Predicting the Price of Bitcoin using Deep Learning

David K. Hubbard
University of Oxford
david.hubbard@trinity.ox.ac.uk

September 19, 2018

Abstract

Bitcoin is arguably the most popular 'cryptocurrency'; a decentralized currency meaning it is not regulated by any bank. The popularity of Bitcoin increased massively at the end of 2017. My aim was to see if machine learning techniques could be used to predict the future price of Bitcoin. More specifically I would use deep learning; namely, Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) architectures to see if models with these structures could learn features of the data and predict its future behaviour.

1 Introduction

Forecasting time series data is a typical problem in data analysis. When it comes to predicting the future price of the S&P 500 or the FTSE 100 for example, the indices have a general increase in trend over time, coupled with seasonal oscillations. However, the price of Bitcoin has been incredibly erratic and seemingly unpredictable over the last two years or so. My aim is to see if machine learning models can closely predict the future behaviour of Bitcoin given only its past behaviour. I will be using the Python programming language and the Keras software to build, train and test deep learning models on the Bitcoin price data to see if it can make useful predictions.

2 Neural Networks

Preamble

In general, suppose we have n items of data: $x_1, x_2, ..., x_n$ and we want to use this data to make predictions on the m data points: $y_1, ..., y_m$. Naturally, we seek a map: $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ such that $\hat{\boldsymbol{y}} := \boldsymbol{f}(\boldsymbol{x})$ where $\boldsymbol{x} := (x_1, ..., x_n)^T$, $\boldsymbol{f} := (f_1, ..., f_m)$ and $\hat{\boldsymbol{y}} := (\hat{y}_1, ..., \hat{y}_m) := (f_1(\boldsymbol{x}), ..., f_m(\boldsymbol{x}))$, is 'close' to the true

value: $\boldsymbol{y} := (y_1, ..., y_m)^T$. We have a choice of definition of closeness by means of a 'loss function' which gives a measure of closeness of our prediction from the true value. Here we will choose Mean Squared Error (MSE) as our loss function which is given by: $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) := (||\boldsymbol{y} - \hat{\boldsymbol{y}}||_2)^2 := \sum_{k=1}^{m}(y_k - \hat{y}_k)^2$. We now have a way of quantifying how good our predictions are but the choice of function is still open. It is rarely the case that some sort of linear function will be a suitable choice for $\boldsymbol{f}$. This is where we turn to neural networks which allow us to form complex non-linear functions that are optimized to minimize loss.

Neural Network Structures

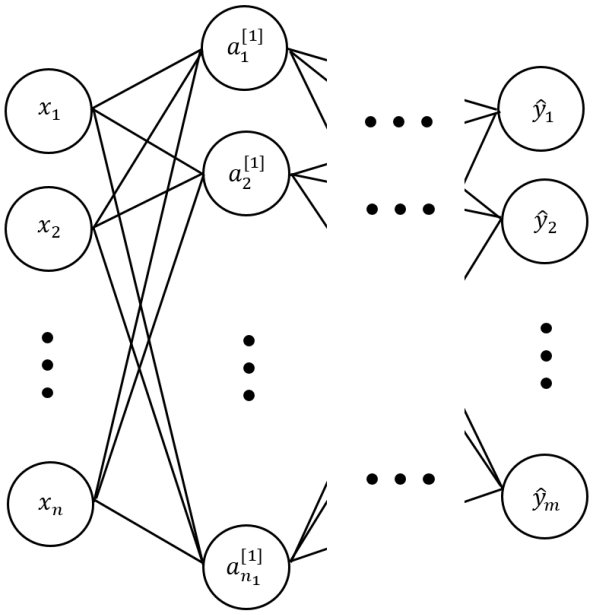

Figure 1: Structure of a Neural Network

The general Neural Network is comprised of an input layer, an output layer and a number of hidden layers. Each layer is comprised of a series of nodes. The input layer simply has each input value as a node, similarly the final prediction has each value as a node of the output layer. Each layer in the Neural Network is formed by taking a weighted sum of the values in the nodes of the previous layer, plus a bias, and feeding this into an activation function.

