Using Artificial Intelligence and Machine Learning to predict Bitcoin Price

UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Cian Doyle - 16165608

Supervisor: Ashish Sai

Computer Games Development

Abstract

Blockchain solutions have seen widespread adoption in numerous industries. The price of Bitcoin is subject to constant manipulation by a multitude of factors. In this, We propose using Artificial Intelligence techniques such as LSTM to predict the price of Bitcoin based on numerous factors such as historical trends. We will identify an appropriate training model for the price prediction. We will analyse available data to check which factors correlate to the price of Bitcoin. The data that is identified as being correlated will be used to train the prediction model. We intend on examining the accuracy of the model's prediction against real-world values.There have been many attempts to do this before, these projects have mainly aimed at predicting long term price changes, whereas this project will focus on short term price fluctuations. The results show that it is possible to lucratively predict short term price changes in Bitcoin

Acknowledgements

I would like to extend my gratitude to my supervisor Ashish Sai.
Thank you for your advise and patience throughout this process.
I would also like to thank Farshaad Toosi for his help.

Contents

List of Figures

1

Introduction

1.1 Background

1.1.1 Bitcoin

Bitcoin is a cryptocurrency, a form of electronic currency. It was created in January 2009 by an anonymous person using the Pseudonym Satoshi Nakamoto, Nakamoto (2019). It's decentralized and is not controlled by any single person or group, there is debate whether this is a good or bad thing, Bitcoin enthusiasts argue that decentralization means the currency is not subject to the whims of any government or ideology, this can prevent harmful monetary policies that have caused long term inflation of Fiat currencies. The decentralization of the Bitcoin means that no central authority has any way to influence the supply and value of Bitcoin which is why people argue it is the currency of the future. The trade off of decentralization is that decentralized systems are usually less efficient than centralized ones. Unlike physical currencies which are backed by some form of physical commodity, Bitcoin is backed only by the math that drive it. The Bitcoin network is run on a set of distributed computers across the world called nodes, these nodes run the open source Bitcoin software. This network of computers is the largest and most powerful computer network in the world, Cohen (2013).Bitcoin gained widespread attention in 2017 when the price rose massively in the latter half of the year. It is not well known whether Bitcoin and other cryptocurrencies are subject to manipulation by various factors. We hope to find some of the correlated factors to help with our predictions.

1.1.2 Neural Networks

Neural networks are algorithms that are modelled on the human brain. They are designed to recognise patterns in the inputted data. There are many types of Neural networks that are used for different purposes. All neural networks are comprised of nodes which resemble individual neurons in the brain. These nodes, like neurons are interconnected and can transmit data between one another. The connection between a node and another node is called an edge. These edges have weights that determine how important the input is in calculating the output, As the neural network is fed information and learns, these weights will be adjusted to optimize the model. Neural networks are used in a variety of areas such as speech recognition, medical diagnosis and even playing games The concept of Neural networks in computing was first thought of when Warren McCulloch Walter Pitts wrote about how neurons might work, they made a model using circuits in 1943. The first artificial neural network was created in 1958 Frank Rosenblatt, He created the simplest and oldest neural network that exists today, the perceptron. An improved recurrent neural network - LSTM was suggested by Hochreiter and Schmidhuber (1997). LSTM networks have the ability to selectively remember and "forget" different bits of information. This gives it the ability to interpret new data based on the data it has seen in the past

1.2 Objectives

The objectives of this project are as follows:

- Research previous similar projects to aid with this project
- Identify factors that influence Bitcoin price on an minute basis
- Research and find the optimal type of Neural network for the model
- Discover if it is possible to make an accurate model for the prediction of Bitcoin price on an minute time frame

1.3 Motivation

There are already several projects out there that attempt to predict the price of Bitcoin on a high time frame, We will attempt to predict it on a small time frame. The motivation for this project is that we want to see if it is possible to predict the short term price changes with the data that can be gathered. This is more

challenging than predicting high time scale as it is much harder to obtain small time frame data on Bitcoin and other factors. Predicting the price changes on small time frames can be extremely difficult as there is a multivariate of factors that can drastically influence Bitcoin price, such as Political news which can be difficult to account for. On the flip side, if an accurate model is produced it could be very lucrative, as it could be used to margin trade Bitcoin , ramping up profits.

1.4 Related Works

There have been several related works in this area, all of which use higher time frames than we plan on using. Nevertheless it is very useful to look at these to consider their findings, which may help us in our project. The following paper: Struga and Qirici (2018), gives a high level understanding of what needs to be done , as well as presenting their findings. This project does not seem to be successful in their attempts to predict prices, as their price predictions seem to lag actual values, meaning their model can only predict price movements after they have happened.
The following paper found that LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit) were both effective in predicting Bitcoin price. *"We also conclude that recurrent neural network models such as LSTM and GRU outperform conventional time-series approaches like SARIMA for price prediction. With limited data, neural networks like LSTM and GRU can regulate past information to learn effectively from non-linear patterns. "* Dutta et al. (2019).
Again, In the following paper: McNally et al. (2018) , LSTM was identified as being the most accurate model tested, achieving a 52% accuracy. This result seems good but they do not specify if this is on training or test data, and they also state that the model has a 8% RMSE (Random Mean Squared Error) on test data, which is worryingly high and sounds like over-fitting of training data. The following paper, Mangla (2019), tested four different methods of price prediction, and found the only one that could predict price changes with over 50% accuracy was ARIMA (Auto Regressive Integrated Moving Average). They did state though that

"*ARIMA performs well for next days predictions but performs poor for longer terms like given last few days price predict next 5-7 days prices*".

Fazeli (2019) found that using technical indicators such as Relative Strength Index

and Volatility reduced improved their predictions over not using indicators, but were still unable to create a model capable of making profit.

1.5 Dataset

The majority of the dataset used in this paper was retrieved from dukascopy.com. This included minute data for intrinsic Bitcoin data such a price and volume, but also a variety of extrinsic factors such as gold price, oil price, Us dollar index among many others. The data range is from October 2018 - August 2019 on an minute time frame, this comprises 980,000 data points on each factor, Totalling 21,560,000 data points. The below table illustrates our dataset size compared to related works. Many related works do not disclose their dataset size so only the ones that do are included. We used JASP, an open source statistics software to calculate the Pearson correlation of each of the factors. Pearson correlation, also known as the Pearson product-moment correlation coefficient is a measure of the linear correlation between two variables, it ranges from -1 to 1, with -1 being a strong negative correlation, 0 being no correlation, and 1 being a strong positive correlation. For our purposes it did not matter if the correlation was negative or positive. We also used data from this dataset Badiola (2019) which comprises of a sentiment analysis of 17 million tweets related to Bitcoin. This dataset had filtered out bots, competitions , giveaways etc which can reduce the effectiveness of data. The dataset used VadarSentiment cjhutto (2020) to analyse the sentiment of each tweet and assign a score every minute for how positive or negative the content of the tweets was. We also used Pytrends GeneralMills (2019) to collect Hourly google search data on Bitcoin, the API did not provide functionality to collect minute data so instead we checked for correlation to an hourly price of Bitcoin.

The list of all factors, correlated and those found to have no correlation, can be found in Appendix A.

