer. While standard MLPs have different weights across the nodes. RNNs can also learn what data was shown to them previously and determine how relevant they are to making predictions. A basic example of an RNN can be seen in Figure 5.

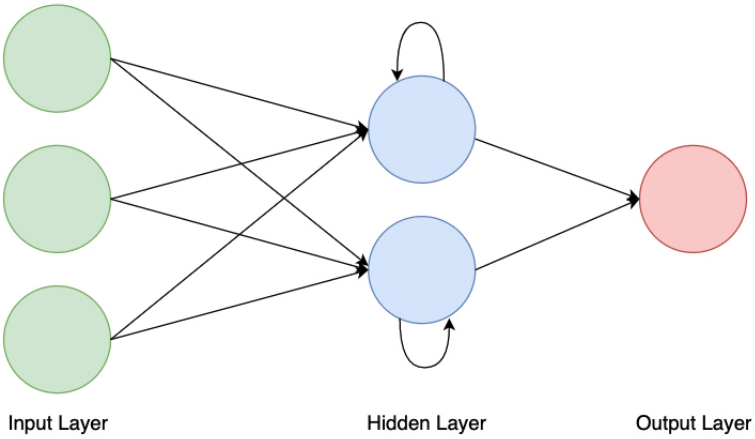

Figure 5: A Simple Recurrent Neural Network.

RNNs are typically trained using a method known as Backpropagation Through Time (BPTT). BPTT is similar to backpropagation for feedforward neural networks. A description of BPTT for RNNs and LSTMs can be found in Staudemeyer and Morris (2019, p. 13-17). For the sake of this study, we will concentrate on the Long Short-Term Memory RNN architecture. We choose the LSTM as our RNN architecture of choice as 'the LSTM is designed to find hidden state processes allowing for lags of unknown and potentially long duration in the time series.' (Chen et al., 2021, p. 17). Malhotra et al. (2015, p. 1) also remarked, 'Because of this ability to learn long-term correlations in a sequence, LSTM networks are capable of accurately modelling complex multivariate sequences'.

5.2 Long Short-Term Memory

The LSTM network was originally proposed in Hochreiter and Schmidhuber (1997). The LSTM network is a widely popular RNN architecture due to its superior ability to capture short and long term trends within the data provided to it (Guida, 2019). As mentioned previously, the LSTM network was introduced as a solution to the vanishing and exploding gradient issue that can occur in standard RNNs. The following description of LSTM mostly follows that of Guida (2019) and Graves (2012). Simple RNNs have long term memory in the form of the weights, which change during the training phase of the model while also having short term memory from the activations passed between units. The LSTM acts as an intermediary storage mechanism.

An LSTM model is similar to a typical RNN, but each standard neuron in a hidden layer is replaced by a memory block. An LSTM memory block with one cell can be seen in Figure 6. Each block contains input, output and forget gates. Although not part of the original LSTM architecture, the forget gate was proposed by Gers et al. (2000) to allow the LSTM cell to learn to reset itself. This was done to address the issue where the internal state of the original LSTM architecture could break down when the internal state grows constantly. The LSTM with a forget gate has now become the standard. Graves (2012) describes these blocks similarly to memory chips in a computer, where these gates provide write, read and reset functions for the cells. These gates allow LSTM cells to store and access information over long periods of time, which prevents the issue of vanishing gradients (Graves, 2012). Further detail of the calculation within an LSTM cell can be found in Appendix B.

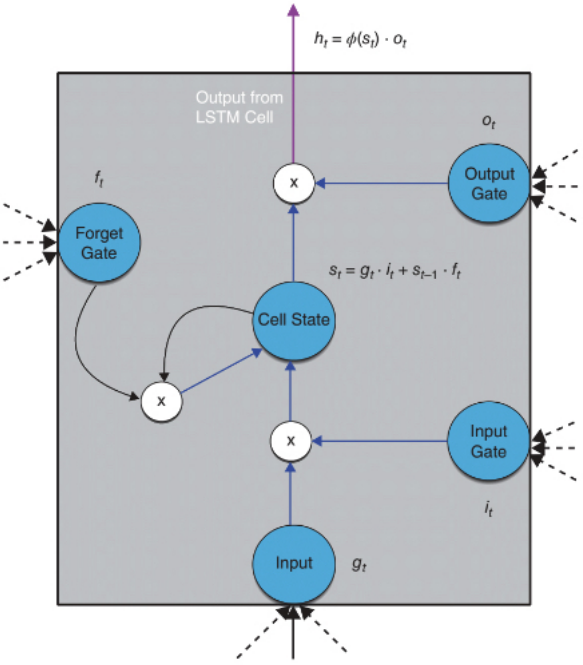

Figure 6: LSTM Memory Block with One Cell (Guida, 2019).

5.3 Model Metrics

As mentioned previously, we will use the MSE as our loss function when training our networks. We will also use the root-mean-squared error (RMSE), the square root of the MSE, as another performance metric. These metrics allow us to assess the accuracy of a model during the training phase and when making out-of-sample predictions. It will also allow us to differentiate between different models during the model building process and evaluate the robustness of the models created, with a lower RMSE being preferred. The RMSE represents the square root of the variance of the difference between predicted and true values.

6 Model Creation Process

6.1 Model Inspiration

The models created in this work for the investigation of robustness are inspired by the models created by Mehtab et al. (2020). The authors used LSTM networks for deep learning based regression to predict NIFTY 50 stock price movements on the National Stock Exchange of India (NSE). In this study, the authors created four different deep learning models. The authors created an RNN with one LSTM layer consisting of 200 nodes and three dense layers with 100, 5, and 5 nodes, respectively. We will initially mimic this infrastructure but will also modify it by adding more LSTM and dense layers and some regularisation to investigate if these additions improve the prediction power from previous models for our purpose. The authors window their data to use the previous five-day data to predict the next five-day prices, which is why we also did this for the sake of consistency. Of course, we can also experiment with different input and prediction window lengths.

We use this study as an influence on the work presented here as it aligns excellently with the overall aim of this study, to test the robustness of a deep learning sequence model in a financial setting. Secondly, the results presented by Mehtab et al. (2020) are extremely impressive. This work shows just how powerful deep learning with LSTM architecture is compared to other machine learning methods applied to the same stock prediction task. Measuring prediction accuracy in terms of root mean squared error (RMSE), the authors found that the LSTM based models outperformed methods such as boosted regression, random forests and SVM regressions. Crucially, the authors do not investigate the robustness of the models they have created in great detail, which is a perfect opportunity for this work to explore the robustness of models that are similar with a similar prediction task in mind while in our case, we are attempting to predict cryptocurrency prices.

6.2 Model Building

Different frameworks were experimented with during the model building process by testing different layer types, numbers of layers, and neurons in each layer in a variety of networks. We will present a select few milestones of the model building process. The model metrics mentioned previously will be used to compare training, validation and out-of-sample accuracy between models. We initially replicated the first LSTM model (LSTM1) created by Mehtab et al. (2020).

The input layer of this model is our five-day windowed training data consisting of close prices and volume traded for each day. Therefore, the shape of the input layer is (5,2). Next, the input data is passed into an LSTM layer with 200 neurons. From here, the output of the LSTM layer

is passed to a dense layer with 100 neurons and then onto another dense layer with five neurons. The output layer is a dense layer with five neurons that produce the five-day ahead predictions for the windowed data we provide to the model. At each layer, the activation function is the ReLU activation function. We use the MSE as the loss function and the ADAM optimiser to train the model for 500 epochs, with a learning rate of 0.00001. A visual representation of the architecture of this model is below in Figure 7.

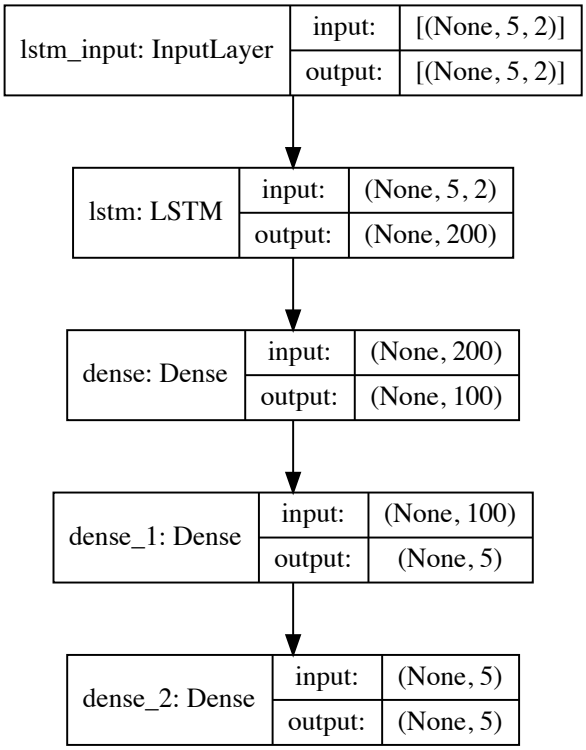

Figure 7: Graphical Representation of LSTM1 Architecture.

Having trained the model, we present plots of the MSE and RMSE of this model below in Figure 8. From Figure 8, there is a stark difference between the loss and RMSE for the training data and validation data. This indicates that the model can fit well to the training data but fails to generalise to the validation dataset. As a result, we continued to try and improve on the performance of this baseline model for our specific case. Although the core aim of this study is not to create a state-of-the-art model for cryptocurrency prediction, we aim to have a model with some satisfactory prediction power to investigate the robustness of a neural network that makes sensible predictions.

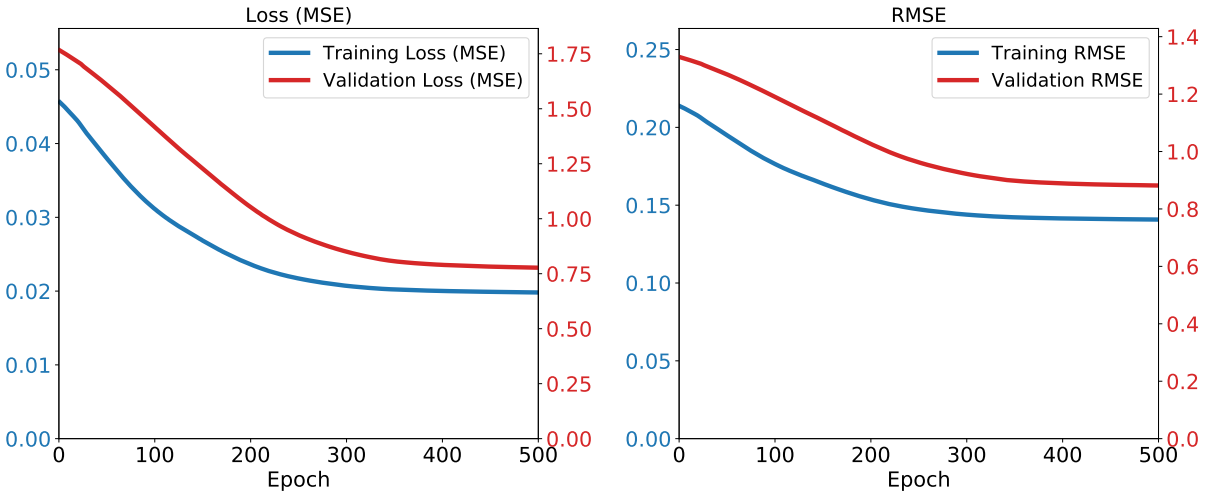

Figure 8: LSTM1 MSE and RMSE for Training and Validation Datasets.

In the quest to improve model performance, we first changed every activation function in the previous model to the Leaky ReLU function using the default α rate in the Keras package of 0.2. We will refer to this network as LSTM2. The overall architecture from the previous model has not changed. This offered some improvement in the RMSE for both the training and validation sets. Plots of the MSE and RMSE for LSTM2 are below in Figure 9. Due to the improved performance offered by the using Leaky ReLU activation function at each layer, it is used as the activation function for each layer in subsequent models.

