University of Waterloo
Faculty of Engineering

Catastrophes in Volatile Financial Markets – Predicting Cryptocurrencies

Prepared by

Shashank Sabhlok
20505648
ssabhlok@edu.uwaterloo.ca
4B Electrical Engineering

Shruti Appiah
20502631
sappiah@edu.uwaterloo.ca
4B Systems Design Engineering

Shrey Khosla
20535696
s2khosla@edu.uwaterloo.ca
4B Software Engineering

Xiaozhou Yu
20515155
x2yu@edu.uwaterloo.ca
4B Software Engineering

Table of Contents

1. Abstract

This project's aim is to predict the behavior of cryptocurrencies which have very volatile markets. Volatile markets, such as those of cryptocurrency, experience major price fluctuations driven by a handful of power-traders who hold a majority of the total crypto-asset. This makes it difficult for quantitative trading algorithms to predict cryptocurrencies, since they are primarily based on Gaussian models in which the probability of catastrophic events is very low. Two methods involving neural networks are investigated – Feedforward and LSTM to identify which method gives the most accurate cryptocurrency price prediction. Both neural networks were trained and tested on historical price data from Bitcoin and Ethereum blockchain explorers. Search history data from Google Trends was used to determine a correlation between a cryptocurrency popularity and its price. The LSTM neural network is a better forecaster of future cryptocurrency price. Our project is meant to be an additional source of input for traders to help them with making more informative decisions.

2. Introduction & Motivation to use the computational approach

Cryptocurrencies experience large and regular price fluctuations due to power traders that possess a large amount of the asset. Since the price of the cryptocurrency is based on their demand in the market, they are able to shift the price by placing large buy or sell orders. A computational approach could be helpful in making comparative predictions about its future price. In this project, we train and test two different neural networks and subsequently predict future cryptocurrency prices. Two approaches are used - a regular Feedforward Neural Network, and a Feedback or Recurrent Neural Network (RNN). We use a specific type of RNN which is composed of long short-term memory (LSTM). The above has been done for two cryptocurrencies - Bitcoin and Ethereum, the two most popular cryptocurrencies in the market today.

3. Theoretical Background

The first neural network used in this experiment is a feedforward neural network, a type of multilayer perceptron. The feedforward network used was obtained from the Tensorflow API. In a feedforward architecture, nodes are hierarchically chained starting with the input layer and concluding with the output layer. There are one or many hidden layers between the input and output layer that perform the computations in the network

The goal of a feedforward network is to approximate some function f. For example, for a classifier, the function $y = f(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x;\theta)$ and iteratively learns the value of the parameter θ that results in the best function approximation.

The second approach for cryptocurrency price prediction was implemented through a Recurrent Neural Network, specifically one with LSTM units (Long Short-Term Memory). Keras was used as the functional framework, with Tensor flow used behind the scenes to ensure that both approaches use the same framework. RNNs are widely used for sequential and time series predictions. The reason behind choosing RNN is due to their ability to make inferences from past data trends. Take an example of sequential data, such as the price of a cryptocurrency. A simple artificial neural network is able to learn to predict the prices based on a number of factors: the volume of the stock, the opening value, historical price trends, etc. While the price of the stock depends on these factors, it is also largely dependent on the stock values in the previous days leading up to that day. The price of the currency on the previous day is a major deciding factor for day-to-day predictions. In conventional feed-forward neural networks, all test cases are considered to be independent. That is, when fitting the model for a particular day, there is no consideration for the stock prices on the previous days. However, this dependency on time is achieved via Recurrent Neural Networks.

RNNs are problematic due to their vanishing gradient. All neural network nodes are associated with a weight In conventional feed-forward neural network, the process of

updating the weight that is applied on a particular layer, is a multiple of the learning rate, the error term from the previous layer and the input to that layer. Thus, the error term for a particular layer is a cumulative product of the errors from all previous layers. Thus, the weights of hidden layer nodes closer to the end of the network experience a lot more change than the ones close to the beginning of the network, near the input layer. The vanishing gradient problem becomes more prevalent and obvious as the network grows in size, thus limiting the researcher's ability to increase the depth of the network by increasing the number of hidden layers. Activation functions for such neural networks also need to be selected with care, since some of them are more immune to the vanishing gradient problem than others. ReLU (Rectifier Linear Unit) functions produce more stable networks than sigmoid functions since their derivative is equal to 1 at all points above 0, whereas the derivative for the sigmoid function is <0.25 everywhere. As a result, the gradient almost vanishes as we move towards the starting layers, and thus it becomes difficult to train these layers.

In time-series modelling, the depth of the network (i.e. the number of hidden layers) is an important parameter than requires change according to the application. Deep RNNs are bad at handling long-term dependencies due to the vanishing gradient problem. This is because as the number of time steps increase, the depth of the neural network increases as well, and this decreases the likelihood of disappearing gradients.

An RNN remembers information for only a small duration of time. If the information is required after a small period of time, it may be reproducible, but oftentimes once a lot of information is fed in, the data we are seeking gets lost within the network. This issue can be resolved by applying a slightly tweaked and improved version of the RNN – Long Short-Term Memory networks (LSTM). LSTM neural networks contain LSTM units (or memory cells) that store state information over time, thus being able to maintain long-term information. This also helps it avoid the vanishing gradient problem [1].