Author	Dataset size (Rows)
This Model	980,000
Fazeli (2019)	1200
Struga et al (2018)	1850
Dutta et al (2019)	3469

Chapter 2

Researching the Neural Network

2.1 Choosing a library

2.1.1 Keras

The first Python library that we considered was Keras, Chollet (2015). Keras is a module that wraps around TensorFlow and makes it extremely simple to use, in Keras you can create a neural network in a few lines of code making it an attractive choice. Keras was our first choice as we were familiar with it from past work. We instead opted to use PyTorch originally as the size of the input data was very large, PyTorch is more optimized than Keras for parallelizing data across multiple GPU's, Wrosinski (2017), to drastically speed up the training of the model. We later opted to use Keras again as PyTorch was not producing optimal results.

2.1.2 PyTorch

PyTorch, Paszke et al. (2016), is another Python module that is a wrapper module for TensorFlow, although it is more low level than Keras. It requires a deeper understanding of TensorFlow than Keras, but it can be finely tuned for more custom models. The main reason to use PyTorch over Keras is the ability to spread the model over several GPUs with the DataParallel class over even over several machines with the DistributedDataParalell class. Keras does have its own utility for this called multigpumodel but the general consensus is that PyTorchs' equivalent is far superior. PyTorch has also been shown to train faster than equivalent Keras models which is extremely important for large models such as the one we are using. *"It seems that there is no significant difference in speed between Pytorch and Tensorflow, when training well-known CNN's. But there is*

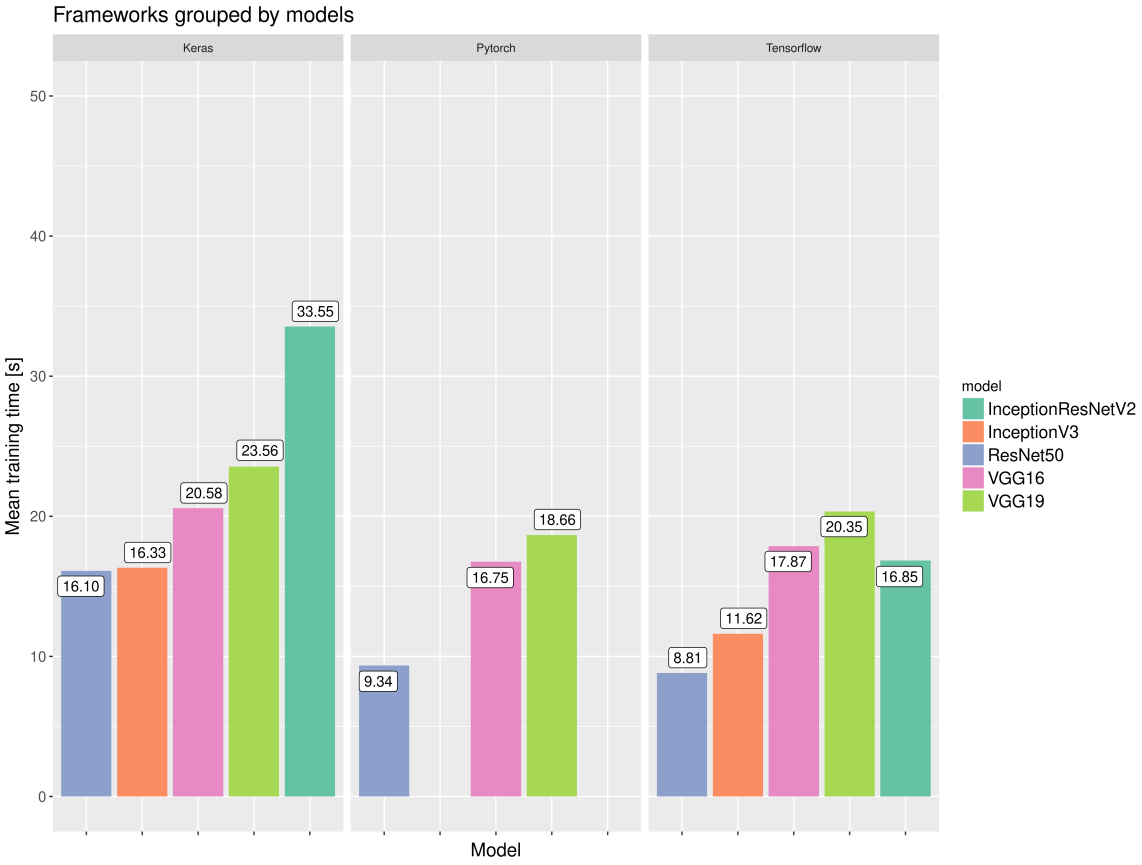

Figure 2.1: Comparison of DL Frameworks Wrosinski (2017)

one, which will be felt, when Keras is chosen over those ones." Wrosinski (2017)

2.2 Neural Network Structure

2.2.1 Hidden Layers

There's not really a guideline on how many hidden layers there should be for each type of problem, but the general consensus is that 1 hidden layer is generally sufficient for most problems, while two layers might be required for more complex problems Stathakis (2009). Having more than 2 hidden layers has been shown to improve performance for specific problems such as Zip Code Recognition, Le Cun et al. (1989). Having too many hidden layers and too many neurons in those layers can cause the network to turn into a giant memory bank that will memorise the entire input which will lead to severe over fitting, to combat this we will only use two hidden layers in the network. The only real way to find the ideal number of layers as well as neurons in those layers is through trial and error, the problem with this is that the training of our model took a a considerable amount of time, If

we were to explore all the combinations possible of hidden layer sizes and neurons in the layers, it would take more time than we have available to us.

2.2.2 Hidden Layer Size

The hidden layer size in an LSTM neural network can have an impact on the accuracy and training capacity of the network, A larger size is required in more complex problems but there is a trade off in terms of the time it takes to train the network. Since the problem being addressed here is a complex one, the hidden layer size should be larger. The formula below is commonly used to approximate the hidden layer size

$$N^h = \frac{N^s}{\alpha * (N^i + N^o)}$$

N^s = The number of samples in the training data, N^i = The number of input neurons, ${}^N o$ = The number of output neurons, α = An arbitrary scalar usually between 2-10.

Using this formula , The number of hidden layer neurons for this project comes out to 207. After tweaking the model we found that decreasing the number of hidden layer neurons to 50 did not have any effect on accuracy but it significantly increased training speed.