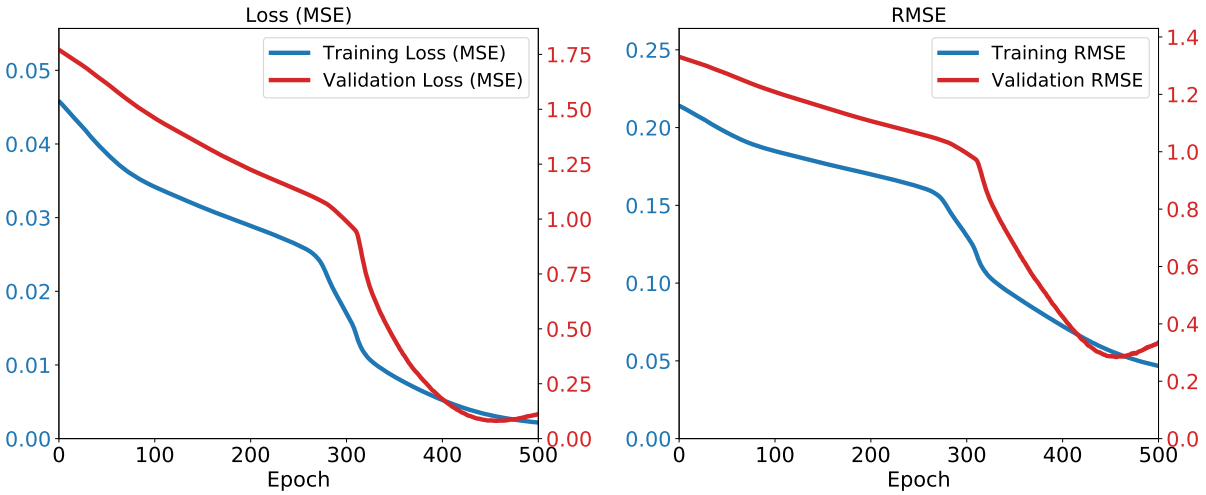

Figure 9: LSTM2 MSE and RMSE for Training and Validation Datasets.

The third model of interest constructed, LSTM3, is similar to LSTM2; however, an extra LSTM layer with 200 neurons was added after the first LSTM layer in the previous model. The architecture of this model can be seen in Appendix D.1. Adding another layer makes the network deeper, which may allow us to capture more minute trends within the data, but this also increases the training time and may cause overfitting. Plots of this model's training and validation loss and RMSE are below in Figure 10. We can see how the addition of the additional LSTM layer improved the accuracy metrics of our deep learning network, hence we kept this LSTM layer in future models.

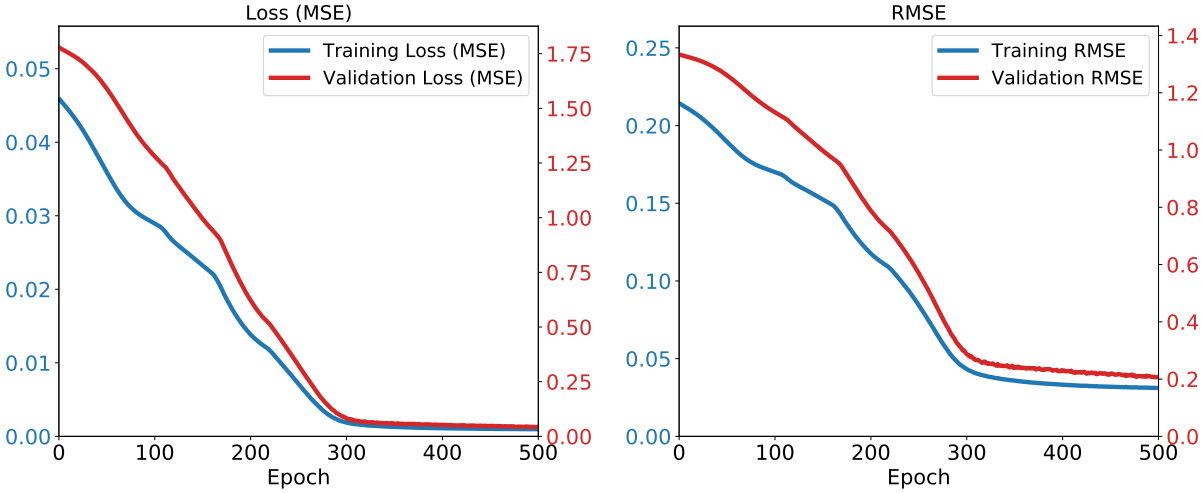

Figure 10: LSTM3 MSE and RMSE for Training and Validation Datasets.

The next significant milestone in the model process involved adding another dense layer to the network while experimenting with the number of nodes in each layer. Next, we propose LSTM4 with two consecutive LSTM layers as before, but now with the four subsequent dense layers having 100, 60, 20, and 5 nodes, respectively. LSTM4 performed well by the metric of lowering the RMSE for the validation set while only slightly increasing it for the training set compared to LSTM3. Figure 11 shows the MSE and RMSE for this model, which are drastically different compared to the results obtained for LSTM1.

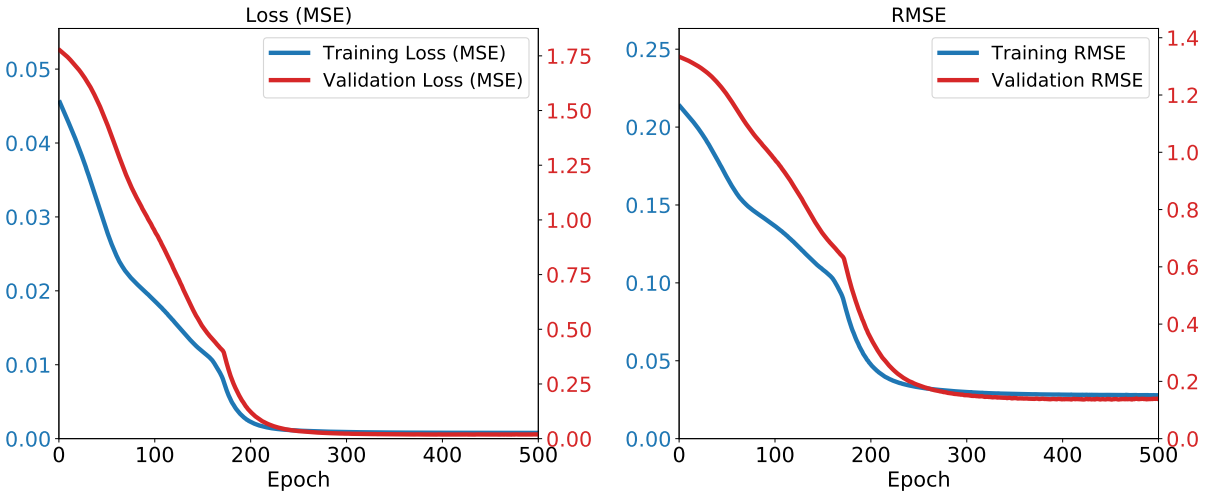

Figure 11: LSTM4 MSE and RMSE for Training and Validation Datasets.

We also experimented with adding some dropout regularisation to LSTM4 between each of the dense layers, but this did not add any extra improvement in lowering the loss or RMSE or reducing potential overfitting. In cases where the dropout rate was somewhat large, this was detrimental to the model's metrics. A plot of the architecture of this model (LSTM5), which includes the regularisation layers, can be found in Appendix D.2. Plots of the MSE and RMSE for LSTM5 with dropout regularisation rate of 0.3 are below in Figure 12.

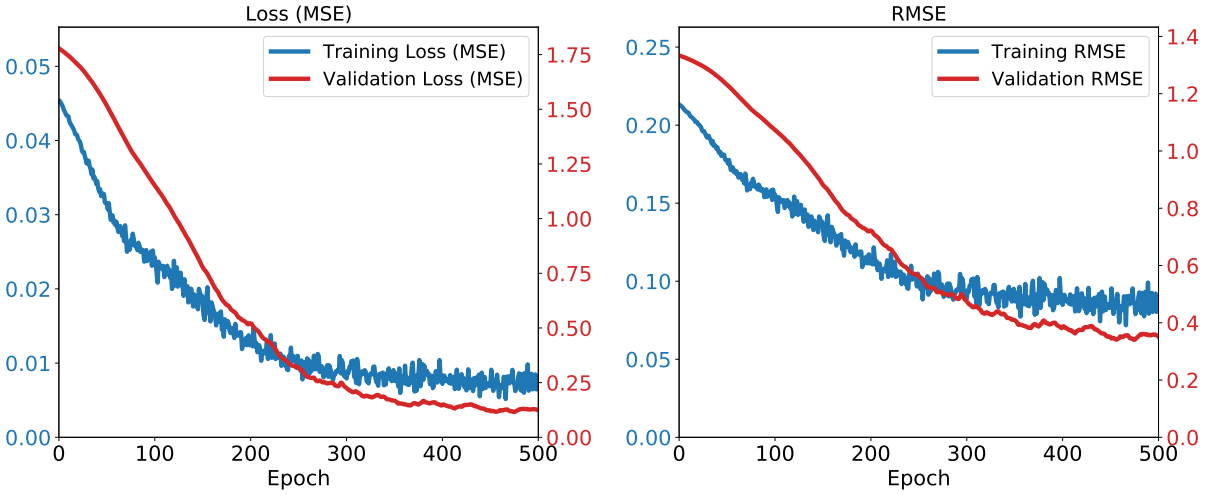

Figure 12: LSTM5 MSE and RMSE for Training and Validation Datasets.

The dropout regularisation caused some oscillatory effects to occur to our performance metrics which means that the training and validation loss failed to converge. Hence, we will not proceed

with adding dropout regularisation to the network, and will proceed with LSTM4 as our final model to carry out the main goal of this work of investigating the robustness of a deep learning sequence model.

6.3 Final Model Performance

The final model moving forward for the main goal of this work is LSTM4, which performed well in terms of having a small loss and RMSE on the training and validation dataset. To review, the RNN proposed for which we will investigate robustness is an RNN with two LSTM layers, both with 200 neurons, and four dense layers having 100, 60, and 20 neurons, with the final output layer having 5 neurons as before. The activation function for each layer was the Leaky ReLU with $\alpha = 0.2$. Comparing the plots of the MSE and RMSE of this final model in Figure 11 to that for LSTM1 in Figure 8, we see that LSTM4 has a significantly lower loss and RMSE. Of course, we could try to improve this model further and continually tune the model hyperparameters and the number of layers within the network while also testing different learning rates and optimisers to improve accuracy, but this is not the core focus for this work. A graphical representation of the RNN architecture for LSTM4 is below in Figure 13.

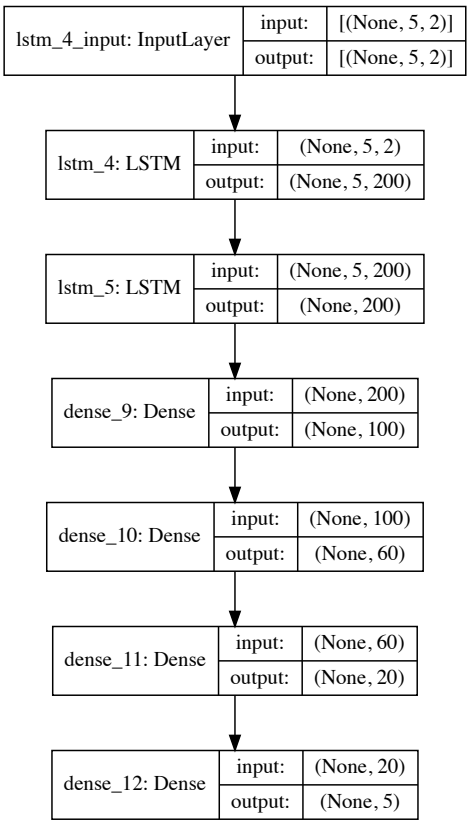

Figure 13: Graphical Representation of LSTM4 Architecture.

Although our unseen out-of-sample dataset has 40 days' worth of prices, it would not be reasonable to make predictions on all 40 days' worth of prices as our model was not trained on the 80 days in the validation dataset, which was a period during a large bull run within the cryptocurrency space in general. Hence our neural network may miss out on some hidden trends when this period is not included in the training data, so we reduced our unseen dataset to the 15 earliest prices and volumes to evaluate out-of-sample performance. Also, in a real-world setting, the model would be adjusted and retrained continually when new data is captured, say, at the end of the day or five-day period when an overall architecture is proven to make market-beating predictions. However, we do not do this for this study, as we will see later, we want to observe how the robustness tests affect the training, validation and out-of-sample loss and RMSE.