Convolutional Neural Networks (CNN) were also considered. However, its application was mostly used in image and video recognition and thus it was not studied for this time-series prediction application.

Using sentimental analysis for cryptocurrencies proved to be challenging since it is a highly volatile market that is dominated by bots trading on a minute-by-minute (if not second-by-second) basis. This encouraged us to take a different approach in getting sentimental analysis since cryptocurrencies are still a very fresh market and any major increase or decrease in the market is usually because of power-traders, also known as "whales", who hold the majority of crypto-assets. Instead of performing sentiment analysis, a correlation study was performed. Search data from Google Trends was used as an indicator of popularity or interest in a particular cryptocurrency. This was used to analyze if the search frequency of cryptocurrency words associated with a particular currency (such as 'ethereum', 'ether', or 'eth' associate with the Ethereum market) has a correlation with the price of the cryptocurrency.

Ethereum is an open-source platform that extends the functionality of a traditional blockchain to beyond just being a cryptocurrency. Not only is it used as a cryptocurrency, it is also used in various other applications. Some of the use cases for Ethereum are in supply chain-management, digital identity, financial sector, decentralized renewable energy trading etc. All these use cases are realized through software applications built on the Ethereum platform and utilize the Ether cryptocurrency for all transactions made on the app. Thus, this extended application for Ethereum introduces multiple additional factors that influence the price of the Ether cryptocurrency. Due to this, the Ethereum ecosystem is far more complex, thus further motivating the need for a computational approach while forecasting its future price.

4. Solution

4.1. Dataset

The dataset used to train the neural networks is the Bitcoin and Ethereum historical price and trading data. These are the top two cryptocurrencies currently in terms of market-share and have free and open historical trading data and pricing charts available online. Pricing data for both cryptocurrencies was availed from https://coinmarketcap.com. This website allowed us to get the historical daily prices and market cap of Ethereum and Bitcoin. The initial input data of Ethereum ranged from August 7th 2015 to March 2nd 2018.

All data was exported from these websites as CSVs. Since this application only requires the price and market cap data, irrelevant columns such as trading volume, circulating supply, orders placed, and daily highs and lows were dropped. One key design decision made was to base all predictions on the daily closing price of the cryptocurrency.

Pandas was used to ingest data. The CSV data was converted into Python data frame objects with rows and columns, similar to table in languages such as R. The data required to be normalized so that it maintains consistency and the same scale. To achieve this, scikit learn's *min_max_scaler* was used to transform input data into values ranging from 0 to 1. The input is the closing price of the cryptocurrency and the output is the closing price of the day after. The input is an array of daily closing prices. For input i (ith index of the array) the output would be the price of i+1. Modelling the data this way allows us to obtain the price of the cryptocurrency one day in advance.

4.2. Tools and Algorithms

Python was used due to its popularity in the machine learning and scientific communities. It has several available libraries such as scikit-learn, matplotlib, numpy, tensorflow, and keras which are relevant to this application. The neural networks were implemented using the Python wrapper for Tensorflow, which is an open-source machine learning library.

We used one hidden layer and varied the number of neurons in the first iteration. The activation function used in the neurons in the hidden layers was the Rectified Linear Unit (ReLU) function. This is because ReLU aids in stability of the neural network due to its immunity to the vanishing gradient problem that is common with sigmoid and tanh function. Additionally, it has been shown that deep neural networks can be trained efficiently using ReLU even without pre-training. An optimizer is required to iteratively compute the weights or biases of the neurons. It calculates the gradient and adjusts the weights of the neurons to reflect the learning performed in that stage, thus optimizing the network's cost function. After trial-and-error, we concluded that the Adam optimizer produced the best results.

The Adam optimizer (stands for Adaptive Moment Estimation) is a variant of the traditional stochastic gradient optimizer. Instead of maintaining the same learning rate for all neuron weigh updates, it re-calculates the learning rate in each iteration based on the first-order and second-order moments of the gradients. In statistics, a moment can be defined as a measure of the shape of a set of points used to characterize probability distributions. A common moment we are all familiar with is the mean, or the expected value, which is the first-order moment. Likewise, the second-order moment is the variance of a distribution.

The Adam optimizer is a combination of two other gradient calculators - AdaGrad (Adaptive Gradient) and RMSProp (Root-Mean-Square propagation). Contrary to RMSProp which is only based on the first moments - i.e. the mean - the Adam optimizer recalculates weights based on the mean and variance of the gradients. In each step, it calculates the exponential moving average of the gradient and the squared gradient. Since it also uses some techniques from AdaGrad, it is able to handle sparse gradients as well. This is done by basing the second moment calculation on multiple past gradients [2].

The cost function of a neural network is a measure of how good a network is with respect to the given training data and the expected output. A cost function $\boldsymbol{C}$ of the neural network is an average of the cost functions $\boldsymbol{C_x}$ of the individual training examples. The

cost function used is the mean-squared error function, which is commonly used in regression problems. Mean-squared-error calculates the average squared deviation between the network's actual outputs and the correct expected outputs. The deviation is minimized as the network learns to make more accurate predictions.

The starting point of the neural network is determined based on weight initializers. The weights were initialized with random uniformly distributed weights. The testing and training data were split using scikit learn's *train_test_split* function. An 80/20 split was used for training/testing respectively. The batch size - which is a randomly selected sample of data points that propagate through the network for a given iteration - was chosen to be 10. These batches were run through 5 epochs. Once the training and testing were completed, the data was inverse transformed to obtain the price and then this result was plotted using *matlibplotlib*.