2.2.3 Batch Size

The batch size is the amount of samples fed into the neural network in one go. If you choose a large batch size, more samples will be fed into the network, Larger batch sizes will help the model to train faster, and they also suffer less from gradient competition. Gradient competition is where

"*One sample wants to move the weights of the model in one direction while another sample wants to move the weights the opposite direction Therefore, their gradients tend to cancel and you get a small overall gradients. Perhaps if the samples are split into two batches, then competition is reduced as the model can find weights that will fit both samples well if done in sequence*" Shen (2018). This is illustrated in Figure 2.2

A smaller batch size is not as good at generalisation as it only adjusts weights based on the current batch, this is detrimental to the model but it can sometimes help escape local minima. Due to the fact that this model will be trained in parallel across multiple GPU's, the batch size will need to be greater than or equal to the

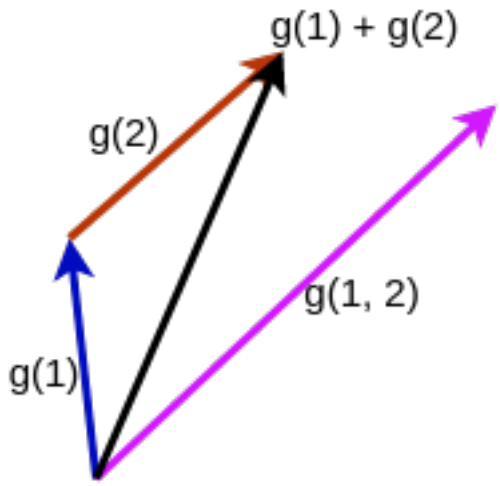

Figure 2.2: Gradient Competition, Shen (2018)

number of GPU's, this is because at least one batch will have to be sent to each GPU, or else it will not be utilised, Since we have six GPU's , We started with a batch size of 6, processing one batch per GPU. This was not producing optimal results so we experimented and found that a batch size of 128 was producing better results.

2.2.4 Dropout

Dropout was proposed by Hinton et al. (2012), The purpose of dropout is to reduce over-fitting of training data by temporarily removing neurons in the neural network.Srivastava et al. (2014) used dropout with feed-forward neural networks and noted the following "*each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units. This should make each hidden unit more robust and drive it towards creating useful features on its own without relying on other hidden units to correct its mistakes.*". This increases the units ability to generalise and will make it perform better when dealing with unseen data. Pham et al. (2013) found that applying dropout on mulitple LSTM layers can decrease error rates by 40 percent, while also noting that "*dropout always improved the error rates*". Although this particular study was done on a handwriting recognition database, Pham et al. (2013) state that this is not limited to that particular subject "*It should be noted that although our experiments were conducted on handwritten datasets, the described technique is not limited to handwriting recognition, it can be applied as well in any application of RNNs.*" For the neural network we are using, We will use a dropout of 0.3 to stop over-fitting of training data, since there is only two LSTM layers in my

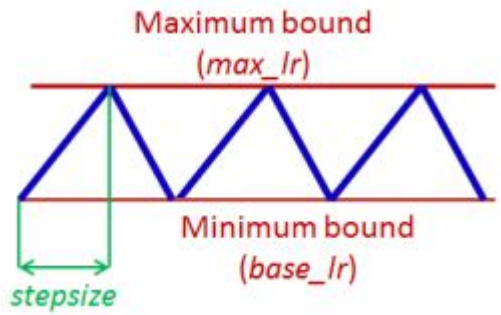

Figure 2.3: Cyclic Learning Rates diagram Paul (2018)

network, we can only apply dropout to the first layer as you cant apply dropout to the last layer.

2.2.5 Learning Rate

Another Hyper-Parameter that can have a massive impact of the performance of the model is the learning rate. The learning rate specifies how fast the model will change the weights associated with input parameters. Too high a learning rate will lead to the model placing too high an emphasis on the current inputs while ignoring old ones, and too low a learning rate can lead to the model ignoring new inputs if they conflict with older ones. So, what is the right learning rate?, Well like a lot of things in neural networks, it depends on the network and is usually found through trial and error. There is a technique that can aid in finding the optimal learning rate, that is Cyclic Learning rates

2.2.5.1 Cyclic Learning Rates

Cyclic Learning Rates is a concept introduced by Smith (2015). The point of CLR is to eliminate the need to "Guess" the right learning rate or to use trial and error. CLR changes the learning rate in a cyclical manner for each batch, as opposed the usual constant rate or decaying rate. The learning rate in CLR is kept between a minimum and maximum bound and will vary between these bounds. This works because having a higher learning rate inside an epoch can help to overcome a local minima that might have been encountered by a lower learning rate in the same epoch.

This technique was used by Fast.ai to beat Google's image net model, this is particularly impressive as Fast.ai spent 40 dollars renting Amazon servers to run the model on in stark contrast with the massive computing power available to Google, Knight (2018).

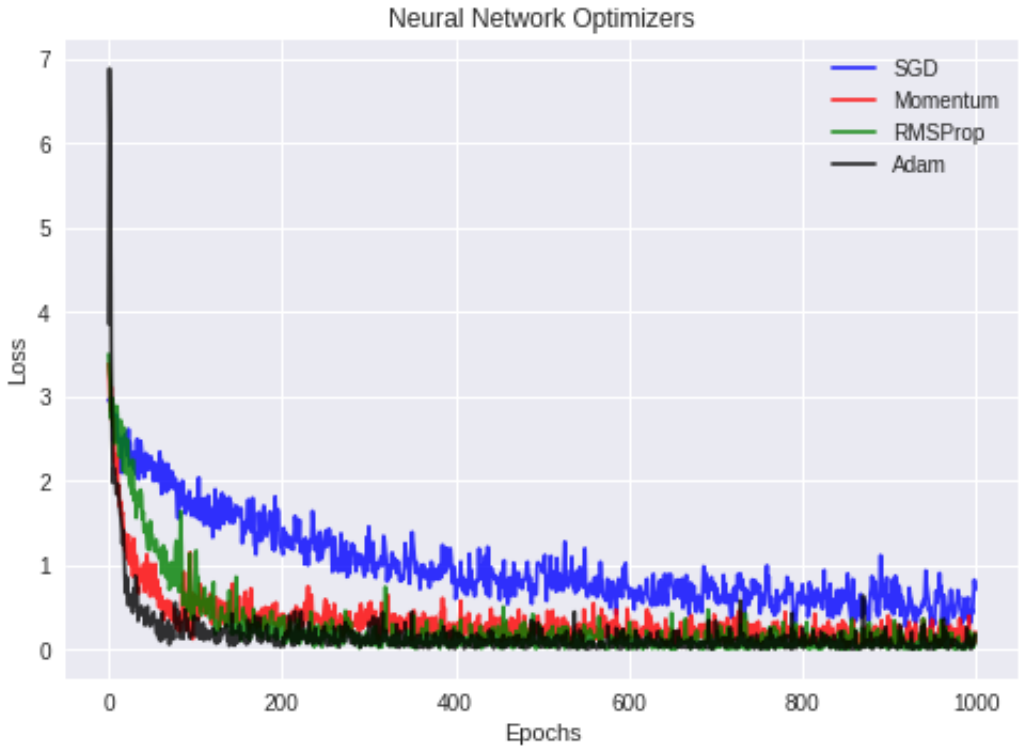

Figure 2.4: Optimizer training comparison Smolyakov (2018)