When predicting prices using the unseen dataset, these predictions have an RMSE of 0.1179. However, these values are scaled, so we reverse the scaling process to obtain the true and predicted values. Plots of the model's predictions and true prices are below in Figure 14. The prediction performance of the model is decent considering we only used closing prices and volume traded as input and the fact that the model was not trained using the validation dataset, which contained more recent prices, with the unseen dataset containing the most recent prices from the entire dataset. A table of the RMSE for each of the models outlined previously can be found in Appendix E.1.

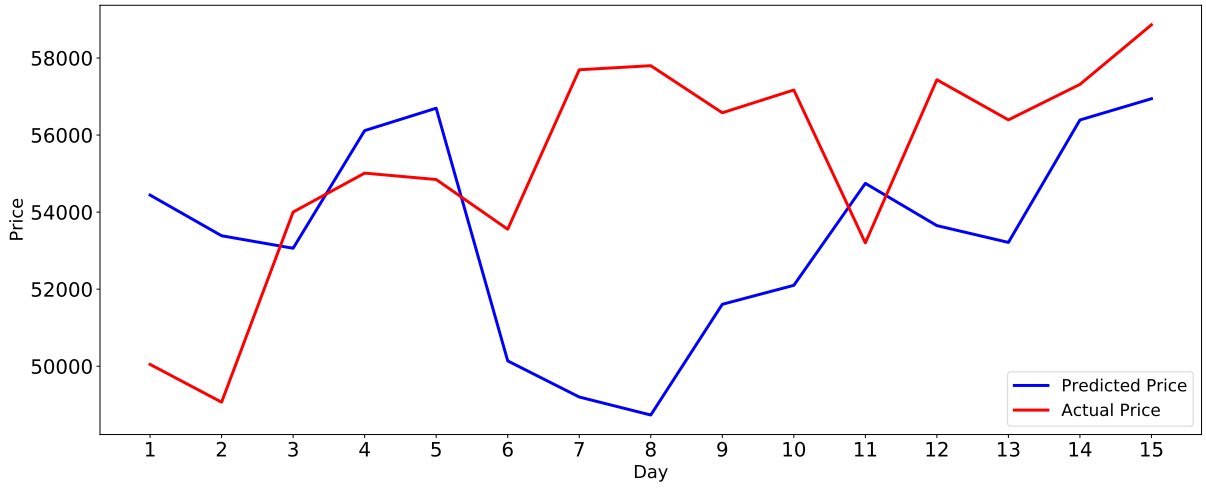

Figure 14: Predicted and Actual BTC Prices(USD) for Final Model.

7 Testing Robustness

7.1 Data Noise Injection

The first method we will use to test the robustness of the model created will involve creating Gaussian noise and adding it to the data used to train the model. The Gaussian noise will have mean zero with a variable standard deviation. To foster the imperceptible nature of the robustness tests, the standard deviation will be relatively small initially to "disguise" the added noise. We will also allow the number of noise perturbations created to be variable. This is to investigate if there is any effect on model training performance if there are a large number of perturbations from the same distribution versus a small number of perturbations in the training data. Each of the perturbations will be randomly distributed among the blocks and randomly added to one of the five closing prices within the block. We will restrict the perturbations only to affect one of the closing prices in each five-day block. To keep the data consistent, we need to apply this perturbation to the input and output vectors. See Figure 15 for one example of the noise injection for one five-day block, where x_i is the closing price at time i, and ϵ is the Gaussian noise, with the perturbed data surrounded in red.

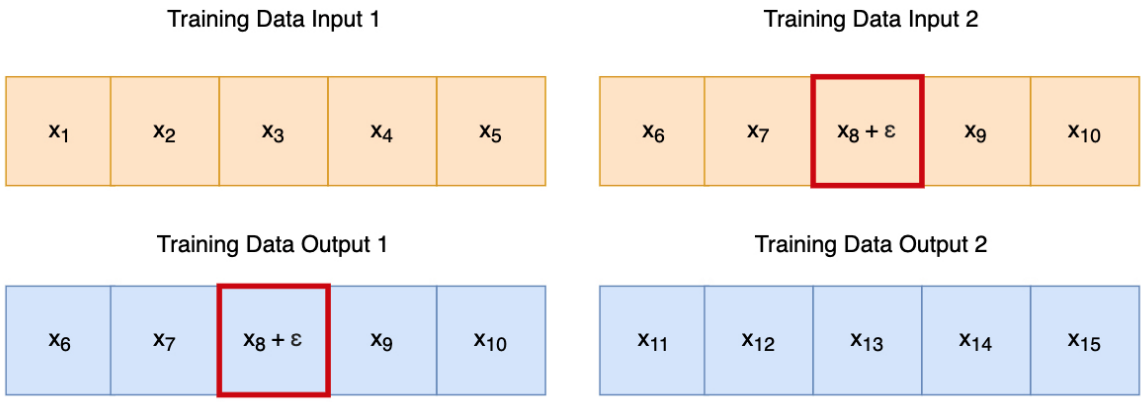

Figure 15: Noise Injection Example.

This method was chosen firstly for its simplicity as it is not a difficult task to add noise to the training data, especially since our training data is scaled between 0 and 1 due to the nature of the MinMax scaler. Secondly, this type of noise injection fits in perfectly with our previous discussion of adversarial attacks where small perturbations are added to the data in the hopes of hampering model performance. However, we must also make this noise injection realistic as the perturbations are designed to be small so that they are somewhat undetectable either by humans or other means. This type of data augmentation is recognised in the literature as a standard regularisation practice and to test model robustness. In a time series classification task,

Fawaz et al. (2019) perturbed a time series with imperceptible noise resulting in misclassifying the time series. We will use the modified noisy training data to fit and train a model using the same deep learning infrastructure as the final model and investigate how varying strengths and frequencies of noise impact the model. We will then evaluate the model's performance on our unseen out-of-sample dataset and compare it to when the LSTM4 was trained on the original data.

7.2 Model Noise Injection

We also have the ability to inject noise into the neural network itself. This can be achieved in numerous ways. Two methods we will focus on will be the addition of Gaussian noise layers in between layers of the neural network and replacing dense layers with noisy dense layers in the deep learning model. This is a slightly less realistic method of adding an adversarial perturbation as a quantitative researcher would physically need to write the code to add these layers to their network. However, this is still a vital experiment to test the robustness of a deep learning model as it contributes to the "black box" aspect some models suffer from. Since we can add artificial noise to the internal infrastructure of the model, it may be indicative of how sensitive the model may be to other unforeseen inputs or errors during the model building process.

7.2.1 Gaussian Noise Layers

Gaussian noise layers add noise to that layer's inputs and return the outputs in the same shape as the inputs. Essentially, the Gaussian noise layer acts as a buffer between two previously consecutive layers and adds noise to the outputs from the first layer that were to be used as inputs to the second layer. This is somewhat similar to injecting noise into the data before it is presented the model for the training phase but, the noise occurs within the model and can be repeated when multiple Gaussian layers are present. Once again, we can control the strength of the noise by varying the standard deviation of the noise.

7.2.2 Noisy Dense Layers

Noisy dense layers are similar to that of a dense layer except that the weights and biases are augmented by factorised Gaussian noise. This Gaussian noise is controlled through gradient descent by a second weights layer. The noisy dense layer is implemented as follows:

$$\text{NoisyDense}(x) = \text{activation}(dot(x, \mu + (\sigma \cdot \epsilon)) + \text{bias}) \tag{12}$$

where μ is the standard weights layer, ϵ is the factorised Gaussian noise, and σ is the standard deviation applied to the Gaussian noise layer. We now have two distinct methods that augment

the neural network where one influences the inputs between layers directly while the other affects the weights and biases at each layer. We will again investigate how these layers influence model behaviour during the training phase of the model and the predictions on the unseen dataset for various noise strengths and differing amounts of these layers.

7.3 Comparison With Linear Models

To put the performance and robustness of the RNN created in this work into context, we will compare its performance and robustness against that of traditional linear methods. We will create an ARIMA model designed with predicting closing prices like our neural network. We will first investigate how well this ARIMA model performs at predicting prices for our unseen data. We will also explore how the ARIMA model's parameters and the statistical significance of these parameters change when we add noise to the data used to fit the model, similar to the noise injection technique for LSTM4. Although we know that ARIMA models are now somewhat obsolete for modelling asset prices, having been replaced in favour of more complicated machine learning methods to find some hidden alpha, this comparison is still important to put into context how robust or potentially how vulnerable the LSTM networks created in this work are. Details of the ARIMA model building process can be found in Appendix F.

8 Results

8.1 Data Noise Injection Results

Initially, we tested robustness using the noise injection method by adding Gaussian noise with a standard deviation of 0.01 to 25 days in the training data. This is comparatively a small amount of days given the training set has over 1000 days' worth of prices, but it acts as a good baseline. From Figure 16, we note how the two lines are almost indistinguishable with the noisy data in blue and the original data in red. This augmentation of the training data would be likely to go unnoticed by humans as a result.

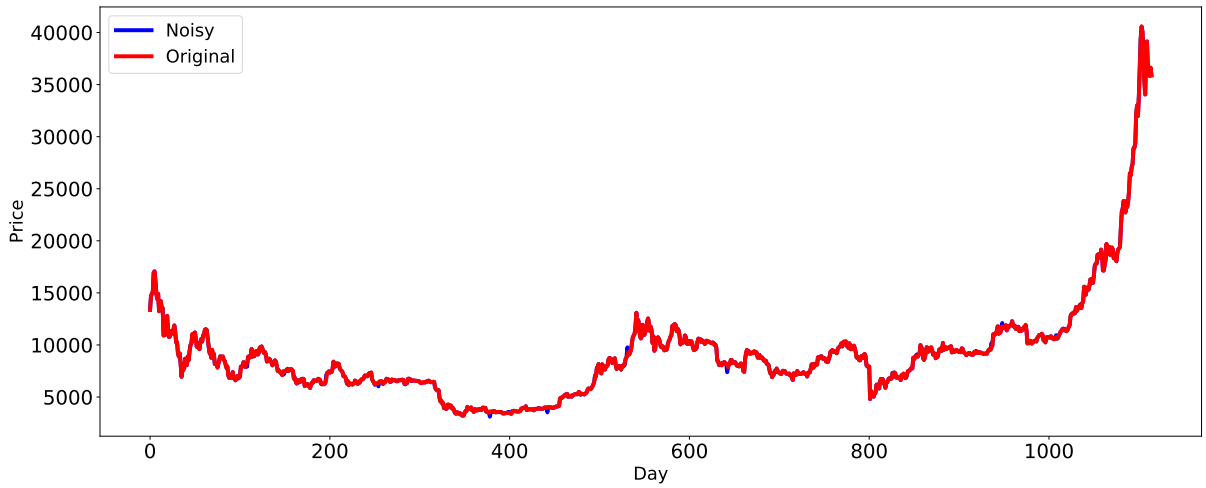

Figure 16: Noisy and Original Training Data with 25 Noisy Data Points ($\sigma = 0.01$).

Plots of the MSE and RMSE of LSTM4 trained on this noisy data sample are seen in Figure 17. The RMSE for the training data decreased very slightly. The RMSE of the predictions for the unseen data remained unchanged at 0.1179.