To train and test the data on different days, *sys.argv* was used to specify the number of days in advance. The argument is then taken into consideration when creating the input and output data of the training and testing set.

At this point, we have a working solution with feedforward network. However, further optimization could be performed. To improve the network, the number of hidden layers, number of neurons in each layer, the batch size, and finally the number of epochs were each varied. One important optimization that was used in this method was mini-batching. The key advantage of using mini-batch as opposed to the full dataset goes back to the fundamental idea of stochastic gradient descent [3]. In batch gradient descent, the gradient is computed over the entire dataset, averaging over the vast amount of information comprising of all daily cryptocurrency prices from the start of the dataset. This is very memory intensive. Since batch gradient descent is a non-convex optimization technique, its trajectory can lead to the point being trapped in a saddle point (the min/max point in a non-convex function) over the optimization iterations.

In pure Stochastic Gradient Descent (SGD), the weights are calculated based on the gradient of a single, random instance. his calculation is performed for each training example in the dataset. Since it's based on one random data point, convergence to the correct gradient result becomes hard as the computations are very noisy and frequently diverge in directions far from the batch gradient result. However, the noisiness is helpful in non-convex optimization since it helps escaping from saddle points and local minima (Theorem 6 in [4]).

With the minibatch stochastic gradient descent (MSGD) methodology, a noise factor can be artificially introduced to each gradient update such that it escapes local minima and saddle points, while still being able to converge to the batch gradient fast [5][6].

There are two important issues that could occur when it comes to the number of hidden layers and neurons [7]. First, machine learning problems that require two hidden layers are rarely encountered. Additionally, neural networks with two hidden layers can represent functions with any kind of shape. There is currently no theoretical reason to use neural networks with any more than two hidden layers. In fact, for many practical problems, there is no reason to use any more than one hidden layer.

In the case of number of neurons per hidden layer, there are potential problems with overfitting and underfitting. Using too few neurons in the hidden layers will result in underfitting. Underfitting occurs when there are not enough neurons in the hidden layers to adequately detect the signals in a complicated data set.

Using too many neurons in the hidden layers can result in several problems. First, too many neurons in the hidden layers may result in overfitting. Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers. In a sense, the machine memorizes the answers to questions instead of building the intelligence to respond appropriately.

A second problem can occur even when the training data is sufficient. An inordinately large number of neurons in the hidden layers can increase the time it takes to train the network. The amount of training time can increase to the point that it is impossible to adequately train the neural network. Some key rules that can be kept in mind while determining the number of hidden neurons are:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

Additionally, this formula makes a good estimate for the optimum number of hidden neurons:

$$N_h = \frac{N_s}{(a * (N_i + N_o))}$$

The resulting optimized neural network produced a ~75% accurate result.

The second approach used for predicting future cryptocurrency prices was Recurrent Neural Networks (RNN). An RNN with LSTM units (also called memory cells) is able to store information from the past and use that to inform its prediction. For sequential data such as the price of stocks or cryptocurrencies, this is particularly beneficial.

The implementation of the LSTM network was done using Keras as it allowed to write and test our network easily. Similar to the feedforward network, the data was normalized to values ranging between 0 and 1 using scikit learn's *min_max_scaler*. To use LSTM with Keras, the Sequential model was required. The sequential model involves stacking individual neural network layers upon each other. At each layer, the the number of LSTM cells to use can be specified. The batch size and epoch were kept the same as the feedforward network since the two were being compared.

5. Analysis and Results

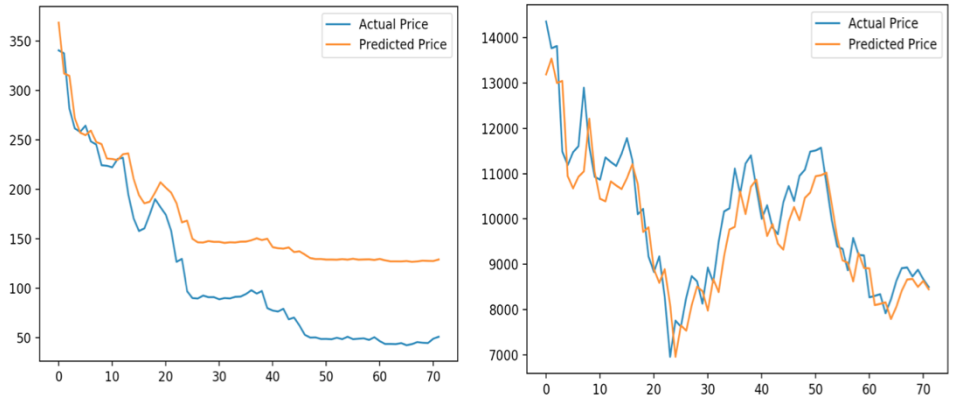


Figure 1: Prediction Bitcoin prices for 1 day. The graph on the left shows the results from the feedforward network and the graph on the right shows the results from the LSTM network