2.2.6 Optimizer

Another important Hyper-Parameter is the Optimizer, The optimizer changes the models parameters (weights) to try and minimise the loss function of the model. It works in tandem with the learning rate, the learning rate tells the optimizer how fast or slow to change the weights in response to new data. There are numerous different optimizers that can be used. Stochastic gradient descent is the traditionally used optimizer in machine learning, it is a reliable optimizer that is very good at generalisation, however as it can be seen in Figure 2.4, it is slower to achieve minimal loss compared to other optimizers. This could be a problem for us due to our large dataset size , therefore we chose Adam Optimizer. The optimizer Adam has been shown to perform well across a variety of tasks compared to other options, as well as being efficient and having low memory requirements, Smolyakov (2018). The formula for Adam is given as:

$$g = \frac{1}{m}\nabla \sum_{i} L(f(x^i;\theta), y^i)$$

$$m = \beta_1 m + (1-\beta_1)g$$

$$s = \beta_2 s + (1-\beta_2)g^T g$$

$$\theta = \theta - \epsilon_k * m/\sqrt{s+\epsilon}$$

The default values are

$$\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e - 8.$$

A full explanation of Adam can be found online: Kingma and Ba (2014)

2.2.7 Loss Function

A loss function measures how well the model is performing. This works in tandem with the optimizer, which attempts to reduce the loss function, and also the learning rate, which defines how the model responds to new data. For our purposes, Mean Squared Error works well. MSE works by squaring the difference between the predicted values and the actual values and gets the average of this across all predictions, its defined as:

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2$$

Chapter 3

Implementing the Neural Network

3.1 Pre-Processing the data

3.1.1 Validating the Data

There was several steps that we had to take to process the data before it was the correct format for the neural network. The first one was to run through the dataset and make sure that all the time frames for the data matches so we were comparing data for the same time, when we ran the python script over the dataset.We found that were in fact some entries missing from some of the datasets, the script filled these in with the last valid value. In total we had to fill in approximately 20,000 entries equating to around 1 percent of the total dataset size. This may have had a small impact on the accuracy of the model due to incorrect data but it should not be significant. Not all of this data was used in the neural network as some of the data was found to have no correlation to the price

3.1.2 Data Normalization

Neural networks require you to scale your data down usually to values between -1 to 1 or 0-1. The reason for this is

"*If one input has a range of 0 to 1, while another input has a range of 0 to 1,000,000, then the contribution of the first input to the distance will be swamped by the second input. So it is essential to rescale the inputs so that their variability*

reflects their importance", faqs.org (2002),

This is very simple in Python as there is a module called sklearn which provides Scalar objects which can transform data into specified ranges in one line. This is MinMaxScalar and the formula is defined as follows:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

During training, the scalar object should be saved for later use with the test data as the test data needs to be scaled in relation to the training data.

3.1.3 Adding lagged variables to the dataset

We wanted to append the last 30 minutes of data onto an entry as we are trying to predict short term price changes by using recent price data, So in this case we are using the last 30 minutes of price data to predict the next minutes price changes. A method was made in python that appended this information onto each entry of the array. The original size of the array was 1.04 million rows x 10 columns, after appending the last 30 minutes this went up to 1.04 million rows x 300 columns putting the total data set size up to around 312 million data points, although really its less as a lot of the data is duplicated. Several implementations of neural networks that we found online had a slightly different approach, they did not append the lagged variables onto a row, instead they used for example the last 299 entries to predict the 300th entry. We did not like this approach as you are massively cutting down on the amount of training the neural network is doing as instead of predicting every entry it would only predict every 300th entry.
After switching our framework back to Keras, we needed to reduce the dataset size to speed up training time. We reduced our lagged variables from 30 minutes down to 10. This may have reduced our accuracy (It also could have reduced noise) , but the trade off was that our training time was drastically approved, which was needed due to hardware limitations.

3.2 PyTorch

3.2.1 Converting dataset to Tensors

PyTorch requires the input data to be in the form of Tensors or list of Tensors. Tensors in TensorFlow are similar to matrices. We created a method that transformed my data in Tensor tuples, the first entry was a list of the inputs and the second tensor was the corresponding output.

3.2.2 Creating a Dataset and DataLoader

Due to the fact that we were planning to running my model in parallel across several GPU's , We needed to implement a way to quickly change batch sizes. The solution to this was to use PyTorch Dataset and DataLoader classes which allow quick changing of batch sizes. The DataLoader class works in conjunction with Dataset to return the data in the required format. There are a couple of other parameters in DataLoader that are useful for other projects, it provides functionality to randomly shuffle the dataset, normally this would help with things like generalization and class bias, but since our data is time series , we cannot shuffle it.

3.2.3 Defining the model

From our previous research , We knew what structure we wanted in terms of layers, hidden layer size etc, looking at related works and research gives structure to the project prior to building it as their are an extremely large amount of combinations of structures that are possible. The structure that we implemented was a stacked LSTM with a linear output layer. The LSTM had two layers with a dropout of 0.5. The dropout is only applied to the first of the LSTM layers, these layers each originally had a hidden size of 200, but this was changed to 50 after tweaking the model. The structure of the model is illustrated in Figure 3.1, Note this is a basic outline it does not show the complex hidden layer structure. We ran into two problems when coding the model , the first was the forward function . The forward function in the model takes in the inputs, passes them through the network and then returns the prediction for those inputs. The first problem we encountered was a Tensor shape issue which was quickly resolved. The second problem occurred when we did not reset the hidden state between between predictions which caused the model to run very slowly.

```
        # Neural network class
class LSTM(nn.Module):
    def __init__(self, input_size=inputSize,hidden_layer_size=200,
    output_size=batchSize):

        super().__init__()

        self.hidden_layer_size = hidden_layer_size
        self.lstm = nn.LSTM(input_size, hidden_layer_size,
        num_layers=2,dropout=0.5)
```

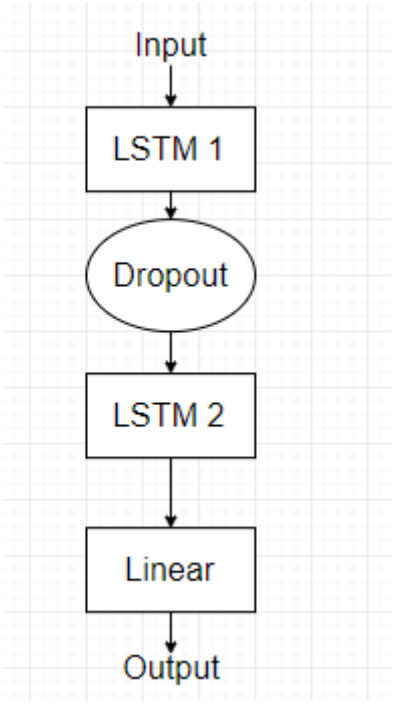

Figure 3.1: Model structure

```
        self.linear = nn.Linear(hidden_layer_size, output_size)

        self.hidden_cell = (torch.zeros(2, 1, self.hidden_layer_size)
                            torch.zeros(2, 1, self.hidden_layer_size))

    def init_hidden(self):
        return (torch.zeros(2, 1, self.hidden_layer_size),
                torch.zeros(2, 1, self.hidden_layer_size))

    def forward(self, input_seq):
        lstm_out, self.hidden_cell =
        self.lstm(input_seq.view(len(input_seq), 1, -1), self.hidden_
        predict = self.linear(lstm_out.view(len(input_seq), -1))
        return predict[-1]
```