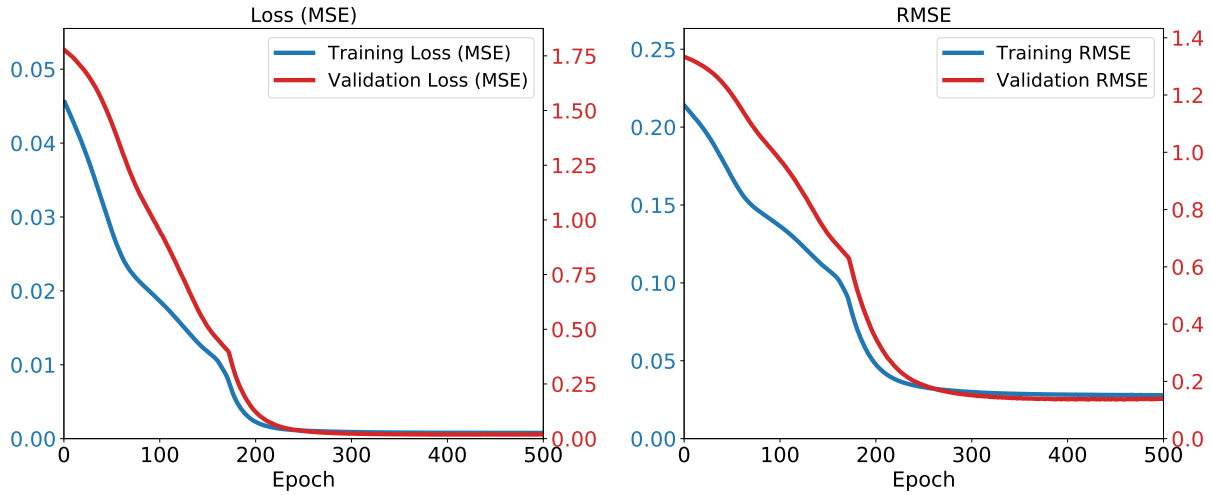

Figure 17: MSE and RMSE for Training Data with 25 Noisy Days and Validation Data.

Next, the noise injection experiment was repeated but with 100 Gaussian noise samples added to the training data. Once again the perturbation was fairly undetectable when looking at a plot of the data in Figure 18.

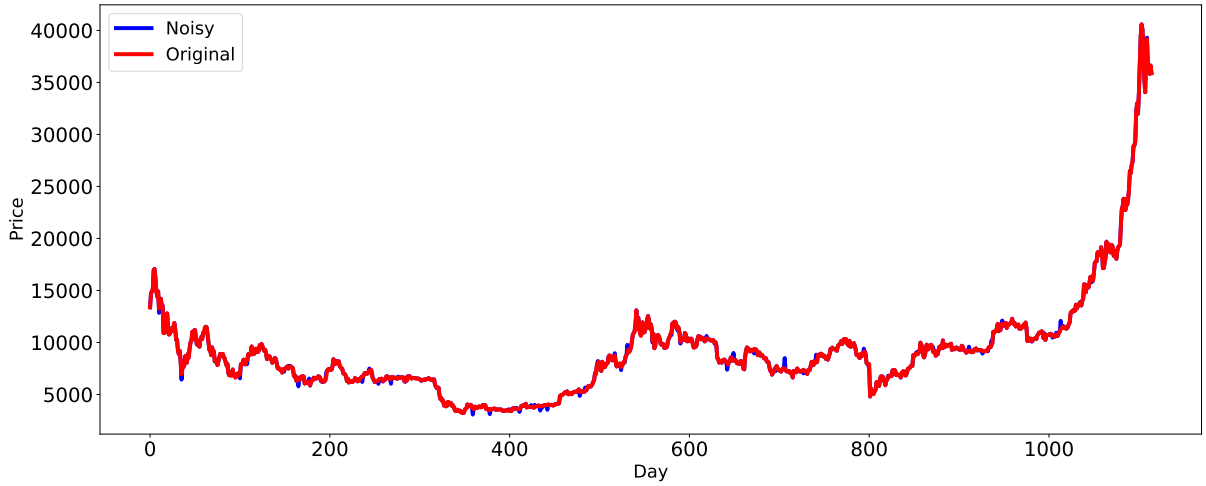

Figure 18: Noisy and Original Training Data with 100 Noisy Data Points ($\sigma = 0.01$).

The training and validation loss and RMSE were again largely unaffected by the augmented data as seen in Figure 19. The RMSE for the out-of-sample data was mostly unchanged at 0.1182.

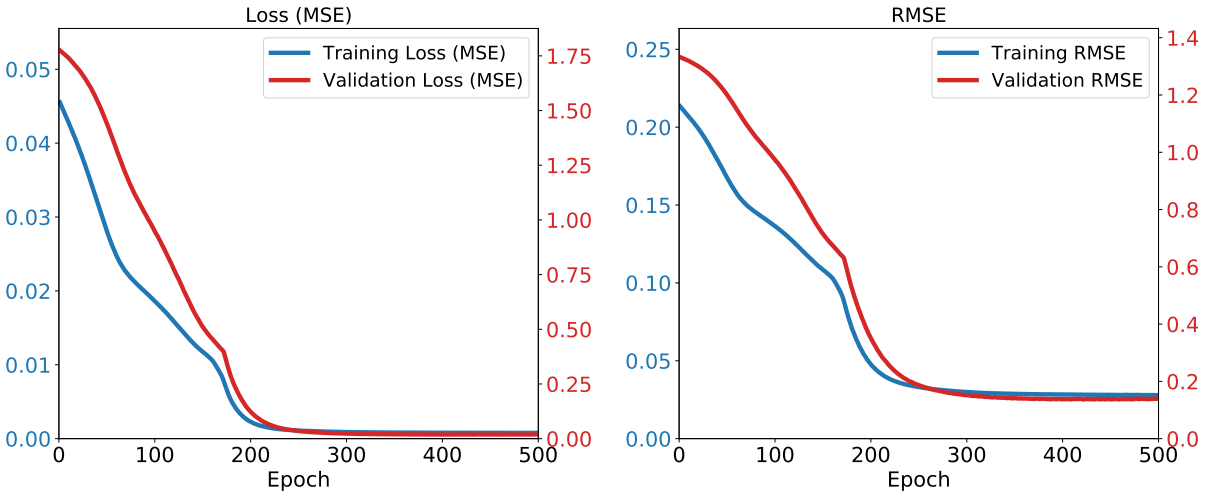

Figure 19: MSE and RMSE for Training Data with 100 Noisy Days and Validation Data.

With no alarming deterioration of the model's prediction power from the two previous noise injection experiments, an extreme example was tested by increasing the standard deviation of the noise to 0.1 and by having 250 noise samples added randomly throughout the training data. The effect of this augmentation can be seen in Figure 20. Obviously, a researcher would immediately recognise that some error has occurred during the data processing phase when looking at this plot. Regardless, we continued as before and trained the model on this augmented data.

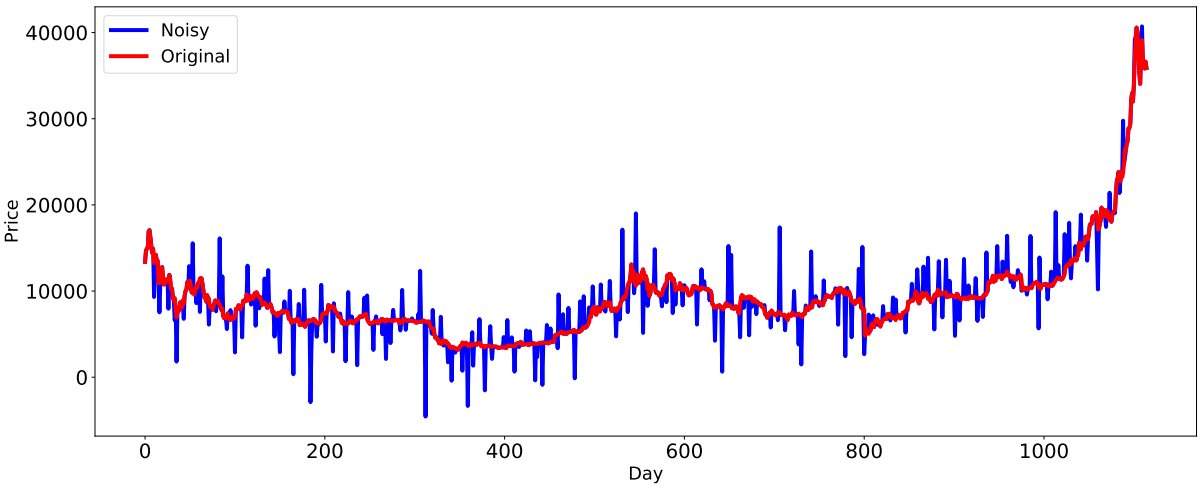

Figure 20: Noisy and Original Training Data with 250 Noisy Data Points ($\sigma = 0.1$).

Surprisingly, despite the extreme noise added to the training data, the noisy data points seemingly had little to no effect on the MSE and RMSE, as seen in Figure 21. Further, the

out-of-sample RMSE was 0.1169, which is similar to that of the previous experiments with noisy and original training data. However, the training RMSE has increased slightly from the previous two experiments.

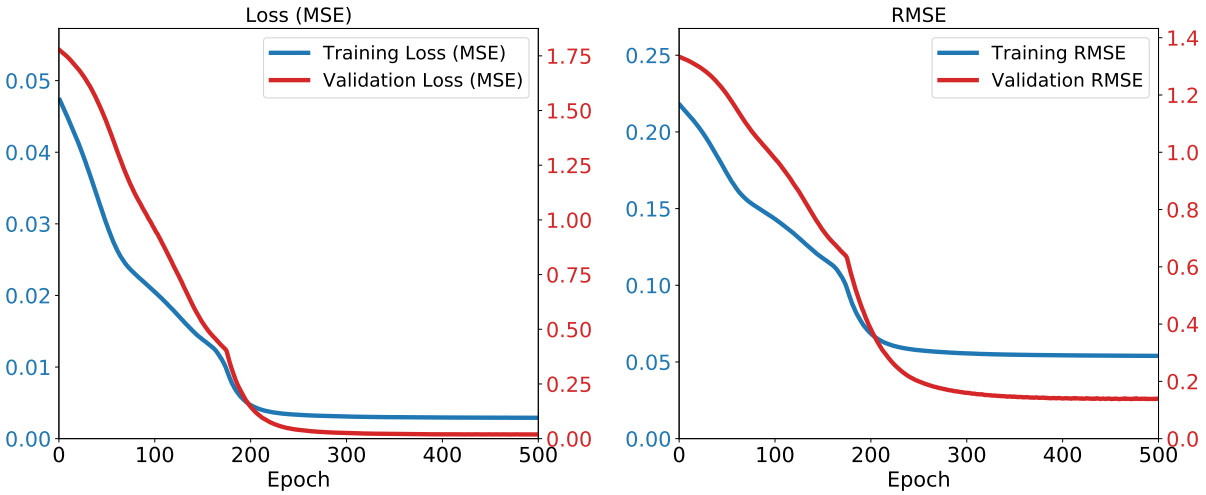

Figure 21: MSE and RMSE for Training Data with 250 Noisy Days and Validation Data.

8.2 Model Noise Injection Results

8.2.1 Gaussian Noise Layers

To test our methods of adding noise to the network itself, we first present the results from adding Gaussian noise layers to the network. First, three Gaussian noise layers were added to the proposed final model with a standard deviation of 0.02. One after the second LSTM layer, and then one after each of the next two dense layers. Plots of the MSE and RMSE during the model training phase are below in Figure 22. The RMSE of prediction on the unseen dataset was essentially unchanged at 0.1166.

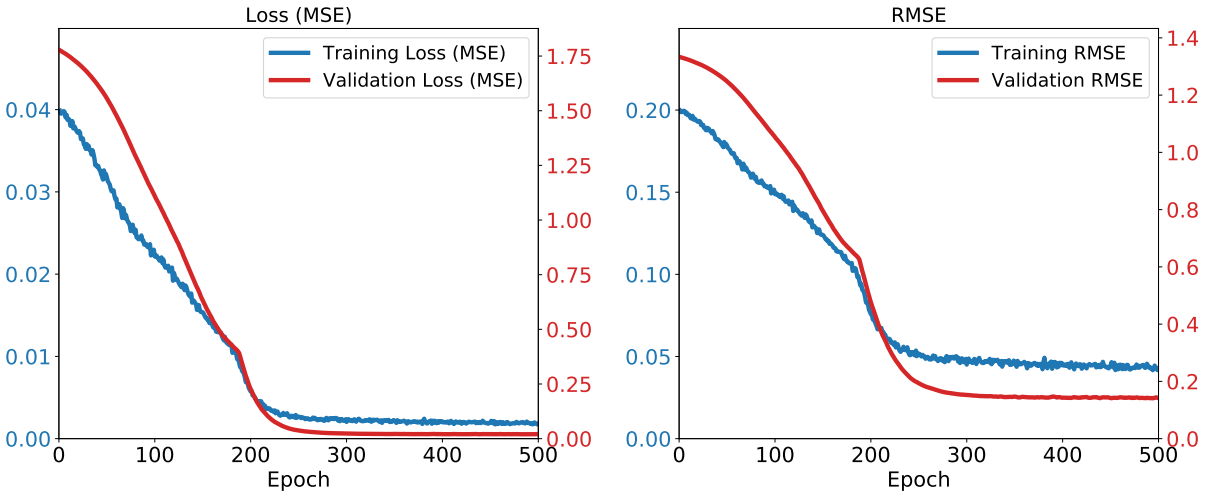

Figure 22: MSE and RMSE for LSTM4 with Three GaussianNoise Layers ($\sigma = 0.02$).