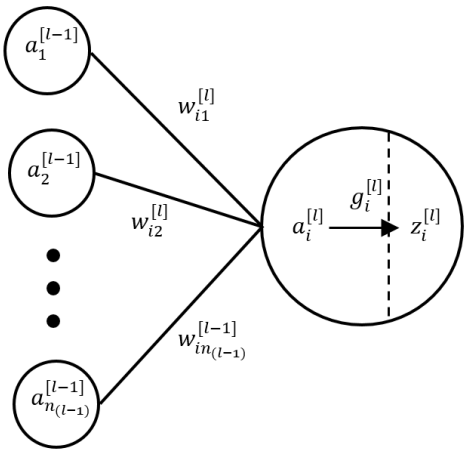

Figure 2: A single node in the lth layer

Suppose we have values $a_1^{[l-1]}, ..., a_{n_{(l-1)}}^{[l-1]}$ in the $(l-1)$th layer and values $a_1^{[l]}, ..., a_{n_l}^{[l]}$ in the lth layer. Consider a general value in lth layer: $a_i^{[l]}$. To obtain the value $a_i^{[l]}$, we first take a weighted sum of the values in the previous layer, plus a bias: $z_i^{[l]} = w_{i1}^{[l]} a_1^{[l-1]} + ... + w_{in_{(l-1)}}^{[l]} a_{n_{(l-1)}}^{[l-1]} + b_i^{[l]}$. We then apply an activation function: $g_i^{[l]}$, to obtain: $a_i^{[l]} = g_i^{[l]}(z_i^{[l]})$. More succinctly, we can use the following notation:

$$Z^{[l]} = (z_1^{[l]}, ..., z_{n_l}^{[l]})^T,\ a^{[l-1]} = (a_1^{[l-1]}, ..., a_{n_{(l-1)}}^{[l-1]})^T,\ b^{[l]} = (b_1^{[l]}, ..., b_{n_l}^{[l]})^T,$$
$$g^{[l]} = (g_1^{[l]}, ..., g_{n_l}^{[l]})^T \text{ and } W^{[l]} = (w_{ij}^{[l]})$$

Then, the inputs from a layer in the Neural Network are derived from the previous layer by the following formula:

$$a^{[l]} = g^{[l]}(Z^{[l]}) = g^{[l]}(W^{[l]} a^{[l-1]} + b^{[l]})$$

Now we have a massive number of complex functions which we can use to make predictions. This is true by virtue of the numerous possibilities for the weight matrices and bias vectors: $W^{[l]}$ and $b^{[l]}$ respectively. However, we need a way of finding the best weights and biases. This leads us to the process of 'back propagation'.

Back Propagation

At its core, back propagation is an optimization algorithm. We've chosen to quantify the performance of our model by how small the loss function is: $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$.

In back propagation we consider our loss function to be dependent on the weights and bias but our inputs and true values to be fixed. Back propagation is our method of minimizing the loss function by optimizing these parameters. We start by initializing our parameters, it is common to either initialize with zeros, or in our case to initialize with a random distribution; the default in Keras is a form of uniform distribution. This way, we're left with randomly chosen parameters that we can seek to optimize. We make a choice of differentiable functions: $g^{[l]}$ (that we will discuss later), thus our loss function is itself differentiable. There are many different optimizers used in machine learning, I have opted for the state-of-the-art Adam optimizer. It is based on gradient descent which is a simple algorithm for optimizing differentiable 'objective' functions.

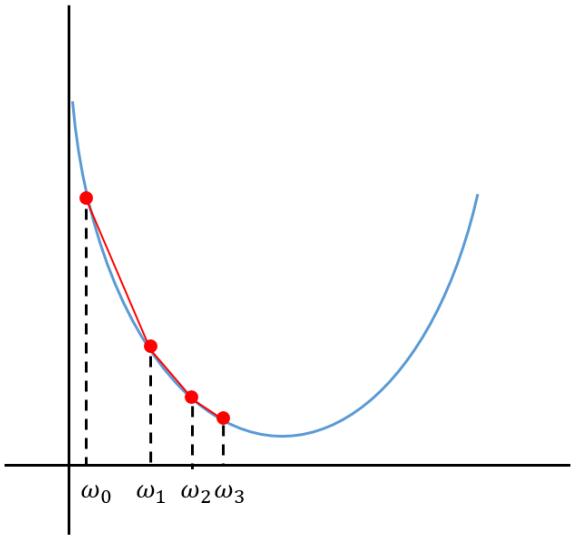

Figure 3: The first few steps of gradient descent for a function of one variable

Gradient descent works by means of the iteration:

$$w_{n+1} = w_n - \alpha \nabla \mathcal{L}(w_n)$$

Where w_n is a place holder for all our parameters and the gradient of our loss function is taken with respect to these parameters. The value α is our 'learning rate' which determines the degree to which our model can change its parameters at a given iteration of the algorithm. A high learning rate will allow the model

to optimize quickly but there is a danger of missing the optimal parameter values and having the gradient explode in value. If the learning rate is too small, there is little chance of the aforementioned problem but many iterations will be needed before reaching a desired level of optimization. The iteration works fairly simply, the gradient of the loss function is in the direction of greatest descent and we take steps in that direction from our initial starting point (i.e. our initialized parameters). The Adam optimizer works by altering the learning rate to reduce computational work and increase performance, it is not the point of this paper to fully explain how this works, but the paper written by the creators of Adam is given in the Further Reading section[1].

Activation Functions

The only thing that remains is to make our choice of activation functions. There are a few important considerations when making this choice. As discussed, we require the function to be differentiable. Secondly, it is important that the function is non-linear as, if we were to pick a linear function it is clear the output will simply be a linear combination of the inputs; a model we are trying to improve on. Lastly, it is important to think about the activation functions in the layer that leads to the output layer, we want the outputs we would expect to be in the range of the function and , ideally, to be exclusively the range of the function. For example, if our problem was to output probabilities, we would take a function which outputs values in the range: $[0, 1]$. In our case, we are simply predicting prices, so a function which obtains the values in $[0, \infty)$ will suffice.

I will largely be using the $ReLU$ function which is often the activation function of choice in Neural Networks. It is simply given by: $ReLU(x) := max(0, x)$. It's popularity is due to its prevention of the 'vanishing gradient' problem which is when an activation function has very small gradient at a point which causes back propagation to be very slow. Clearly the gradient of the $ReLU$ function is given by: 0 for $x < 0$ and 1 for $x > 0$ which clearly will not have the same problem.

In my model I also experimented, in part, with the more traditional $tanh$ function in the final hidden layer of the network.

3 Convolutional Neural Networks

Convolutional Neural Networks, or ConvNets for short, are a type of Neural Network mostly used for image data. In this case, the inputs are 2-dimensional arrays whose elements describe the pixel intensity at a given point in the image. The input may also have a number of 'filters' depending on whether it is a colour image or not, or if there are other features of the input. To apply ConvNets to

time series data, we have 1-dimensional inputs. There are three main types of layer in ConvNets: Convolutional, Pooling and Fully Connected (Dense).

Convolutional Layers

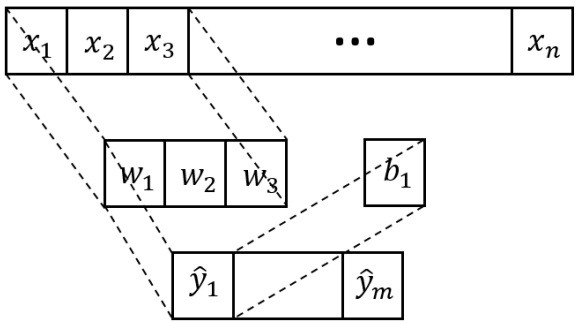

Figure 4: 1-dimensional convolution with a kernel of size 3

The idea of a convolution layer is to form a 'kernel', which is an array whose entries are the weights for the layer. This kernel scans across each input performing element-wise multiplications which are then summed and output to another array, generally of smaller size. It is common to 'pad' the inputs, most commonly with zeros, so that the entries at the edges of the input array do not intrinsically have less weight on them than the other values.

Pooling Layers

Pooling layers work similarly to convolutional layers in that a kernel scans across the input. However, instead of performing multiplications, the kernel 'pools' the entries it scans over, most commonly by averaging or taking a maximum, with the purpose of reducing the output size. It is common to pad the inputs of convolutional layers, as to keep the outputs the same size, then to systematically reduce the output size with pooling layers.