3.2.4 The Training Loop

With the model implemented, We started training and testing it just to test if it was working, what we realised was the training predictions were good but the test predictions were not good, a clear sign of over fitting, to get a window into the system, We implemented test predictions into the training loop. Meaning the model did one iteration of training, then tested, then another iteration etc . This is different to the usual approach of just doing all the training iterations and then test once at the end. To make sure the test does not create gradi-

ents and affect the network weights, The test code should be surrounded with , "with torch.nograd", this stops the network accumulating gradient. The model is also switched between training and evaluation mode, the evaluation mode removes dropout layers from the network while testing. This approach helped us to be able to see if the network was over fitting much quicker. The training and test loop also included collecting data to be used for metrics and visualization. The training loop also included several other steps which will be shown in the code below such as: Zeroing gradients, Clearing the hidden state, Making predictions, Calculating the loss, Back-propagating the loss, Clipping the gradients and stepping the optimizer. Clipping the gradients can help solve the problem of exploding/vanishing gradients faced by some Recurrent neural networks.

```
for i in range(epochs):
    loss = 0
    model.train()
    for seq, labels in train_loader:
                # Send inputs + labels to the device
        seq = seq.to(device)
        labels = labels.to(device)
        labels = labels.view(-1)

                # Zero the gradient and initialize hidden state
        optimizer.zero_grad()
        model.hidden_cell = model.init_hidden()

                # Predict and calculate loss
        y_pred = model(seq)
        single_loss = loss_function(y_pred, labels)

                # Backpropogate
        single_loss.backward()
        loss = single_loss.item()
                #Clip the gradient
        nn.utils.clip_grad_norm_(model.parameters(), 5)
        optimizer.step()
```

3.3 Keras

As previously mentioned, the PyTorch model was not providing satisfactory results so we rebuilt the same model in Keras.