With no drastic effect on the MSE and RMSE from the previous experiment, another Gaussian noise layer was added with the first three in the same position as before in the model, with an additional Gaussian noise layer before the final dense layer. The standard deviation was also increased to 0.1 in each layer to investigate the effect of increasing the perturbation would have, if any. From Figure 23, it is clear that the combination of adding an extra noisy layer and increasing the standard deviation of the Gaussian noise proved to disrupt the accuracy of the network for the training and validation data. This jitter effect is an indication that the loss cannot converge. Of course, this then proved detrimental to our predictions for the unseen dataset, with these predictions having an RMSE of 0.4019, compared to an RMSE of 0.1179 for LSTM4.

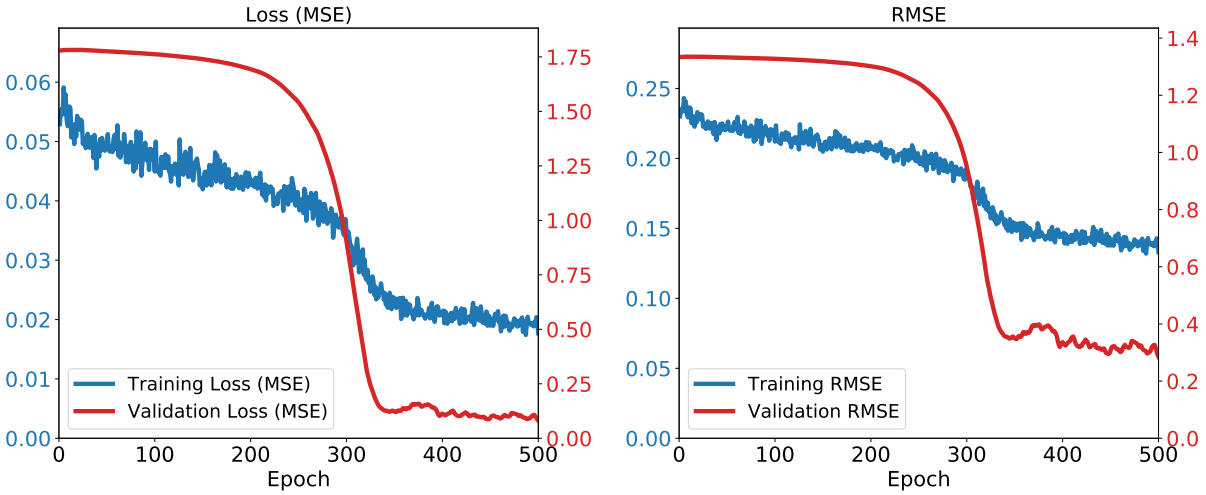

Figure 23: MSE and RMSE for LSTM4 with Four GaussianNoise Layers ($\sigma = 0.1$).

A plot of the predictions made by this perturbed network is below in Figure 24. We can see how these predictions are visibly worse than those of LSTM4.

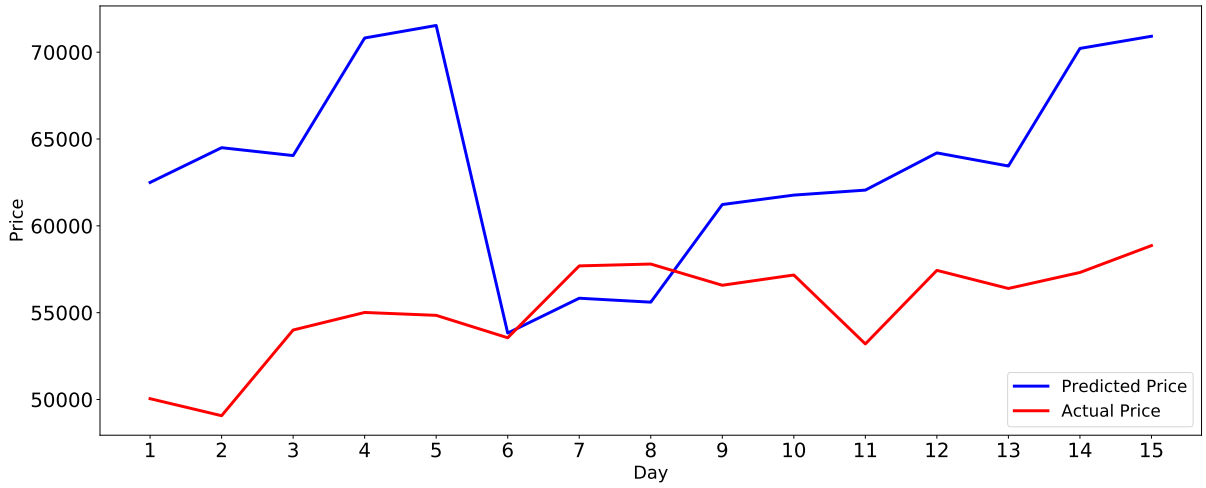

Figure 24: Predicted and Actual Prices For LSTM4 with Four Gaussian Noise Layers ($\sigma = 0.1$).

8.2.2 Noisy Dense Layers

Next, we present the results of replacing some of the dense layers of the RNN with noisy dense layers. First, the dense layer which contained 100 neurons was replaced with a noisy dense layer with a standard deviation of 0.1. We can see from the plots of the MSE and RMSE in Figure 25 that replacing this layer with the noisy layer had an immediate effect in adding some oscillatory effect to the metrics for validation data. The RMSE of the predictions for the unseen data was 0.1567 using this model, which was slightly worse than our final model's accuracy.

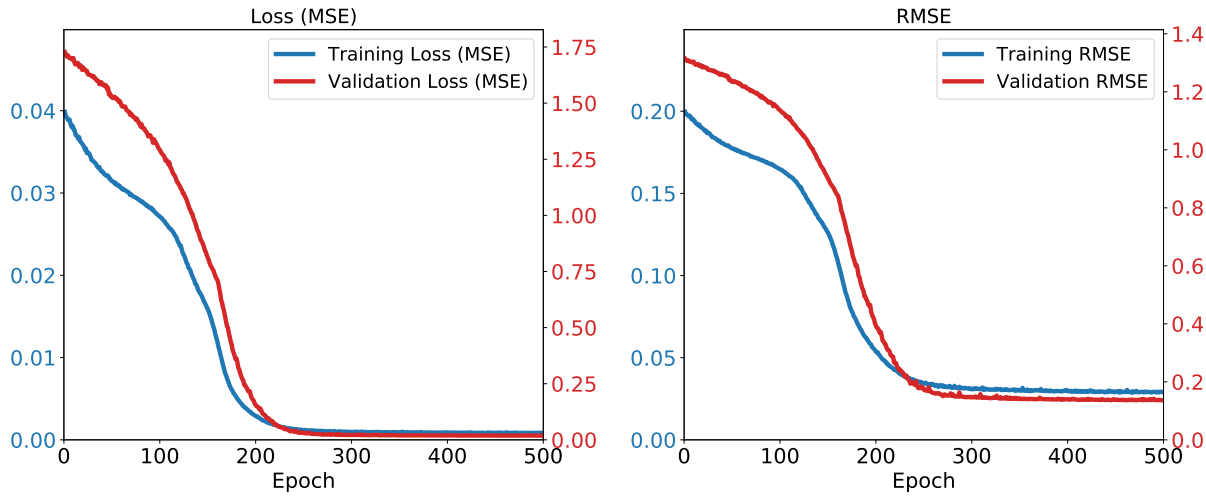

Figure 25: MSE and RMSE for Training Data with One Noisy Dense Layer ($\sigma = 0.1$).

Having seen how detrimental the effect of one noisy dense layer is to the model metrics during training, we investigated just how much of an undesirable effect having all dense layers except the final output layer being replaced would have. The dense layers with 100, 60, and 20 neurons were replaced with noisy dense layers with the same number of respective neurons in each layer with standard deviations of 0.1. Unsurprisingly, the effect from the previous experiment was magnified, as seen in Figure 26. This also impacted the RMSE of the predictions on the unseen dataset, which was 0.1829. We note how this is worse than the RMSE for LSTM4 but better than the results from the final experiment with Gaussian noise layers.

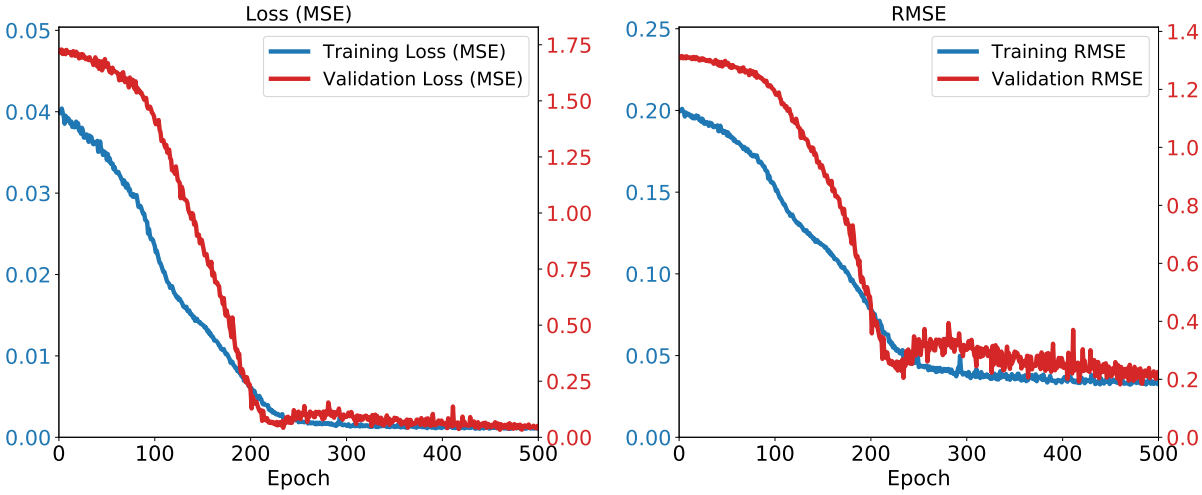

Figure 26: MSE and RMSE for Training Data with Three Noisy Dense Layers ($\sigma = 0.1$).

8.3 Comparison With ARIMA Model

Using the ARIMA(2,1,2) model proposed in Appendix F, we first present the out-of-sample predictions for the same 15 days as we have done for our deep learning methods. If we project the next 15 days using the ARIMA(2,1,2) model, we obtain the plot below in Figure 27. Quite clearly, this model is inferior at making predictions on the out sample data compared to our LSTM4 network.