Fully Connected Layers

Fully Connected Layers are the same as in a traditional Neural Network. They are found at the end of a ConvNet, usually only a small number are used, and it is important that they output the correct size array into the output layer,

whether that be two nodes for a classification problem or, in our case, an array the length of time units we want to predict.

Intuition for Time Series Data

As stated previously, ConvNets are most commonly used for image data. The benefits of ConvNets compared to regular Neural Networks is in their 'local connectivity'. When the kernel scans over an input in a convolutional layer, by making element-wise multiplications it is associating the elements in the input together. This has the benefit of firstly being less computationally intensive than a fully connected layer, but in our case, with temporal data, we expect the data that is close in time to be more associated, so the way the network learns matches our intuition about time series data in this way. There are more features of ConvNets than I have mentioned here, there is a link to a more detailed explanation of ConvNets in Further Reading[2]. The Keras Documentation also explains thoroughly the multiple features you can implement in each layer.

4 Recurrent Neural Networks

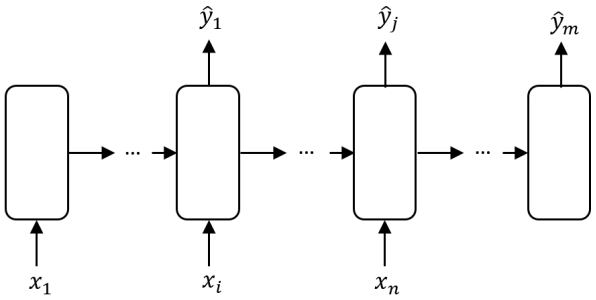

Figure 5: Structure of a Recurrent Neural Network

Recurrent Neural Networks are designed for sequence data and to have an emphasis on time dependency. They work by taking a sequence as an input, one element at a time, and creating outputs at each stage. Each unit of the network receives the next element of the input and the previous units output at each stage. As before, the outputs are made by a weight of the inputs and a bias that are fed into an activation function. Continually feeding the outputs of each unit back into the model has the effect of allowing the model to better 'remember' temporally earlier parts of the input.

LSTMs and GRUs

As is perhaps expected, the longer the input data is, the less weight the earlier input elements have as they're propagated deeper into the network. Ideally, we are able to predict as far into the future as possible, so a network architecture that has the capability to retain information about the input data in this way is certainly useful. This is where we turn to the design of the Long Short Term Memory (LSTM) Network. As well as the channel that propagates the input through each unit by making outputs and feeding them back into the model, the LSTM has a second channel running parallel to the first with the purpose of retaining desired properties in the data to be output later in the network. It does this by gates in each unit which decide whether a property of the input is worth keeping at an 'input' gate, and at each further unit, whether it still needs to be retained at a 'forget gate'. A slight modification on this architecture is given by the Gated Recurrent Unit (GRU) which combines the 'input' and 'forget' gates into one linear combination which allegedly decreases runtime without affecting performance. I have neglected to explain in detail the equations that dictate these architectures, these can be found in the link provided in Further Reading[3].

It is clear that LSTMs and GRUs are good candidate models for time series prediction as they are designed for temporal sequence data. I will test whether the ability of the models to retain information over a long period of time will allow for good predictions of the price of Bitcoin.

5 Data Processing and Method

Data

The dataset I used for predictions was daily Bitcoin prices from 2016 to 2018. As we are dealing with time dependent data, it is important that the test data is set aside first, then we are left with the training data on which we can choose the best method to train the models. To respect the time dependence of the data, I set aside the last 20% of Bitcoin prices for testing, and trained the models on the previous 80%.

Training

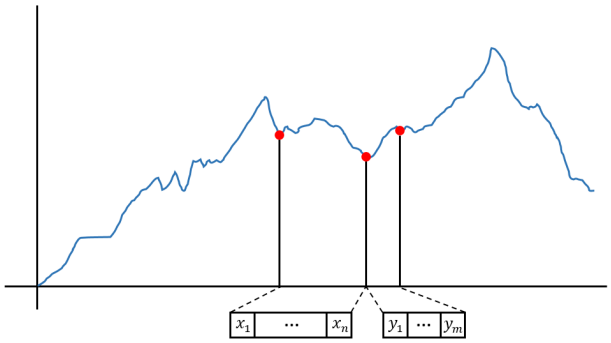

Figure 6: Sliding window

We repeat the same training and testing process for each model so we can compare performance. The input and output data are collections of 1-dimensional arrays which contain the prices of Bitcoin for a chosen number of days. Using the training data, we make a choice of values N and K which determine the number of consecutive days of prices in the input and output for the model respectively. i.e. we take a window of N days of prices and get the model to try and predict the next K days. We then show it the true prices for those K days and get it to update its parameters. We slide this window across systematically so that the model is always making predictions on unseen data. We do this until the model has been trained on all of the training data.