3.3.1 Defining the model

Defining a model in Keras is considerably easier than PyTorch. We did not have to explicitly convert the Dataset into Tensors as it was done automatically. We still had to scale the values between 0-1. We also did not have to create any kind of Dataset or DataLoader or equivalent as Keras allows you to flexibly change the batch size.

```
model = Sequential()
model.add(LSTM(activation='relu',units=200,return_sequences=True,
input_shape=(train_window,n_features),dropout=dropout))
model.add(LSTM(activation='relu',units=200))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

One difference between the Keras model and the PyTorch one is that in the Keras model we used the 'relu' activation, which is Rectified Linear Unit, in the PyTorch model we used the standard Sigmoid activation. Sigmoid activation can suffer from the vanishing gradient problem. The vanishing gradient problem occurs when the weight gradients are all less than 1, multiplying all these numbers together results in a tiny number that results in no learning of weight values. This problem is solved with ReLu AdventuresInMachineLearning (2018).

Chapter 4

Analysis of Results

4.1 Introduction

For the testing of the Keras model, Three metrics were selected. These metrics are Average difference, which indicates the average difference in percentage between the predicted results and the actual ones.The next metric is Prediction Accuracy, This is a measure of the percentage of predictions that the model got right. This was measured by checking a predictions trend over the last two minutes (E.g Up, Down or flat) and comparing that with the actual trend for that time. The final metric selected was Profit/Loss generated by predictions. The latter carries more weight as a model can be accurate without correctly following the trend of Bitcoin, meaning it is not useful for stock trading. The P/L was calculated by checking a prediction against the actual change and increasing or decreasing the P/L based on this. These results were collected on a dataset size of 1 Million data points. In conjunction with this, we also used the Mesa framework to simulate how other strategies would perform. The four strategies tested were: Following our model, Doing the inverse of our model (I.E when our model says buy, this would sell) , A random trader who has a 50/50 chance to buy or sell, and finally a Buy and Hold strategy. The Buy and Hold strategy is considered to be the baseline to beat.

4.1.1 Mesa Framework

The Mesa Framework "*allows users to quickly create agent-based models;visualize them using a browser-based interface; and analyze their results using Python's data analysis tools.*"mesa.readthedocs.io (2016). The Mesa framework allowed us to quickly create a simulation with several agents to see how they would perform in a simulation of Bitcoin prediction. It also provides an easy way to collect data

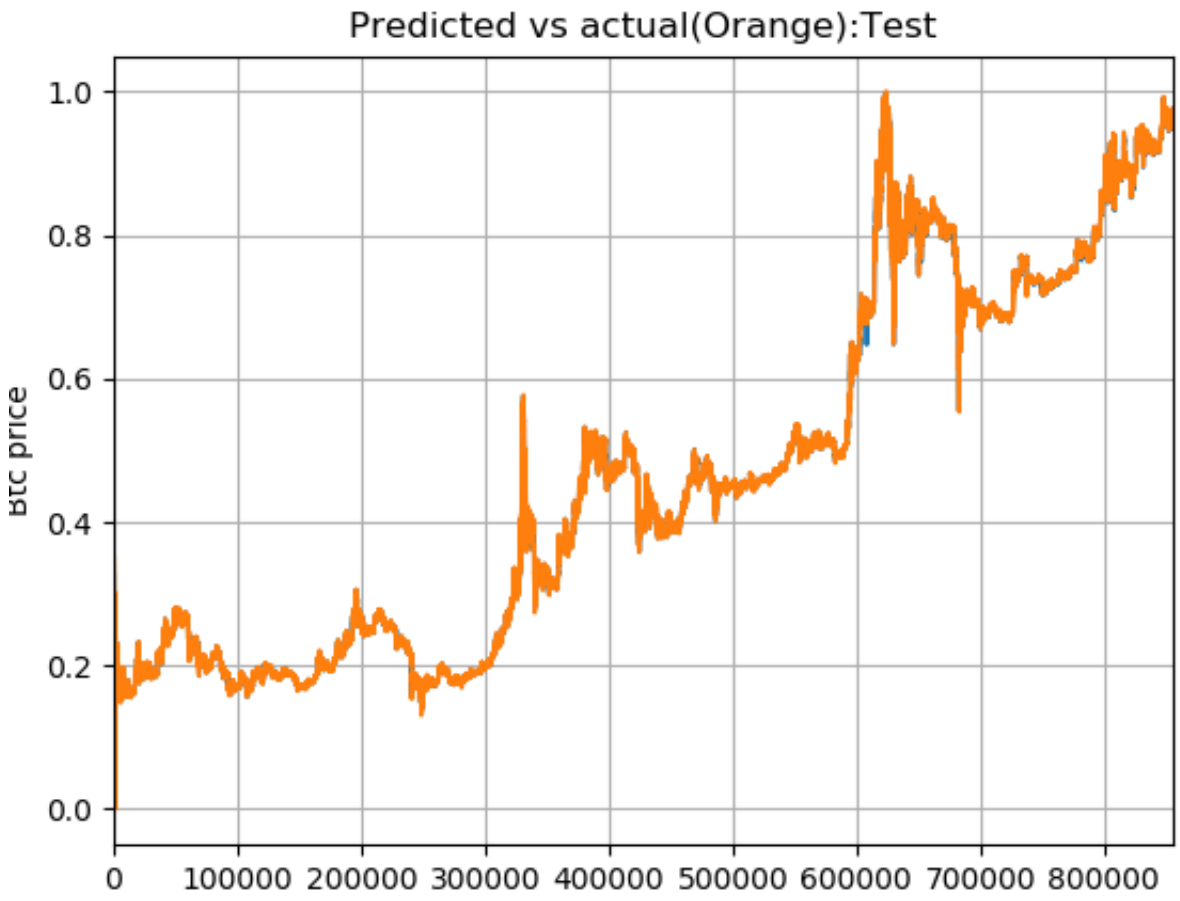

Figure 4.1: Graph of Predictions(Blue) vs Actual (Orange), Note: Data is in range 0-1 as it was not scaled back to actual values

during the simulation which provides us a way to visualize the data during the simulation to give an insight into the performance of each actor, this is detailed further in the next Chapter

4.2 Average Difference

The average difference result for this model was 0.18%, This difference is very small as can be seen with the graph of results in Figure 4.1 where it is not possible to see the predictions line most of the time as it is so close to the actual line. This is definitely an impressive result as it shows the network is able to learn to closely predict the price of Bitcoin. There is a caveat to this though, The model knows the price from 1 minute prior when making a prediction, meaning it cant go too far wrong when it tries to predict for this minute as there generally will not have been a large price change in just one minute. Nevertheless this is a good result.

4.3 Prediction Accuracy

The accuracy of the entire model was 45%, Although when parameters were put in place that only counted predictions when a larger price shift was predicted by the model, the accuracy of the model increased to 56.2%. These numbers show that the model is only accurate when it is more obvious that the price change will be larger. This is not unlike a real trader who would only act when they were confident they would be correct.. 56.2% is a result that would still lead to a profit but again this comes with a caveat. When filtering out the smaller price change predictions and only counting the larger ones, You are eliminating a lot of potential profit by playing it safe. The random trader achieved an accuracy of 48.6%, this is an average of 10 simulations as it will be slightly different each time due to the random nature.

4.4 Profit/Loss