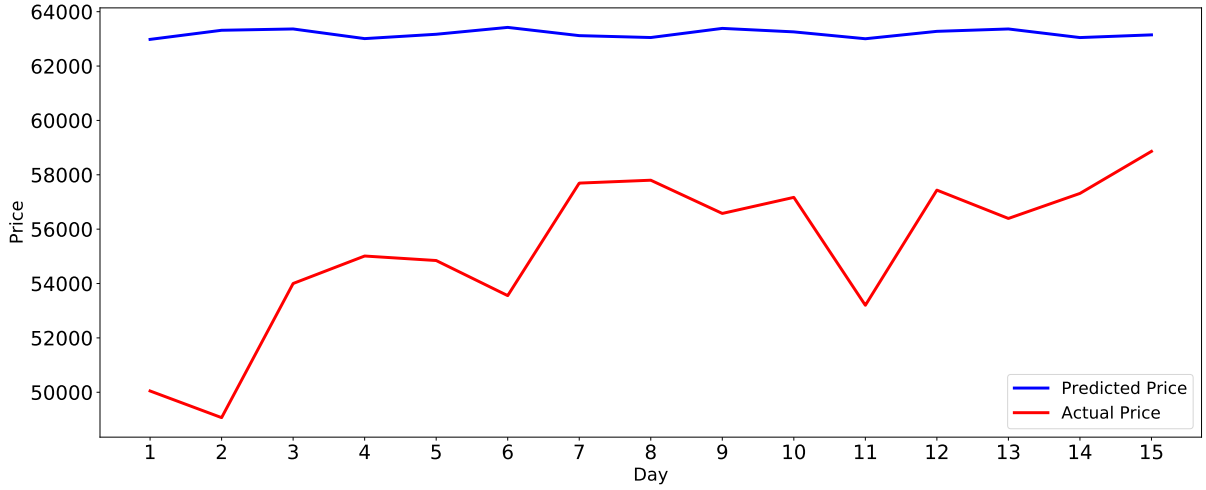

Figure 27: ARIMA(2,1,2) 15-Day Projections.

We also ran the same projection but allowed the ARIMA(2,1,2) model to be updated after making a 5-day prediction by fitting the ARIMA(2,1,2) model again with the actual values from the unseen dataset being added to the training data for this model. The predictions for this model are below in Figure 28.

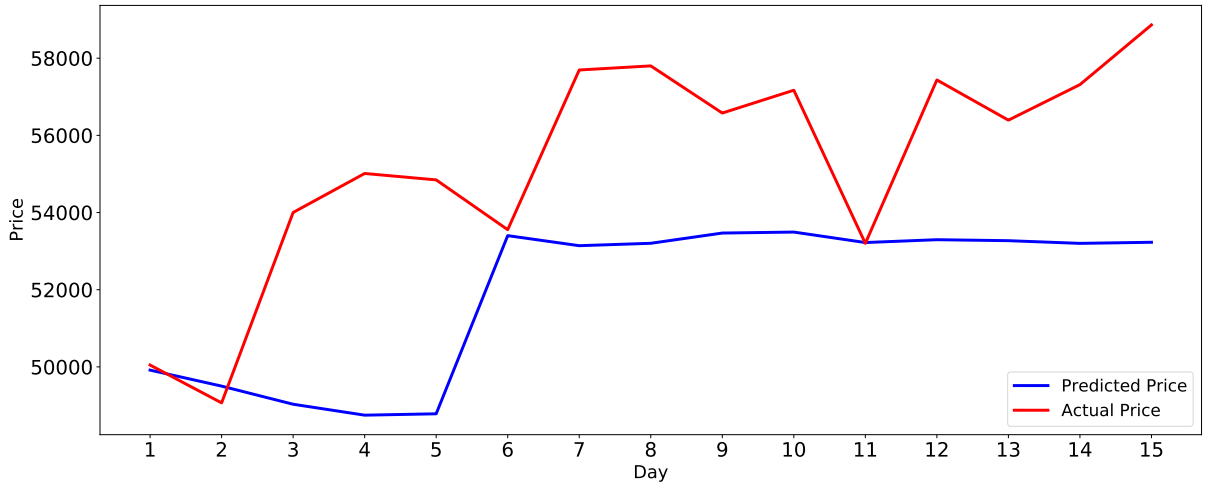

Figure 28: ARIMA(2,1,2) 15-Day Projections with Model Updating.

This slightly modified version of fitting the model, although superior to the previous model, still does not offer much prediction power within the five-day period and essentially determines that the BTC price will remain essentially the same until the model receives the actual price at the start of the next five-day window and updates accordingly when we refit the model.

Further, Gaussian noise was added to the training data used to fit the ARIMA model, similar to the noise added for training the RNN previously. The noise added had a standard deviation of 0.1 and was added to 15% of the 1200 days' worth of closing prices. A plot of the noisy and original prices is seen below in Figure 29. Once again, this noise is fairly indistinguishable from the original data.

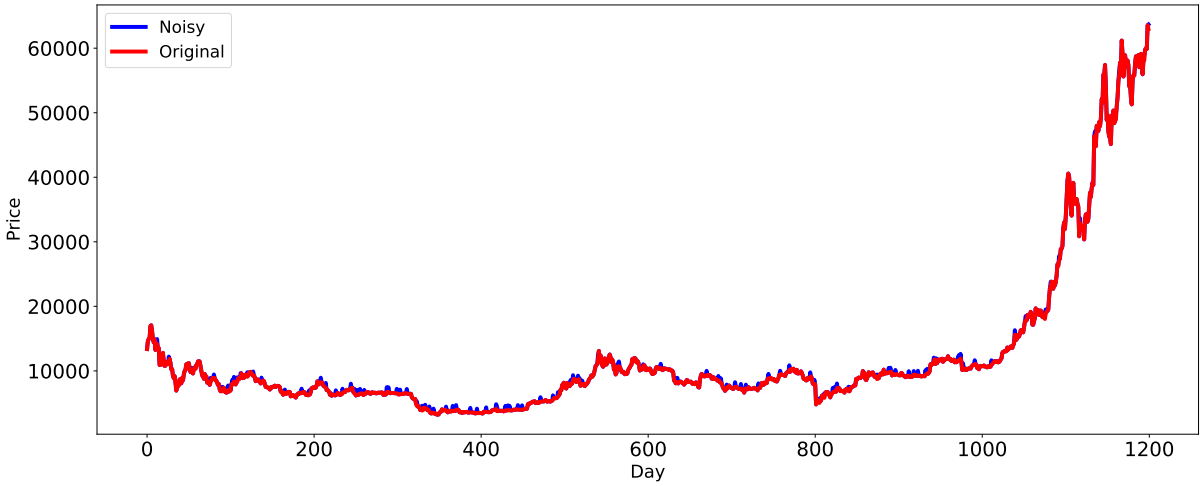

Figure 29: ARIMA(2,1,2) Noisy Training Data.

Fitting the ARIMA(2,1,2) model again with this noisy data similar to that in Appendix F, the model contains the following parameters for the model in Table 1.

Table 1: ARIMA Model Parameters Trained On Noisy Data

Parameter	Coefficient	Std Error	P-Value
AR 1	-1.0512	0.088	0.000
AR 2	-0.5489	0.1	0.000
MA 1	0.9768	0.092	0.000
MA 2	0.4543	0.108	0.000

We note how all of the parameters of the model trained on the noisy data are vastly different to those of the model trained on the original data in Appendix F.1. We also note how the standard errors for the parameter estimates are approximately ten times larger than the parameters in the original ARIMA(2,1,2) model.

8.4 Results Discussion

From the RMSE and the plot of the predictions of our final model, we can see how the model makes decent predictions for the three five-day window periods in the out-of-sample dataset. To criticise the predictions slightly, there are some instances in the 15-day predictions where the predicted prices are roughly 7000 less than the true price. It would be extremely difficult to predict daily BTC prices with market-beating accuracy simply from historical closing prices and trading volumes. We admit that there are slight signs of overfitting when looking at the MSE and RMSE for the training data, with the RMSE being approximately 0.04. Adding dropout regularisation to try and alleviate this resulted in the RMSE being unable to converge. Future work could address this by adding more input variables, using other regularisation methods, and potentially varying the learning rate.

When we trained the model with the noisy data, we observed that this did not drastically affect the RMSE for the out-of-sample predictions. Although we do note that when trained on the noisiest data with $\sigma = 0.1$, the training loss and RMSE increased slightly, which is not too surprising. These results could be encouraging as practitioners may discover some noise in their data due to randomness or lack of measurement precision. They could still be confident that this noise may not hamper their prediction power too much. This agrees with the findings of Cheng et al. (2020), where sequence-to-sequence models are deemed to be robust to adversarial attacks in the data.

From our first example of adding three Gaussian noise layers to the network with a standard deviation of 0.02, the RMSE for the out-of-sample predictions was slightly larger than that for the original model. This is not a huge increase, but we notice how the plot of the training RMSE in Figure 22 has a slight jitter pattern. This is likely due to the fact that the Gaussian noise is randomly generated for each epoch of the training period. This effect was magnified when we added another Gaussian noise layer to the network and increased the standard deviation to 0.1. We note how both the training and validation MSE and RMSE increase while also having a jitter effect. This indicates that the Gaussian noise layers hamper the model's ability to converge. This carried through to the out-of-sample predictions where the RMSE of the predictions is 0.4019, which was far greater than that of LSTM4.

The addition of one noisy dense layer initially had a small effect on the training and validation loss, with the out-of-sample RMSE increasing to 0.1576 from the LSTM4 baseline of 0.1179. When we replaced two more dense layers, the effects on the MSE and RMSE were exacerbated. Interestingly, the validation RMSE suffered the most, with the noisy dense layers preventing the RMSE from converging. But the effect on the out-of-sample accuracy, although increasing the RMSE to 0.1829, was not as large as the Gaussian noise layers. Both the standard deviation

of the noise and the number of noisy layers were important for the disruption to the model. These results may indicate that perturbations carried between layers in a network are more detrimental to an RNN than perturbations within a layer.

We also found that the ARIMA model was extremely susceptible to perturbations in the data while also performing worse than the LSTM models when making out-of-sample predictions. This agrees with Petneházi (2019), who asserts that RNNs are far more powerful than traditional linear methods.

9 Conclusion

We successfully created an RNN with LSTM architecture tasked with predicting Bitcoin prices with a reasonable degree of accuracy. We found that increasing the depth and adding LSTM layers to the network for our specific task increased prediction accuracy for the validation dataset and, subsequently, the out-of-sample dataset. Testing the robustness of this model, we found that this deep learning sequence model was especially robust to random perturbations in the data even if the size and number of perturbations in the training data were unrealistically large. However, the robustness of the model suffered when Gaussian noise layers and noisy dense layers were added to the model, especially when there were more than one of these disruptive layers in the network. The Gaussian noise layers caused the largest deterioration in model accuracy. However, the standard deviation of the noise was increased to 0.1, which was arguably large when disrupting model weights. These disruptions within the model are illustrative of the care that needs to be taken when making networks that are even deeper than those in this work, such as that in AlphaFold that has numerous networks linked together. The RNN outperformed a linear ARIMA model for out-of-sample prediction and was especially robust to perturbations within the data, which significantly changed the ARIMA model's AR and MA parameters. Overall the LSTM was fairly robust, where invasive and often unrealistically large perturbations had to be added to the network's architecture to sabotage it.

Of course, this work is not without its flaws and leaves the door open for further discussion. One could argue that a model that is trained on only closing prices and volume traded would not provide any hidden excess alpha in the long run. However, building a state-of-the-art model that could compete with quantitative hedge-funds and market makers with far more staff, computing power, and data signals was not the focus here. The models and experiments were kept relatively simple to carry out the core focus of this work. Although, LSTM4, with only six layers, had over 500,000 parameters in the model, which is a large number of parameters in a relatively simple RNN. These models are also potentially limited by the EMH, with many hedge funds now scraping Twitter and forum data as an avenue to create alpha using sentiment analysis. One modification worth looking into the future could be to look at the data for different time scales and prediction windows and investigate if similar models are as robust for prediction tasks at minute, second, or even smaller time scales. However, at extremely small time scales, market dynamics can change, which would need to be factored into account before testing the robustness of these models. Investigating the robustness of financial deep learning models will remain critical with their usage in the future.

References

Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A. and Criminisi, A. (2017), 'Measuring neural net robustness with constraints'.

Bishop, C. M. (2006), 'Pattern recognition', *Machine learning* **128**(9).

Cheah, E.-T., Mishra, T., Parhi, M. and Zhang, Z. (2018), 'Long memory interdependency and inefficiency in bitcoin markets', *Economics Letters* **167**, 18–25.
URL: *https://www.sciencedirect.com/science/article/pii/S0165176518300582*

Chen, L., Pelger, M. and Zhu, J. (2021), 'Deep learning in asset pricing'.

Cheng, M., Yi, J., Chen, P.-Y., Zhang, H. and Hsieh, C.-J. (2020), 'Seq2sick: Evaluating the robustness of sequence-to-sequence models with adversarial examples'.

Fan, F., Xiong, J., Li, M. and Wang, G. (2021), 'On interpretability of artificial neural networks: A survey'.