Testing

With our learned weights from training, we show the model the first N days of test data and get it to make a prediction on the next K days. I attempted two methods of forecasting. The first method was to feed the first N days of test data into the model and getting it to make predictions on the next K days. Then, without updating the weights of the model, test data is continuously fed into the model until it has made rolling predictions on the whole test set. The second method was to feed the initial prediction back into the model, and repeat this process until predictions are made on the whole test set also. The first method represents wanting to predict the next K days on a rolling basis, the second could be used to predict as far into the future as desired.

6 Results

The model was trained and tested with the choice of $N = 16$ and $K = 10$, so as in our previous discussion, the model would be taught how to predict 10 days in advance.

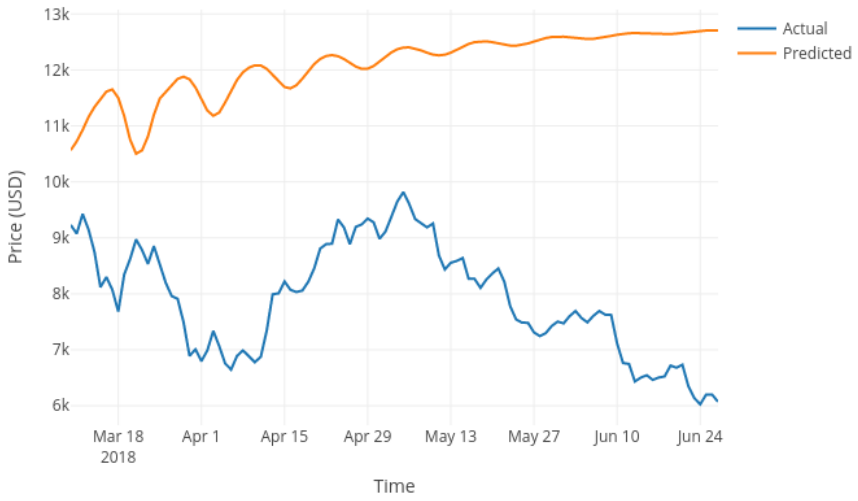

Figure 7: CNN Model Predictions

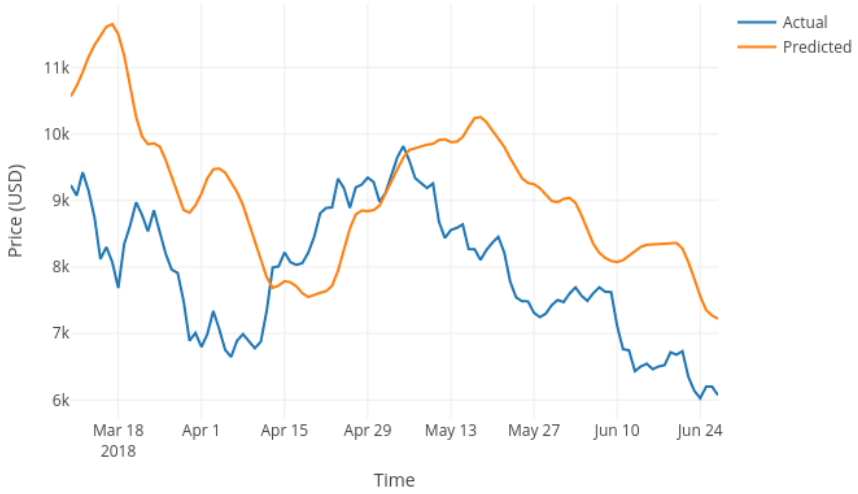

Figure 8: CNN Rolling Predictions

Out of the three models tested the CNN model performed the worst. It doesn't seemed to capture the behaviour of the Bitcoin price data, and also struggles to predict any large change in price. As was expected, the rolling predictions are better, but the model has appeared to learn to predict the future price as a lagged version of the previous price which is of little use in forecasting.