This metric is probably the most important as it shows if the model is viable in earning money. We ran this simulation with no fees for buying/selling. The P/L when counting all the predictions was -23%, But again this P/L increased to 268% when filtering out the smaller predictions and only acting on the larger movements. The latter is clearly a better way of operating than the former in regards to this model. This result is made even more impressive by the fact that a Buy and Hold strategy for the same time period would have resulted in a -76% Return. The random trader fared even worse returning -87%, while the inverse of our algorithm returned -77%. The random trader figured was achieved by running the simulation 10 times and taking an average of its result, as its going to be different every time.

4.5 Results Table

Model	Prediction Accuracy	Avg Diff	P/L
Our Model	56.2%	0.18%	268%
Random Trader	48.6%	N/A	-87%
Buy and Hold	N/A	N/A	-76%

Chapter 5

Visual Representation of Model

To make it easier to visualize the performance of the model, We created a web interface that displays data relating to the performance of the model.

5.1 Web Interface Back-End

The web interface back-end was written in Python, using the Flask framework, this framework provides a extremely easy way of coding up a back-end in just a few lines of code. The Flask code runs on a separate thread to the operation of the neural network model. This back-end has two different functions, the first and simplest is simply serving the appropriate HTML file for each page. The second task is to dynamically render graphs of the models performance to send to the front end. The back-end re-renders the graph every time the page is refreshed.

5.2 Web Interface Front-End

The front-end for the web interface was designed with HTML, CSS and Javascript. There are two different pages in the front-end. The first page, which is the main page, provides several key data points on the performance of the model, including its P/L compared to the control models, and also a graph of each of the control models wealth. This page is updated when the page is refreshed to provide up-to-date information. This page hopefully gives a much easier to understand interface than simply looking an the console output of the simulation, which can just look like a mess of numbers. Figure 5.1 and 5.2 show this page

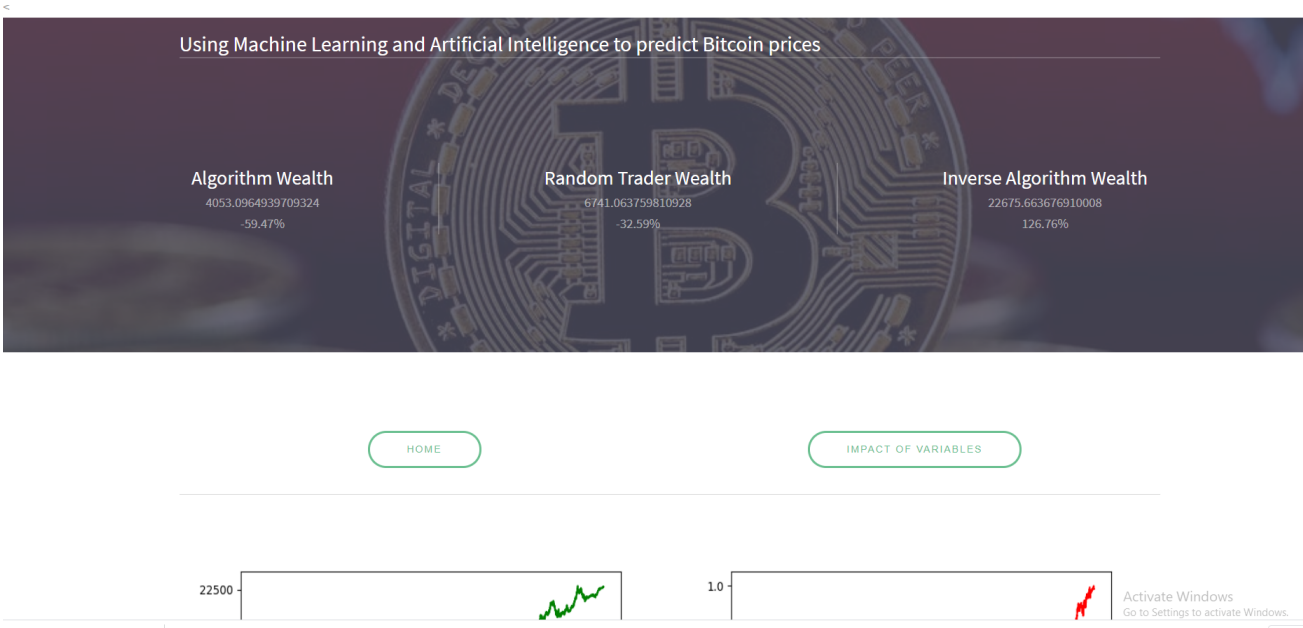

Figure 5.1: Web Frontend (Values are not indicative of results)

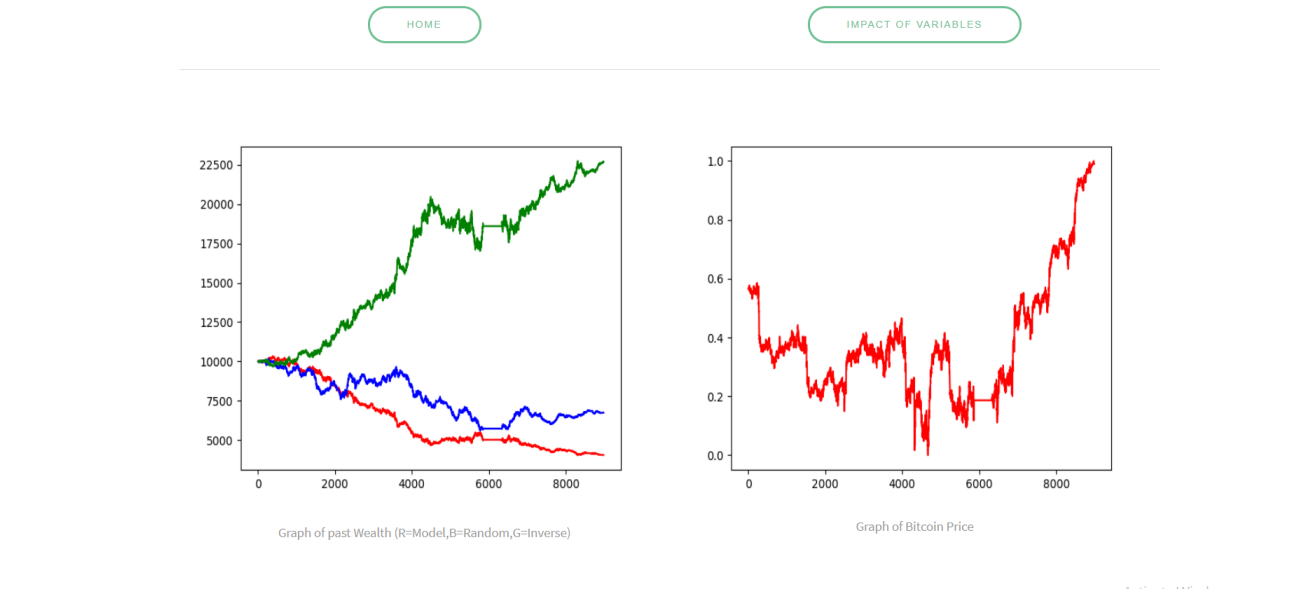

Figure 5.2: Web Frontend (Values are not indicative of results)

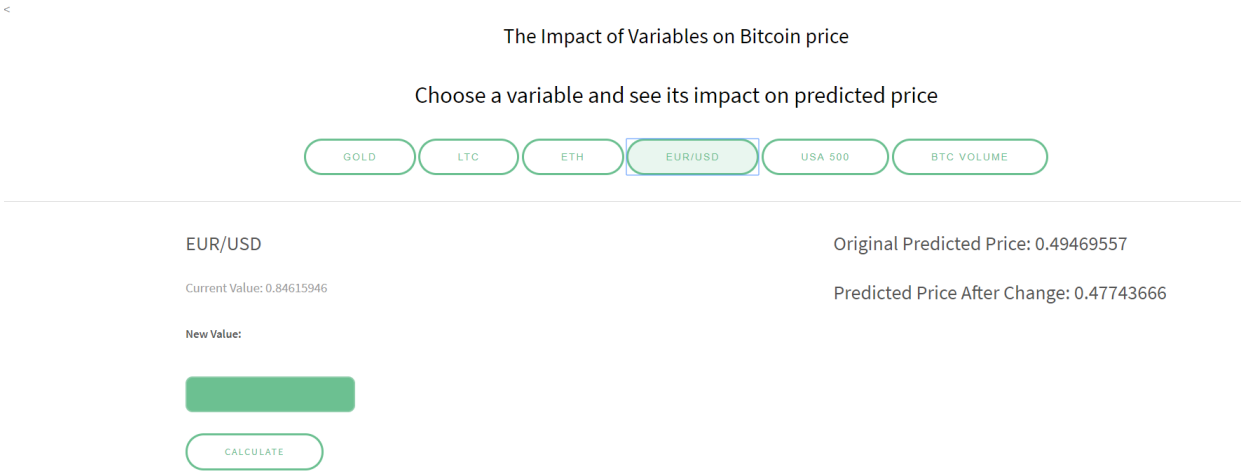

Figure 5.3: Impact of Variables page)

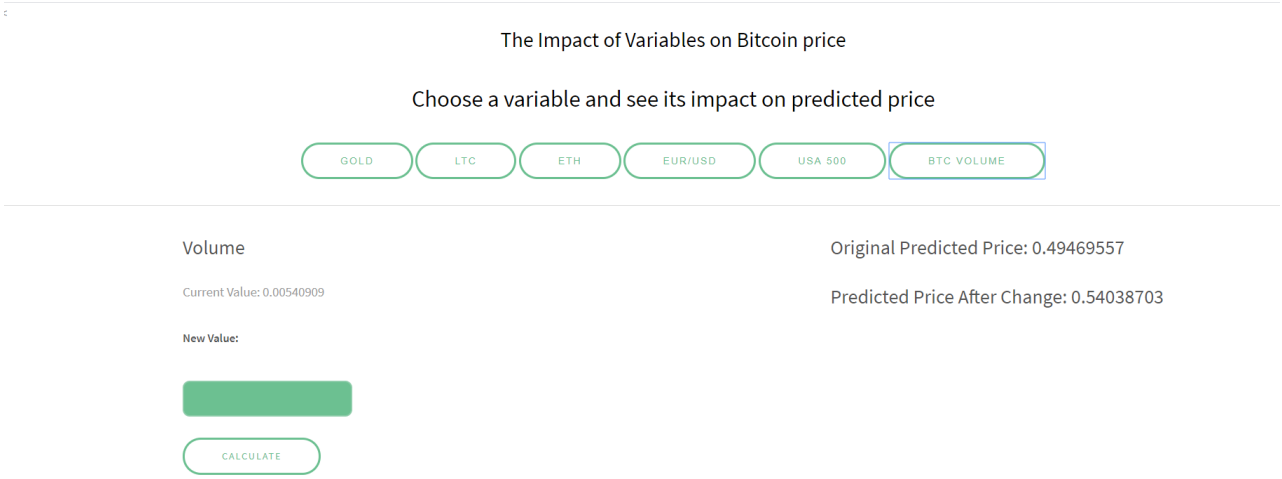

Figure 5.4: Impact of Variables page

The second page on the front-end can be used to manually tweak certain input variables in the neural network model to gain an insight into how much weight the model gives to each of those inputs. This can be used to help understand what way the model is thinking when it gives a prediction.

Chapter 6

Conclusions and Future Directions

6.1 Summary

To summarise, The work we have completed so far includes researching data and neural networks. We have also implemented a model based on the data we found and showed that it can be used to lucratively and accurately (50+%) predict Bitcoin prices. To further consolidate these findings, we would have liked to have more data to work with. Nevertheless, we believe this work is the first to openly prove it to be possible to predict short term price changes on a one minute time frame using a neural network and historical data.

6.2 Future Work

Going into the future,we hope that collecting more data , and also having more time available to us to retrain our model can increase the accuracy that we can achieve with the model. There are several other factors we could investigate for correlation if we had more time, these include full sentiment analysis on platforms like twitter and also Google search data, these would need to be web scraped as they are not currently available on a one minute time-frame. We could also include technical indicators such as RSI (Relative Strength Index) or Volatility as input data as Fazeli (2019) found that including these indicators can reduce MSE of the model. We also hope to implement an interface between our model and platforms that perform paper trading services. This will allow us to test our model using real, live data while not having to risk real money. This would be needed before

we could confirm whether the model can be profitable in real world scenarios.

Appendix A

Table of Factors

The factors marked in bold are deemed to be correlated. ** Factor was correlated but not enough data could be acquired to use in the model

Factor	Pearson's R	P Value
EUR/USD	**-0.318**	**<.001**
Btc Volume (USD)	**0.348**	**<.001**
No.Trades (Btc)	**0.331**	**<.001**
Gold Price	**0.692**	**<.001**
LTC/USD	**0.820**	**<.001**
ETH/USD	**0.879**	**<.001**
USD Index	**0.261**	**<.001**
Tweet Volume**	**0.638**	**<.001**
USA 500 Index	**0.242**	**<.001**
Tweet Sentiment	-0.136	<.001
Google Searches	-0.142	<.001
CPI	0.156	<.001
EFFR	-0.032	<.001
Chinese 500 Index	0.126	<.001
Us Inflation Rate	-0.015	<.001
Copper Price	-0.003	<.001
ETH Volume	0.089	<.001
TRX	0.124	<.001
TRX Volume	0.076	<.001
No. Trades LTC	0.109	<.001
No. Trades ETH	-0.098	<.001