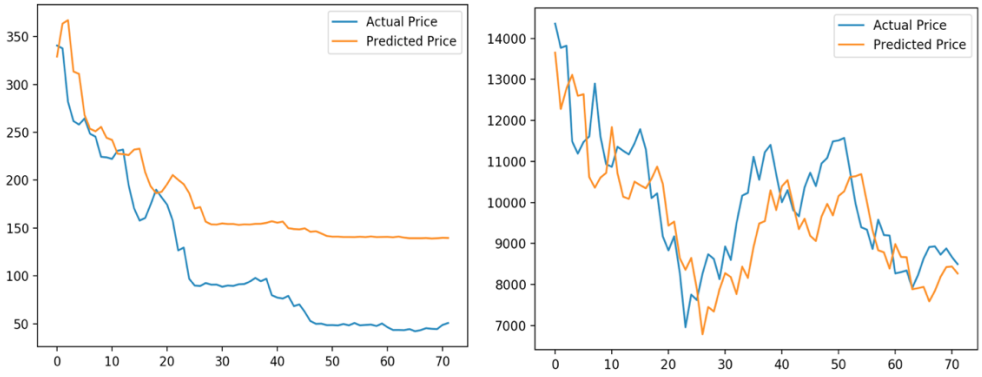


Figure 2 : Prediction Bitcoin prices for 3 days. The graph on the left shows the results from the feedforward network and the graph on the right shows the results from the LSTM network

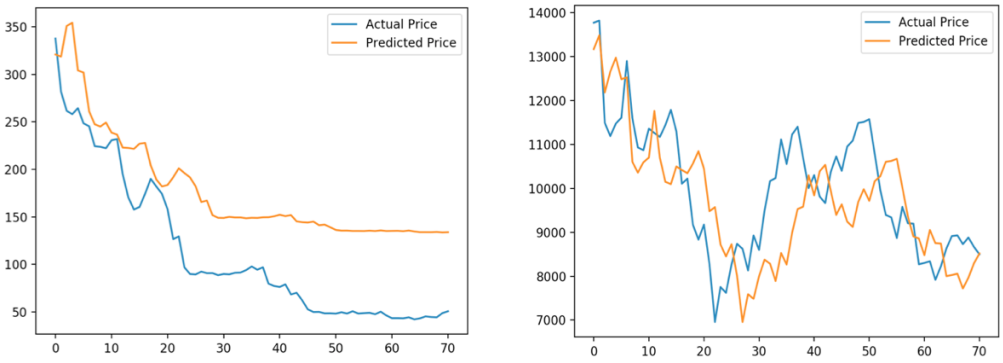


Figure 3 : Prediction Bitcoin prices for 5 days. The graph on the left shows the results from the feedforward network and the graph on the right shows the results from the LSTM network

From the graphs above it is evident that the LSTM neural network performs far better than the feedforward neural network. As described above, LSTM has the ability to update

weights on the network as it gathers new information. The LSTM network is able to remember information for a long time as it stores state information over arbitrary periods of time. Therefore, it is no surprise that the prediction graph fits the actual data much better with the LSTM neural network when predicting over a day, three days and five days.

Over a span of few days, in both types of neural networks an interesting trend is observed. As the prediction period gets longer, the prediction graphs fit the actual graph lesser. Simply comparing a 1-day prediction graph against a 5 day prediction graph makes the discrepancy fairly obvious.

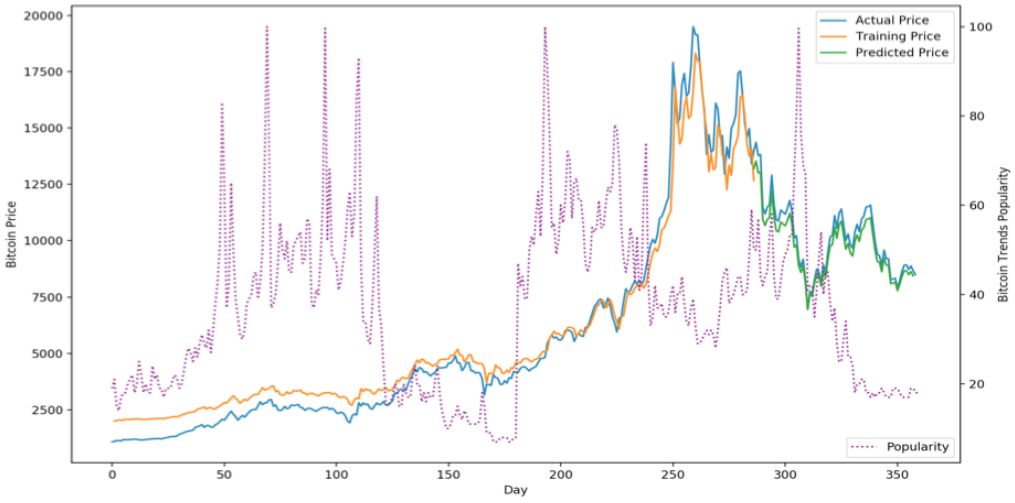


Figure 4 : Detailed result of Bitcoin Price Prediction graph for a 1 day. The Google Trends data (purple) is also plotted on this graph to show a correlation.

Figure 4 gives a more detailed view of the Bitcoin prediction for the single day. It can be seen the trained model (indicated in orange) fits the actual pricing graph of Bitcoin fairly well.