Fawaz, H. I., Forestier, G., Weber, J., Idoumghar, L. and Muller, P.-A. (2019), Adversarial attacks on deep neural networks for time series classification, *in* '2019 International Joint Conference on Neural Networks (IJCNN)', IEEE, pp. 1–8.

Franklin, J. (2005), 'The elements of statistical learning: data mining, inference and prediction', *The Mathematical Intelligencer* **27**(2), 83–85.

Germain, M., Pham, H. and Warin, X. (2021), 'Neural networks-based algorithms for stochastic control and pdes in finance'.

Gers, F., Schmidhuber, J. and Cummins, F. (2000), 'Learning to forget: Continual prediction with lstm', *Neural computation* **12**, 2451–71.

Goldblum, M., Schwarzschild, A., Patel, A. B. and Goldstein, T. (2020), 'Adversarial attacks on machine learning systems for high-frequency trading'.

Graves, A. (2012), *Supervised Sequence Labelling with Recurrent Neural Networks*, Studies in computational intelligence, Springer, Berlin.
URL: *https://cds.cern.ch/record/1503877*

Guida, T. (2019), *Big Data and Machine Learning in Quantitative Investment*, John Wiley & Sons.

Göpfert, J. P., Artelt, A., Wersing, H. and Hammer, B. (2020), 'Adversarial attacks hidden in plain sight', *Advances in Intelligent Data Analysis XVIII* p. 235–247.

Heaton, J. B., Polson, N. G. and Witte, J. H. (2018), 'Deep learning in finance'.

Hochreiter, S. and Schmidhuber, J. (1997), 'Long Short-Term Memory', *Neural Computation* **9**(8), 1735–1780.
URL: *https://doi.org/10.1162/neco.1997.9.8.1735*

Hu, Y., Valera, H. G. A. and Oxley, L. (2019), 'Market efficiency of the top market-cap cryptocurrencies: Further evidence from a panel framework', *Finance Research Letters* **31**, 138–145.
URL: *https://www.sciencedirect.com/science/article/pii/S154461231830610X*

Janiesch, C., Zschech, P. and Heinrich, K. (2021), 'Machine learning and deep learning', *Electronic Markets* .
URL: *http://dx.doi.org/10.1007/s12525-021-00475-2*

Jiang, Y., Nie, H. and Ruan, W. (2018), 'Time-varying long-term memory in bitcoin market', *Finance Research Letters* **25**, 280–284.
URL: *https://www.sciencedirect.com/science/article/pii/S1544612317306682*

Kim, M. and Gilley, J. (2008), 'Artificial neural network estimation of soil erosion and nutrient concentrations in runoff from land application areas', *Computers and Electronics in Agriculture* **64**, 268–275.

Ko, C.-Y., Lyu, Z., Weng, T.-W., Daniel, L., Wong, N. and Lin, D. (2019), 'Popqorn: Quantifying robustness of recurrent neural networks'.

Kocic, J., Jovicic, N. and Drndarevic, V. (2019), 'An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms', *Sensors* .

Maas, A. L., Hannun, A. Y. and Ng, A. Y. (2013), Rectifier nonlinearities improve neural network acoustic models, *in* 'in ICML Workshop on Deep Learning for Audio, Speech and Language Processing'.

Malhotra, P., Vig, L., Shroff, G. and Agarwal, P. (2015), Long short term memory networks for anomaly detection in time series.

McCulloch, W. S. and Pitts, W. (1943), 'A logical calculus of the ideas immanent in nervous activity', *The bulletin of mathematical biophysics* **5**(4), 115–133.
URL: *https://doi.org/10.1007/BF02478259*

Mehtab, S., Sen, J. and Dutta, A. (2020), 'Stock price prediction using machine learning and lstm-based deep learning models'.

Nadarajah, S. and Chu, J. (2017), 'On the inefficiency of bitcoin', *Economics Letters* **150**, 6–9.
URL: *https://www.sciencedirect.com/science/article/pii/S0165176516304426*

Nair, V. and Hinton, G. E. (2010), Rectified linear units improve restricted boltzmann machines, *in* 'Proceedings of the 27th International Conference on International Conference on Machine Learning', ICML'10, Omnipress, Madison, WI, USA, p. 807–814.

Nwankpa, C., Ijomah, W., Gachagan, A. and Marshall, S. (2018), 'Activation functions: Comparison of trends in practice and research for deep learning'.

Petneházi, G. (2019), 'Recurrent neural networks for time series forecasting'.

Rosenblatt, F. (1958), 'The perceptron: A probabilistic model for information storage and organization in the brain', *Psychological Review* pp. 65–386.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986), 'Learning representations by back-propagating errors', *Nature* **323**(6088), 533–536.
URL: *https://doi.org/10.1038/323533a0*

Senior, A. W., Evans, R., Jumper, J., Kirkpatrick, J., Sifre, L., Green, T., Qin, C., Žídek, A., Nelson, A. W. R., Bridgland, A., Penedones, H., Petersen, S., Simonyan, K., Crossan, S., Kohli, P., Jones, D. T., Silver, D., Kavukcuoglu, K. and Hassabis, D. (2020), 'Improved protein structure prediction using potentials from deep learning', *Nature* **577**(7792), 706–710.
URL: *https://doi.org/10.1038/s41586-019-1923-7*

Sensoy, A. (2019), 'The inefficiency of bitcoin revisited: A high-frequency analysis with alternative currencies', *Finance Research Letters* **28**, 68–73.
URL: *https://www.sciencedirect.com/science/article/pii/S1544612318302320*

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014), 'Dropout: A simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research* **15**(56), 1929–1958.
URL: *http://jmlr.org/papers/v15/srivastava14a.html*

Staudemeyer, R. C. and Morris, E. R. (2019), 'Understanding lstm – a tutorial into long short-term memory recurrent neural networks'.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. and Fergus, R. (2014), 'Intriguing properties of neural networks'.

Thompson, N. C., Greenewald, K., Lee, K. and Manso, G. F. (2020), 'The computational limits of deep learning'.

Urquhart, A. (2016), 'The inefficiency of bitcoin', *Economics Letters* **148**, 80–82.
URL: *https://www.sciencedirect.com/science/article/pii/S0165176516303640*

Wei, J. N., Duvenaud, D. and Aspuru-Guzik, A. (2016), 'Neural networks for the prediction of organic chemistry reactions', *ACS Central Science* **2**(10), 725–732. PMID: 27800555.
URL: *https://doi.org/10.1021/acscentsci.6b00219*

Yun, K., Huyen, A. and Lu, T. (2018), 'Deep neural networks for pattern recognition'.

Zaremba, W., Sutskever, I. and Vinyals, O. (2015), 'Recurrent neural network regularization'.

Zeiler, M., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J. and Hinton, G. (2013), On rectified linear units for speech processing, pp. 3517–3521.

Appendices

A Data Scaling

Before training a neural network, the data should be scaled. This is because the data used may be in different units such as dollars and hours. Also, input variables may be of different scales, e.g. the volume traded of a stock could be in the millions, while the asset price could be less than 100. A problem with having inputs that may differ on a large scale is that the weights that of the neural network could also be large, which can lead to the networks being unstable or biased by inputs that are measured in larger units. Preprocessing data may also speed up computation (Bishop, 2006). To scale the data, the MinMax scaler from the Scikit-learn Python library for machine learning is used. The MinMax scaler works as follows.

$$x_{std} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{13}$$

$$x_{scaled} = x_{std} * (max - min) + min \tag{14}$$

where x is the unscaled value, x_{min} and x_{max} are the smallest and largest values within the dataset respectively and with min and max being being the bounds of our scaled values which for this study are zero and one.

However, one must be careful the data is scaled in a way to maintain the training, validation and unseen aspects of the data. Hence we first scale the training data and then apply this scaler to the validation and unseen test data. We do this as we do not know what future data will look like in practice so it cannot be included in the data scaling process when fitting the scaler to prevent introducing a look ahead bias.

B LSTM Calculation

Formally, the computation within an LSTM model is as in the following equations which are performed at each time step. These equations detail the full workings of an LSTM with forget gates. The input gate, i_t and output gate, o_t multiply the input, g_t and output, h_t of the cell and the forget gate, f_t multiplies the previous state of the cell. The gate activation function is usually the logistic sigmoid, where the gate activations are between 0 (closed) and 1 (open). The input and output node activation functions are typically tanh or logistic sigmoid also. The input node utilises the activation from the input layer x_t and the hidden state h_{t-1} at time $t-1$. The input gate then uses its activation function along with the data at time t and output from the hidden units at time $t-1$. The input gate then multiplies the value of the input node and since it is sigmoidal is between zero and one, controls the flow of the signal it multiplies. The internal state contains a self-recurrent connection and is denoted s_t, where $\odot$ denotes the element-wise product and f_t is the forget gate. The forget gate multiplies the internal state at $t-1$ and can dispose of all the contents in the past. The output from the cell is obtained by multiplying the value of s_c by the output gate o_c.

$$g_t = \sigma(W^{gX}x_t + W^{gh}h_{t-1} + b_g) \tag{15}$$

$$i_t = \sigma(W^{iX}x_t + W^{ih}h_{t-1} + b_i) \tag{16}$$

$$f_t = \sigma(W^{fX}x_t + W^{fh}h_{t-1} + b_f) \tag{17}$$

$$o_t = \sigma(W^{oX}x_t + W^{oh}h_{t-1} + b_o) \tag{18}$$

$$s_t = g_t \odot i_t + s_{t-1} \odot f_t \tag{19}$$

$$h_t = \phi(s_t) \odot o_t. \tag{20}$$

where h_t is the value of the hidden layer at time t, h_{t-1} is the output of each memory cell in the hidden layer at time $t-1$. The weights $[W^{gX}, W^{iX}, W^{fX}, W^{oX}]$ are the connections between the inputs x_t with the input node, input gate, forget gate, and output gate respectively. Similarly $[W^{gh}, W^{ih}, W^{fh}, W^{oh}]$ are the connections between the hidden layer with he input node, input gate, forget gate, and output gate respectively. The bias terms for components of the cell are given by $[b_g, b_i, b_f, b_o]$.

C Regularisation

Regularisation is the term given to various techniques with the goal of preventing overfitting and ideally improving accuracy in a machine learning context. Generalisation error refers to the performance of our neural networks on unseen data that is not part of the training process. Crucially, regularisation aims to reduce the generalisation error but not the training error of our models. One form of regularisation we will apply to the neural networks built in this work is dropout regularisation. In simple terms, dropout essentially means that during the training phase of the model that individual neurons are dropped out of the neural network with a probability $1 - p$, or kept with a probability p. This regularisation improved the performance of feedforward neural networks in a variety of supervised learning tasks such as vision and speech recognition while also reducing signs overfitting (Srivastava et al., 2014). Zaremba et al. (2015) found that dropout regularisation can successfully reduce overfitting when used for RNNs with LSTM units.