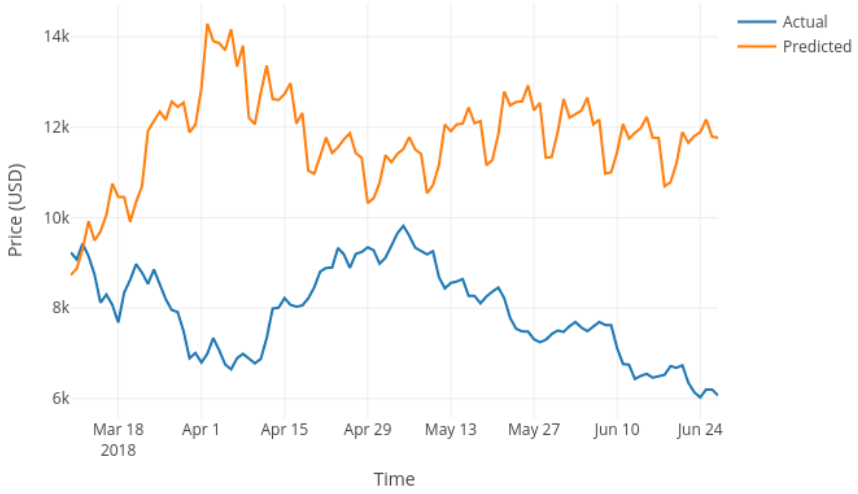

Figure 9: LSTM Model Predictions

Again, the predictions made by feeding predictions back into the model diverge from the true results quite quickly and similarly, in a small number of iterations of this process, an apparent periodicity appears in the predictions which are not present in the true prices.

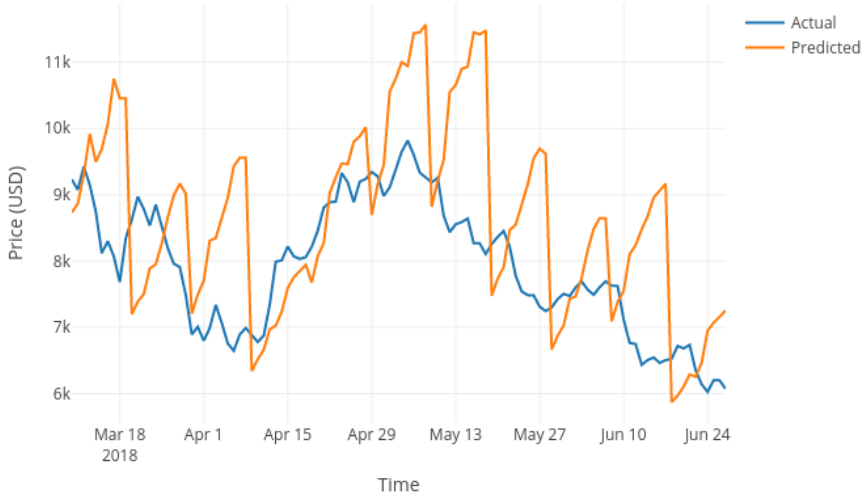

Figure 10: LSTM Rolling Predictions

The LSTM model seems to perhaps outperform the CNN model but generates more erratic predictions.