LTC Volume	0.148	<.001

References

AdventuresInMachineLearning (2018), 'The vanishing gradient problem and relus – a tensorflow investigation adventuresinmacinelearning.com', Available: https://adventuresinmachinelearning.com/vanishing-gradient-problem-tensorflow/. [Accessed 12 Feb 2020]. 18

Badiola, J. (2019), 'Bitcoin 17.7 million tweets and price', Available: https://www.kaggle.com/jaimebadiola/bitcoin-tweets-and-price. [Accessed 21 Feb 2020]. 4

Chollet, F. (2015), 'keras', Available: https://keras.io//. [Accessed 14 Nov 2019]. 5

cjhutto (2020), 'Vader sentiment analysis.', Available: https://github.com/cjhutto/vaderSentiment. [Accessed 27 Apr 2020]. 4

Cohen (2013), 'Global bitcoin computing power now 256 times faster than top 500 supercomputers, combined!, Forbes', Available:https://www.forbes.com/sites/reuvencohen/2013/11/28/global-bitcoin-computing-power-now-256-times-faster-than-top-500-supercomputers-combined/#eb08bdc6e5e4. [Accessed: 28 Sep 2019]. 1

Dutta, A., Kumar, S. and Basu, M. (2019), 'A gated recurrent unit approach to bitcoin price prediction', Available: https://arxiv.org/ftp/arxiv/papers/1912/1912.11166.pdf. [Accessed 02 Apr 2020]. 3

faqs.org (2002), 'comp.ai.neural-nets faq, part 2 of 7: Learning', Available: http://www.faqs.org/faqs/ai-faq/neural-nets/part2/. [Accessed 09 November 2019]. 14

Fazeli, A. (2019), 'Using deep learning for predicting stock trends', Available: http://dr.library.brocku.ca/bitstream/handle/10464/14506/Brock_Fazeli_Arvand_2019.pdf?sequence=1&isAllowed=y. [Accessed 27 Apr 2020]. 3, 27

GeneralMills (2019), 'Pseudo api for google trends', Available: https://github.com/GeneralMills/pytrends. [Accessed 23 Dec 2019]. 4

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2012), 'Improving neural networks by preventing co-adaptation of feature detectors, CoRR', Available: http://arxiv.org/abs/1207.0580. [Accessed 10 Nov 2019]. 8

Hochreiter, S. and Schmidhuber, J. (1997), Lstm can solve hard long time lag problems, *in* 'Advances in neural information processing systems', pp. 473–479. 2

Kingma, D. P. and Ba, J. (2014), 'Adam: A method for stochastic optimization', Available: https://arxiv.org/abs/1412.6980. [Accessed 23 Dec 2019]. 11

Knight, W. (2018), 'A small team of student ai coders beats google's machine-learning code, technologyreview.com', Available: https://www.technologyreview.com/s/611858/small-team-of-ai-coders-beats-googles-code/. [Accessed 12 Nov 2019]. 9

Le Cun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E. and Hubbard, W. (1989), 'Handwritten digit recognition: applications of neural network chips and automatic learning', *IEEE Communications Magazine* **27**(11), 41–46. doi: 10.1109/35.41400. 6

Mangla, N. (2019), 'Bitcoin price prediction using machine learning', Available: https://www.researchgate.net/publication/333162007_Bitcoin_Price_Prediction_Using_Machine_Learning. [Accessed 10 Apr 2020]. 3

McNally, S., Roche, J. and Caton, S. (2018), 'Predicting the price of bitcoin using machine learning', Availible: http://trap.ncirl.ie/2496/1/seanmcnally.pdf. Accessed: 29/04/2020. 3

mesa.readthedocs.io (2016), 'Mesa: Agent-based modeling in python 3+', Available: https://mesa.readthedocs.io/en/master//. [Accessed 04 April 2020]. 19

Nakamoto, S. (2019), 'Bitcoin: A peer-to-peer electronic cash system', Available: https://git.dhimmel.com/bitcoin-whitepaper/. [Accessed 10 Nov 2019]. 1

Paszke, A., Gross, S., Chintala, S. and Chanan, G. (2016), 'Pytorch', Available: https://pytorch.org/. [Accessed 15 Nov 2019]. 5

Paul, S. (2018), 'Introduction to cyclical learning rates, datacamp', Available: https://www.datacamp.com/community/tutorials/cyclical-learning-neural-nets. [Accessed 16 Nov 2019]. iii, 9

Pham, V., Kermorvant, C. and Louradour, J. (2013), 'Dropout improves recurrent neural networks for handwriting recognition, CoRR', Available: http://arxiv.org/abs/1312.4569. [Accessed 07 Nov 2019]. 8

Shen, K. (2018), 'Effect of batch size on training dynamics', Available: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e. [Accessed 02 Dec 2019]. iii, 7, 8

Smith, L. N. (2015), 'Cyclical learning rates for training neural networks, CoRR', Available: http://arxiv.org/abs/1506.01186. [Accessed 11 Nov 2019]. 9

Smolyakov, V. (2018), 'Neural network optimization algorithms, towardsdatascience', Available: https://towardsdatascience.com/neural-network-optimization-algorithms-1a44c282f61d. [Accessed 16 Nov 2019]. iii, 10

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014), 'Dropout: A simple way to prevent neural networks from overfitting', Available: http://jmlr.org/papers/v15/srivastava14a.html. [Accessed 02 Nov 2019]. 8

Stathakis, D. (2009), 'How many hidden layers and nodes?', *International Journal of Remote Sensing* **30**, 2133–2147. 6

Struga, K. and Qirici, O. (2018), 'Bitcoin price prediction with neural networks,', Available: http://ceur-ws.org/Vol-2280/paper-06.pdf. [Accessed 02 Apr 2020]. 3

Wrosinski (2017), 'Deep learning frameworks speed comparison, Github.io', Available: https://wrosinski.github.io/deep-learning-frameworks/. [Accessed 16 Nov 2019]. iii, 5, 6