D Network Architectures

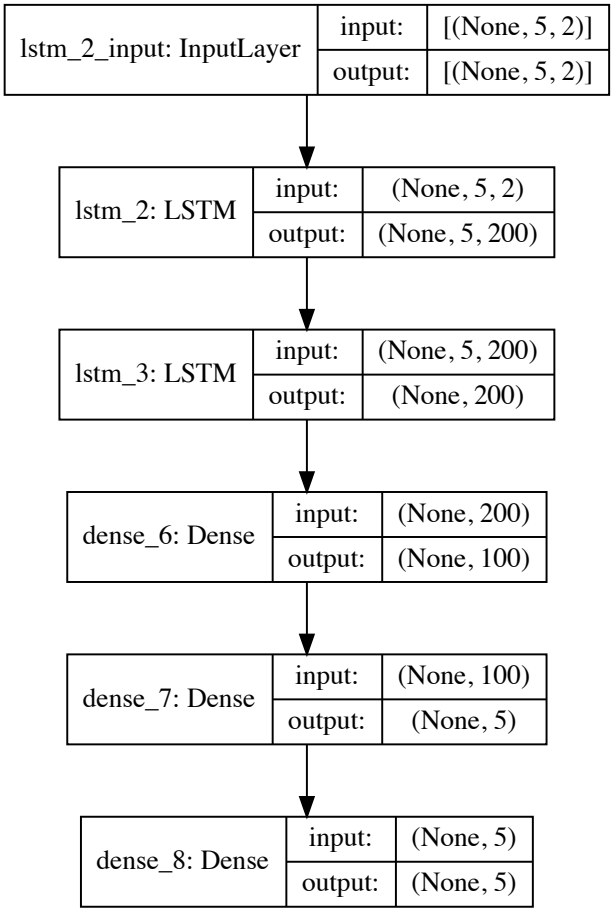

Figure D.1: Graphical Representation of LSTM3 Architecture

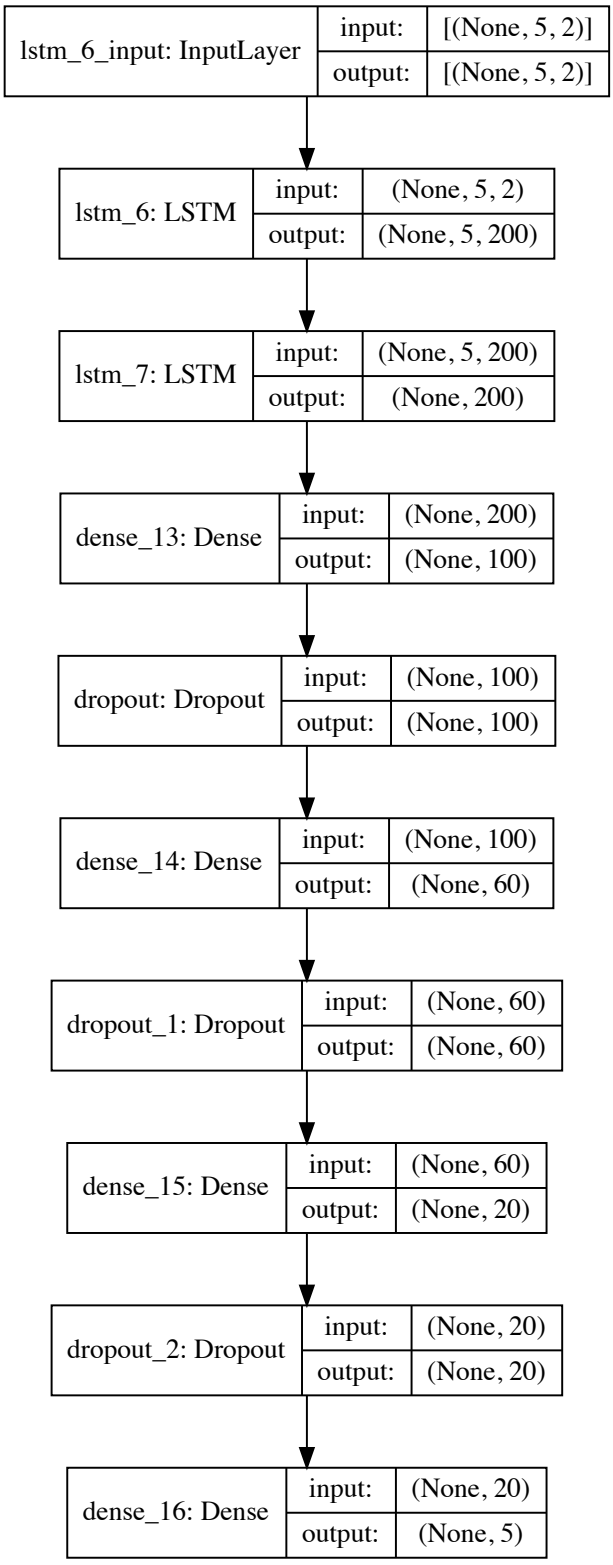

Figure D.2: Graphical Representation of LSTM5 Architecture with Dropout Regularisation

E Model Building RMSE Results

The following is a table of the RMSE on the out-of-sample data for each LSTM model built in this work.

Table E.1: RMSE Values for Out-Of-Sample Predictions Per Model

Model	RMSE
LSTM1	0.9307
LSTM2	0.3348
LSTM3	0.1953
LSTM4	0.1179
LSTM5	0.3608

F ARIMA Model Building Process

To build our ARIMA model the training and validation datasets were taken without applying the windowing process used for the RNN, and created the following decomposition plot in Figure F.1

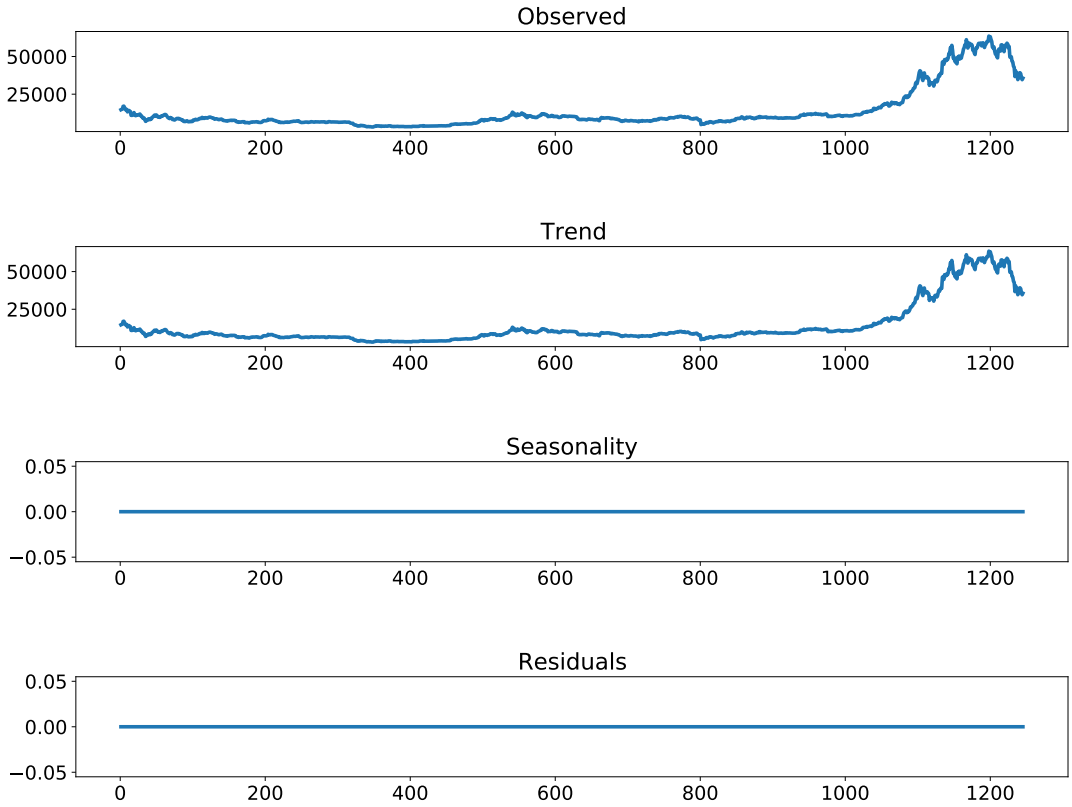

Figure F.1: Decomposition of BTC Time Series.

From the plots the only present trend is simply the price time series with no other clear linear trend. There is also no seasonality with the residuals being equal to zero.

To determine if the time series was stationary an Augmented Dickey-Fuller (ADF) test was carried out to test if a unit root is present in the time series. With a test statistic of 4.76 and a p-value of 1.0 we fail to reject the null hypothesis of the test. Hence, there is a unit root present in the data and the data is non-stationary. The data was then differenced once we and ran the ADF test again, which had a p-value <0.0001, hence we reject the null hypothesis and conclude that the differenced data is now stationary.

We then created plots of the autocorrelation function (ACF) and partial autocorrelation function (PACF) for the differenced series. These figures are seen below in Figure F.2 and Figure F.3

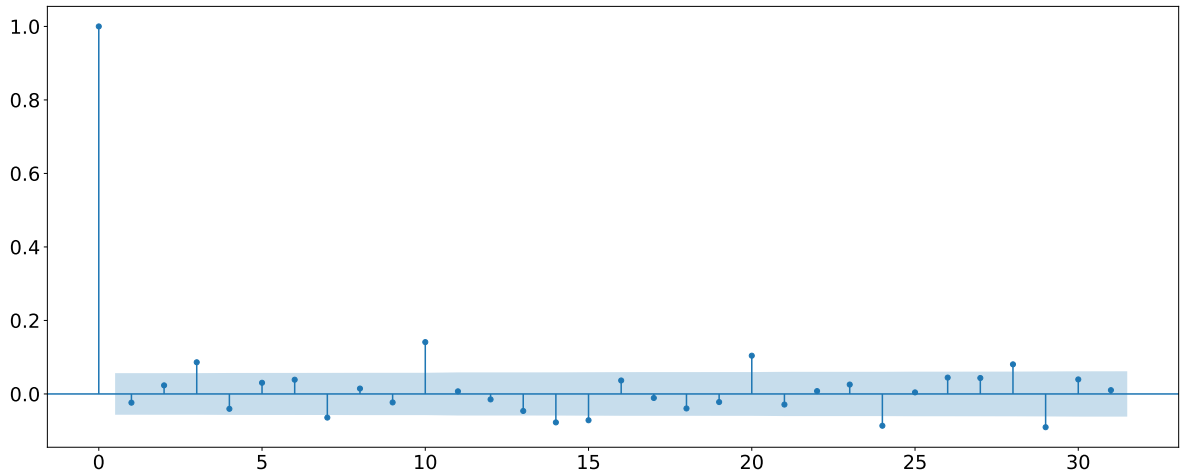

Figure F.2: ACF for Differenced Closing Prices.

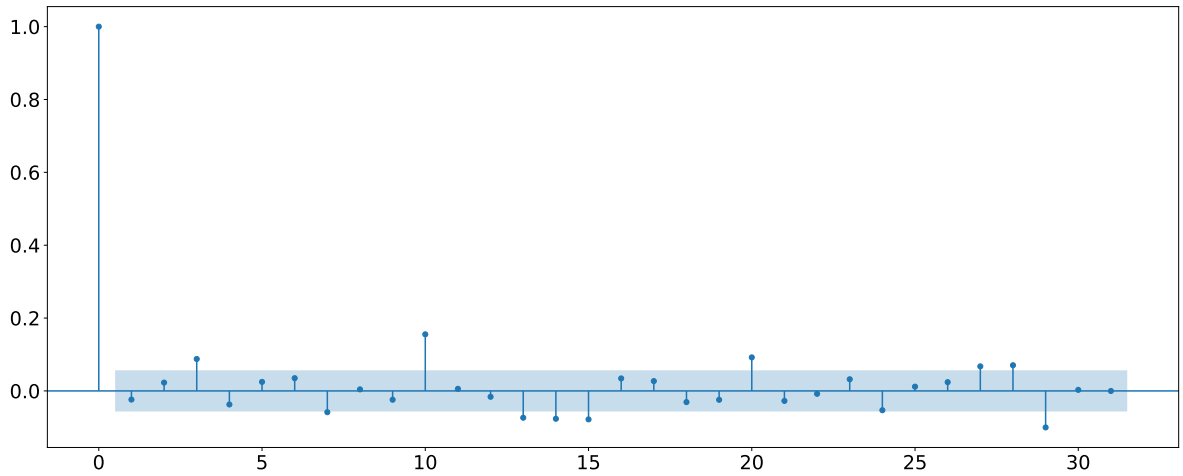

Figure F.3: PACF for Differenced Closing Prices.

Although there are no significant ACF or PACF values until lag 3, while experimenting with different models we found that an ARIMA(2,1,2) performed the best in terms of lowering AIC from a number of different models we tested with varying combinations of AR and MA parameters. Since the prices were differenced once the d parameter is equal to 1 in this model. Table F.1 details the values of the AR and MA terms in the model along with their p-values and standard errors.

Table F.1: ARIMA Model Parameters

Parameter	Coefficient	std error	p-value
AR 1	-0.5831	0.009	0.000
AR 2	-0.9745	0.009	0.000
MA 1	0.5488	0.009	0.000
MA 2	0.9769	0.007	0.000

This model will be used as a comparison versus the RNN created to evaluate its prediction accuracy and how robust the ARIMA model is when we add Gaussian noise to the data also.