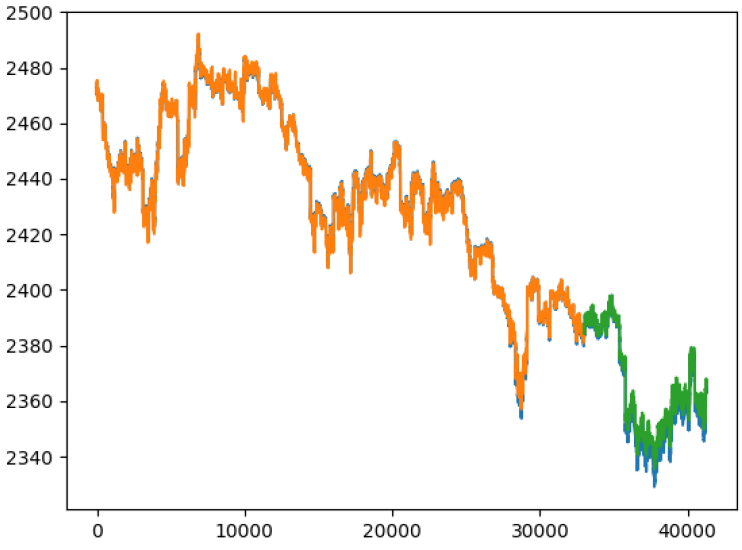
Furthermore, this is further evident through the predicted pricing line (green) as well. The purple line on the graph indicates search popularity data from Google Trends and throws light on some interesting correlations. A categorical example is the spike in the price of bitcoin between days 250-300, immediately after a spike in searches between days 175-245.

5.1. Analyzing Performance with different datasets

After building both the models, we found LSTM to be a better neural network solution for this problem. We used this LSTM solution to do some analysis with using different datasets to train the model and to even re-use the trained model to predict other datasets, to really be able to see the diverse solution capability of our neural network model.

5.1.1. S&P 500 Dataset

Initially, we implemented our solution with a dataset comprised of a popular stock market outside of cryptocurrency. S&P500 was chosen for this purpose. We were able to get minute-by-minute data with a much larger dataset than was possible for either of our cryptocurrencies. The graph below shows the capability of our LSTM network to be used on any sort of time-series based numerical model. It is almost impossible to see the blue line that represents the actual price, which is also shown through the details about how accurate the results are!



('Last Timestep Prediction Price: ', array([2366.4507], dtype=float32))
('Last Timestep Actual Price: ', array([2363.61], dtype=float32))
('Training Accuracy: ', 99.97568640226372)
('Testing Accuracy: ', 99.88736607331604)

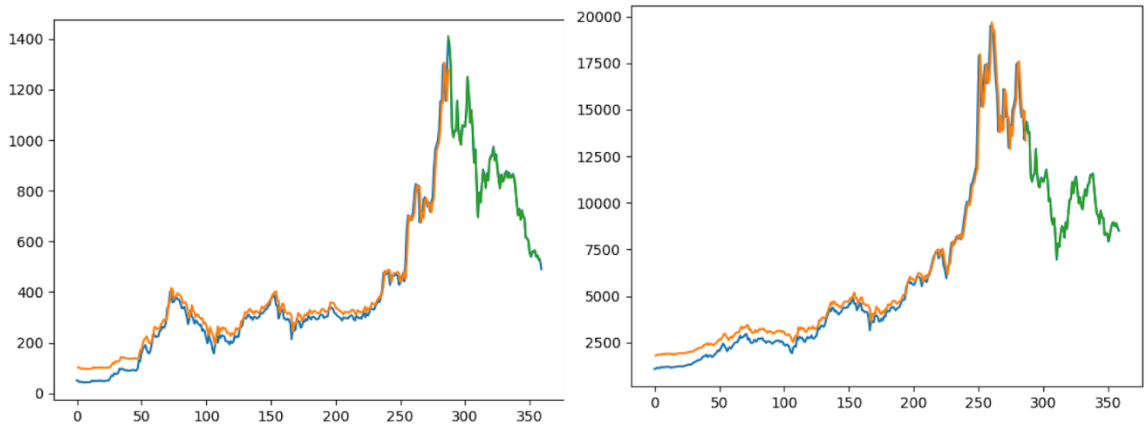


Figure 5 : Ethereum(Left) & Bitcoin (Right) prices predicted using a LSTM model trained with S&P 500 dataset

The S&P500 trained Model performs quite well on the testing dataset for both Ethereum and Bitcoin but seems to falter when it comes to their training datasets when taking average price accuracy into account. Its considerable accuracy for its own dataset shown in the first figure above doesn't seem to be seen when using the model for the cryptocurrency datasets.