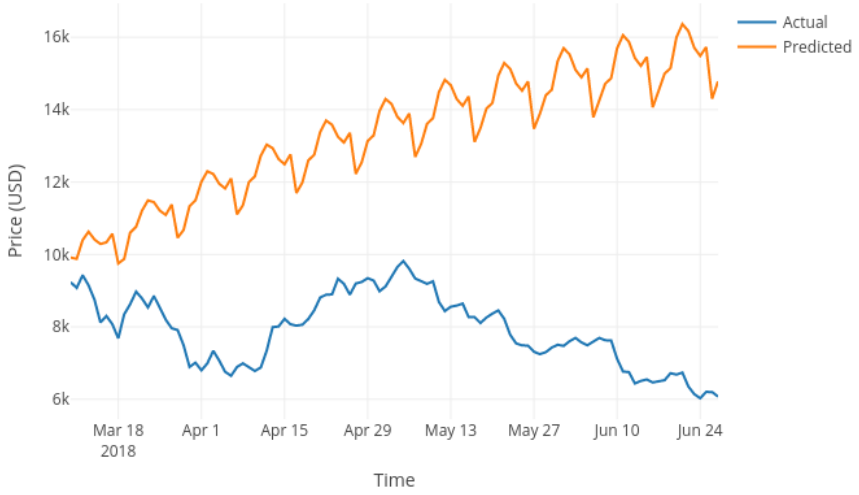

Figure 11: GRU Model Predictions

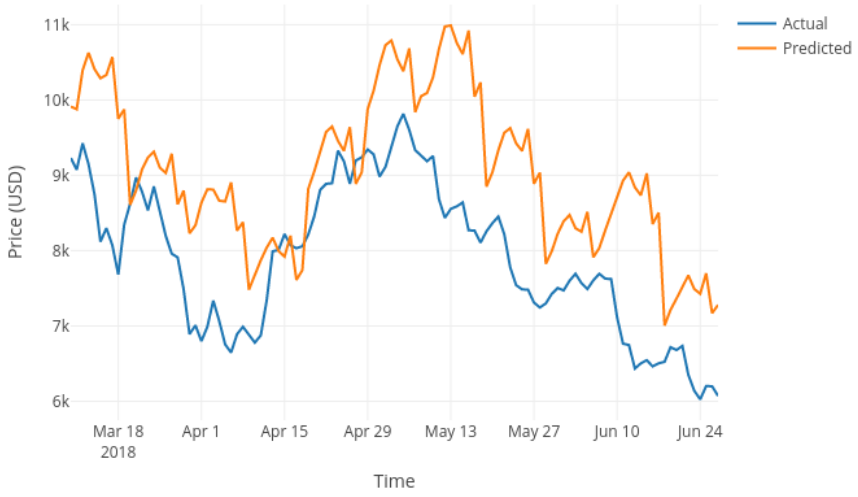

Figure 12: GRU Rolling Predictions

The GRU model appears to perform marginally better than the LSTM. However, the issues of periodicity in the first set of predictions and lag in the rolling predictions are clear.

7 Conclusion

As can be seen in the results, the deep learning models suffer from two problems. Firstly, feeding the predictions back into the models causes a periodicity which is not present in the data. The second problem is that the rolling predictions can only do as well generating a lagged version of the price, which is not really an improvement on the naive model of predicting the next days price to be the same as the previous. This is most likely as a result of punishing models via the MSE. This approach is not one that necessarily rewards capturing the behaviour of the price data and thus we are left with a model that always tends towards the naive model as it is trained on more data.

Going Forward

I am interested in considering the percentage change data for Bitcoin. Compared to the price data it stays in a fixed range and perhaps will be easier for the deep learning models to learn from. Further, the above models mentioned have the capability to input several filters of data, ideally in time series forecasting we have several variables which will help us make predictions whereas the predictions we made for future price were using only the previous price. One possibility for this is to use sentiment analysis on social media data regarding Bitcoin and

see if this increase the accuracy of predictions. Another is inputting the price data for other cryptocurrencies such as Litecoin and Ethereum, however there is a possibility that these features are highly correlated with the price of Bitcoin and will not help the models learn but may cause an unnecessary increase in noise.

Related Work

When learning how to process data for the models, the information from the following blog posts was very helpful in terms of how to handle time series and data and preparing data for use in Keras' Sequential Model; which is used to create CNNs, LSTMs and GRUs:

(a) https://medium.com/@huangkh19951228/predicting-cryptocurrency-price-with-tensorflow-and-keras-e1674b0dc58a

(b) https://medium.com/activewizards-machine-learning-company/bitcoin-price-forecasting-with-deep-learning-algorithms-eb578a2387a3

Further Reading

[1] Diederik P. Kingma and Jimmy Ba (2014). "Adam: A method for stochastic optimization". https://arxiv.org/abs/1412.6980

[2] http://cs231n.github.io/convolutional-networks/

[3] http://colah.github.io/posts/2015-08-Understanding-LSTMs/