S&P Model used to predict Ethereum Data above

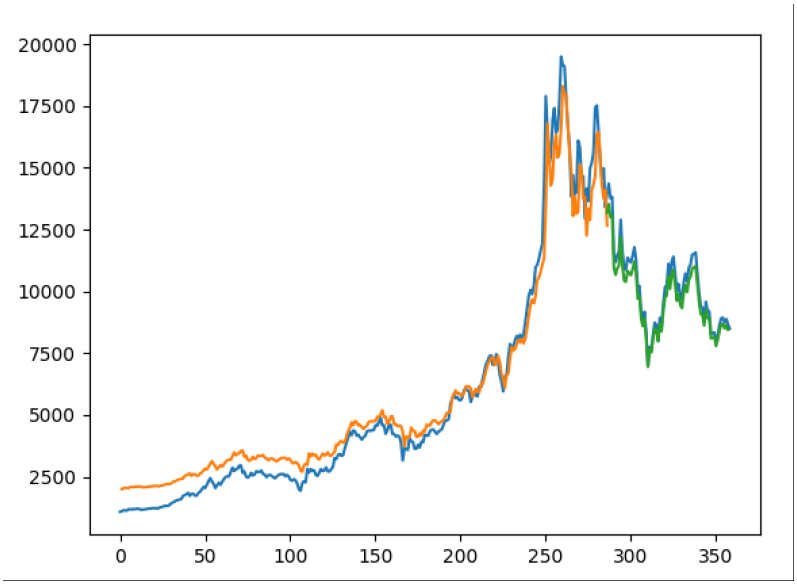
('Last Timestep Prediction Price: ', array([529.9234], dtype=float32))
('Last Timestep Actual Price: ', array([489.95004], dtype=float32))
('Training Accuracy: ', 78.84977377906982)
('Testing Accuracy: ', 94.9355009866445)

S&P Model used to predict Bitcoin Data above

('Last Timestep Prediction Price: ', array([8690.031], dtype=float32))
('Last Timestep Actual Price: ', array([8495.78], dtype=float32))
('Training Accuracy: ', 82.44838786024073)
('Testing Accuracy: ', 95.26516954796939)

5.1.2. Bitcoin Dataset

The model for the Bitcoin dataset, shown again here for comparison purposes with its accuracy information below:



('Last Timestep Prediction Price: ', array([8444.084], dtype=float32))
('Last Timestep Actual Price: ', array([8495.78], dtype=float32))
('Training Accuracy: ', 78.12852344145573)
('Testing Accuracy: ', 94.71744138475819)

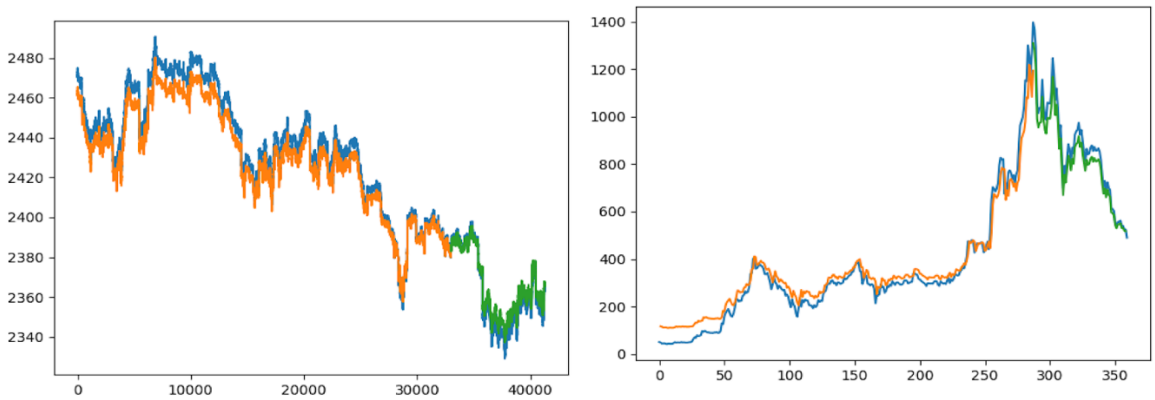


Figure 6: S&P500 (Left) & Ethereum(Right) prices predicted using a LSTM model trained with Bitcoin dataset

As can be seen the Bitcoin model was able to predict the S&P dataset really well and although had some trouble through the training dataset of the Ethereum Cryptocurrency, it was able to get a solid performance in the testing part.

Bitcoin Model used to predict S&P Data

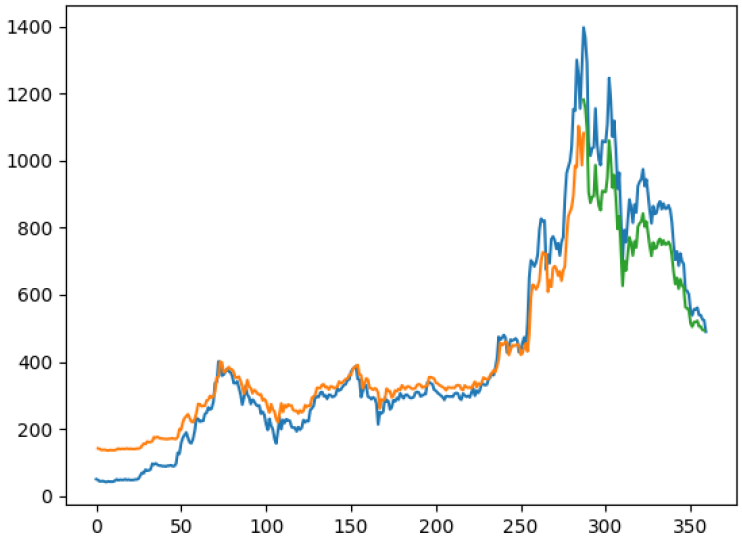
('Last Timestep Prediction Price: ', array([2366.4548], dtype=float32))
('Last Timestep Actual Price: ', array([2363.61], dtype=float32))
('Training Accuracy: ', 99.74844646423064)
('Testing Accuracy: ', 99.8632029283608)

Bitcoin Model used to predict Ethereum Dataset

('Last Timestep Prediction Price: ', array([517.5121], dtype=float32))
('Last Timestep Actual Price: ', array([489.95004], dtype=float32))
('Training Accuracy: ', 73.93549144351596)
('Testing Accuracy: ', 94.46891265181168)

5.1.3. Ethereum Dataset

The Bitcoin model, shown again here for comparison purposes with its accuracy information below:



('Last Timestep Prediction Price: ', array([493.90067], dtype=float32))
('Last Timestep Actual Price: ', array([489.95004], dtype=float32))
('Training Accuracy: ', 65.45871782273976)
('Testing Accuracy: ', 89.28075941760957)

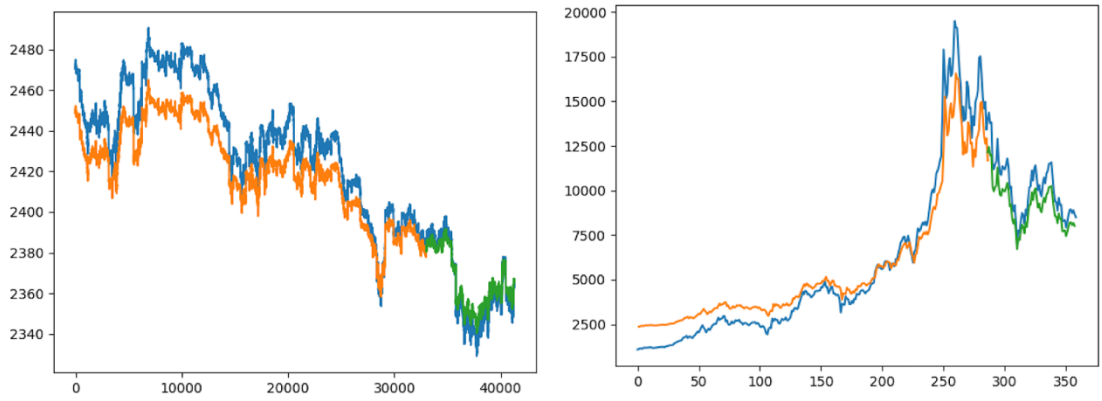


Figure 7: S&P500 (Left) & Bitcoin (Right) prices predicted using a LSTM model trained with Ethereum dataset

The Ethereum model seems to follow the same footsteps as the Bitcoin Model, in that it is performing really well on the S&P dataset but seems to falter when testing its neighbouring cryptocurrency, Bitcoin, but seems to catch back up in the testing dataset.

Ethereum Model used to predict S&P Dataset

('Last Timestep Prediction Price: ', array([2366.0544], dtype=float32))
('Last Timestep Actual Price: ', array([2363.61], dtype=float32))
('Training Accuracy: ', 99.4004183579738)
('Testing Accuracy: ', 99.8003428842307)

Ethereum Model used to predict Bitcoin Dataset

('Last Timestep Prediction Price: ', array([8008.1367], dtype=float32))
('Last Timestep Actual Price: ', array([8495.78], dtype=float32))
('Training Accuracy: ', 70.07860334624066)
('Testing Accuracy: ', 90.76132622754407)

5.1.4. Combined Dataset (training with all datasets)

A combined model was also created to show that the more the model is trained the better it becomes. An 80/20 split was created for each dataset again to train the model sequentially through each dataset and then show the combined models prediction for each dataset after training is completed. This model generated the best results for avg accuracy for both the training and testing datasets. The fact that it had muddled through and trained

with diverse price time-series through different input sources it was able to learn through the patterns better than any other model. Graphs and their correlating accuracy information is shown below :

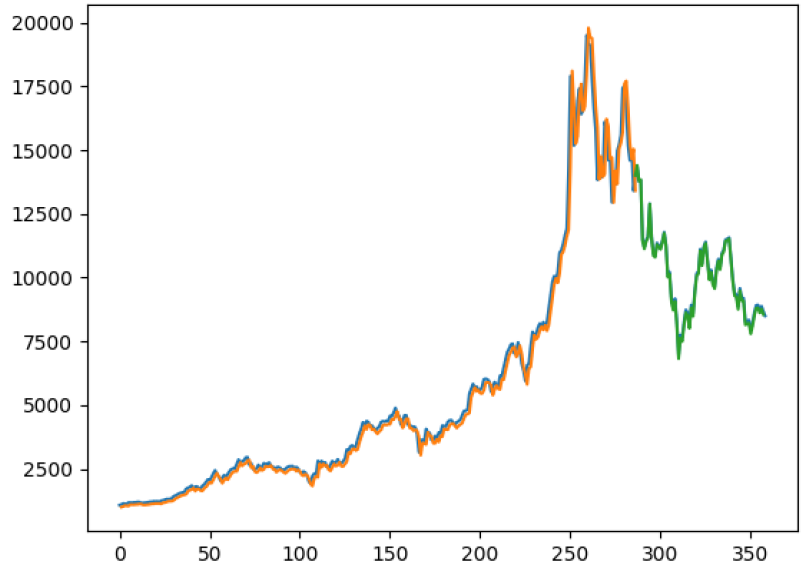


Figure 8: Combined Model used to predict Bitcoin

Model for the graph above:

('Last Timestep Prediction Price: ', array([8553.172], dtype=float32))
('Last Timestep Actual Price: ', array([8495.78], dtype=float32))
('Training Accuracy: ', 94.32112462292133)
('Testing Accuracy: ', 95.15469771593214)

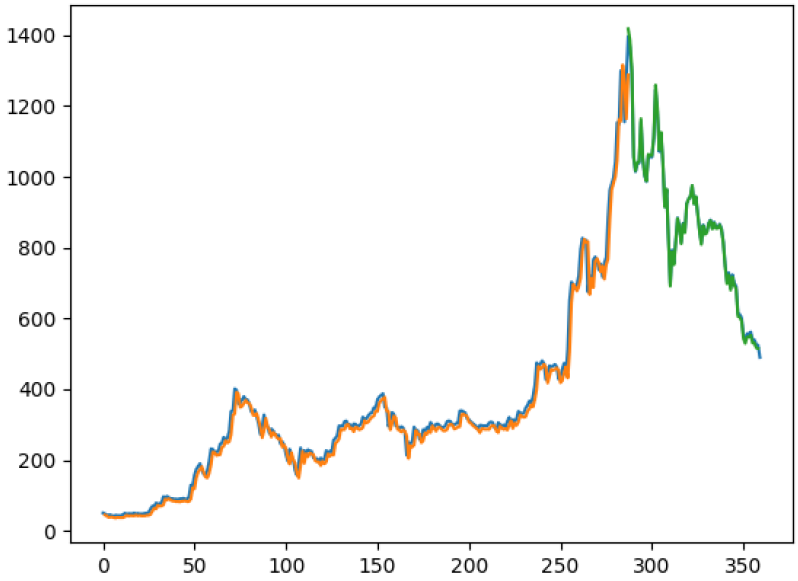


Figure 9: Combined Model used to predict Ethereum

Model for the graph above:

('Last Timestep Prediction Price: ', array([514.9161], dtype=float32))
('Last Timestep Actual Price: ', array([489.95004], dtype=float32))
('Training Accuracy: ', 93.22016324905942)
('Testing Accuracy: ', 94.95572755283015)

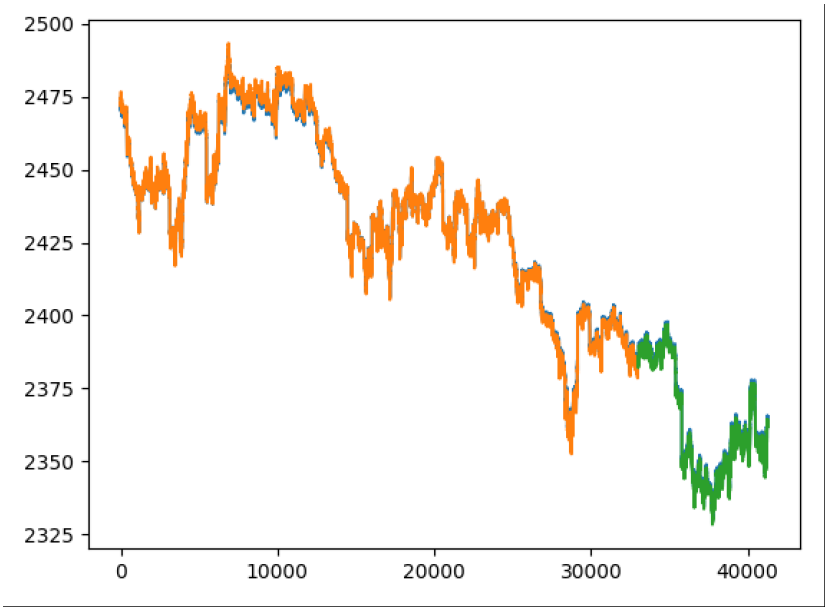


Figure 10 : Combined Model used to predict S&P500

Model for the graph above:
('Last Timestep Prediction Price: ', array([2362.9097], dtype=float32))
('Last Timestep Actual Price: ', array([2363.61], dtype=float32))
('Training Accuracy: ', 99.96455514192868)
('Testing Accuracy: ', 99.95301151968692)

6. Conclusion

In this experiment, both FeedForward and LSTM neural networks were implemented. Their accuracy in forecasting future cryptocurrency prices was quite substantial. LSTM avoids the problem of vanishing gradient. The LSTM neural network can remember prices from the past which allow the LSTM neural network to identify long-term dependencies and predict more accurate prices based on them. Additionally, it was observed that there is a positive correlation between the search frequency of a cryptocurrency - obtained from Google Trends data - and its price. Further research will determine the extent of this correlation.

References

[1] Srivastava, P. (2018). Essentials of Deep Learning: Introduction to Long Short-Term Memory. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/ [Accessed 4 Apr. 2018].

[2] D. P. Kingma and J. L. Ba (2015),” ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION” in Proc. ICLR, 2015 (pp. 1 -13)

[3] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT,2010 (pp. 177-186).

[4] Ge, R., Huang, F., Jin, C., & Yuan, Y. (2015). Escaping From Saddle Points-Online Stochastic Gradient for Tensor Decomposition. In COLT (pp. 797-842).

[5] Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1–14. Retrieved from http://arxiv.org/abs/1406.257

[6] Brownlee, J. (2018). A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size - Machine Learning Mastery. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/ [Accessed 4 Apr. 2018].

[7] Heaton, J. (2008). Introduction to Neural Networks for Java, Second Edition. St. Louis, Mo.: Heaton research.