Short Term Bitcoin Price Prediction with Deep Learning

Neural Networks to Forecast Cryptocurrency Prices with Python

Motivation

Due to the recent hype in cryptocurrencies -mainly all recording new all-time highs (ATH)- I found interesting to try to predict Bitcoin's price.

In a nutshell

The goal of this project is to predict Bitcoin's price with Deep Learning. More precisely, I'll be showing a stacked Neural Network model with Long Short-Term Memory cells (LSTM for short). Additionally, I will include 2 techniques to avoid overfitting called Early Stopping and Dropout. Lastly, the model will be validated and used to forecast BTC price for 10 future days.

Disclaimer

The model and predictions do not pretend to be used as financial advice. They were created for educational purposes and with a time constraint. Do not take financial decisions based on the results.

Remember: '*all models are wrong, but some are useful*' George E.P. Box

Dataset

The dataset used for this project is simply the BTC-USD data retrieved from [Yahoo Finance] including BTC Open, High, Low, Close, Adj Close and Volume for the desired time interval. In this case, I retrieved all data available and, in the script, I chose to use the latest 1200 observations (n_past_total) considering that the first years of it may present a behavior considerably different of what is shown nowadays.

Model Definition

The algorithm chosen for this analysis is a Long Short-Term Memory (LSTM) Neural Networks. The details about this kind of Recurrent Neural Network are out of the scope of this article, yet brief bullets may clarify the main points:

1. Recurrent Neural Networks (RNN) differs from a standard feed-forward approach by the use of previous input sources within the calculation.
2. A problem that may pop up while training a RNN is the explosion of gradient errors during iterations loop leading to unstable neural network.
3. LSTM solves this issue by adding the ability of memorizing and updating new information or just deleting them. This is possible thanks to a more complex architecture.

In this case, the model was developed with Keras which is a Python library that uses TensorFlow in the backend. Keras' API simplifies the implementation of the Neural Network.

Data preparation

The training dataset is the Close Price and Volume for past observations (n_past) and the output will be the Close Price predictions for n future days (n_future). Since I am going to use 2 variables as input, the model is called Multivariate.

The first step of the data preparation is to scale all the observations, in this case I used the MixMaxScaler, but the StandardScaler could also be used.

Secondly, I will re-arrange the training data in the required format for the RNN. For this, I defined the X_train array that contains the n_past observations that will be used to predict the n_future prices in a way that, if n_past is 30 and n_future is 10:

- X_train will contain: X_{t-30}, X_{t-29}, ..., X_{t-2}, X_{t-1}.
- y_train will contain: y_t, y_{t+1}, ..., y_{t+8}, y_{t+9}.

It is very important to understand that these vectors are needed for every possible iteration in the time window. Since the input of the model is the past 30 observations, it is not possible to predict values for the first 30 observations, leaving those datapoints to be used only as input. Additionally, since we are predicting for n_future days, the latest datapoints from the dataset won't belong to X_train, but instead only to y_train.

Furthermore, above explanation should be applied for each input feature we want to train the model with. But, given that what we want to predict is the BTC Price, meaning only 1 variable, y_train will only have 1 dimension, dropping all other features used as input.

So, to sum up, we should end up having the following matrices:

- X_train with shape (n_past_total – n_past – n_features, n_past, n_features)
- y_train with shape (n_past_total – n_past – n_features, n_future)

Building and training the model

The model was built with Keras, using the Sequential class and stacking different LSTM layers.

For input layer, I defined the quantity of nodes/neurons to be n_past so each of the variables are represented.

For the output layer, since we are building a Multi-Step model, the number of neurons should be the same number of future predictions we desire, named n_future.

In between the input layer and the output layer, a number of hidden layers can be added. There is no rule to determine the optimal quantity of hidden layers, nor their quantity of neurons.

In order to prevent overfitting, dropout and early stopping were included. As a quick recap:

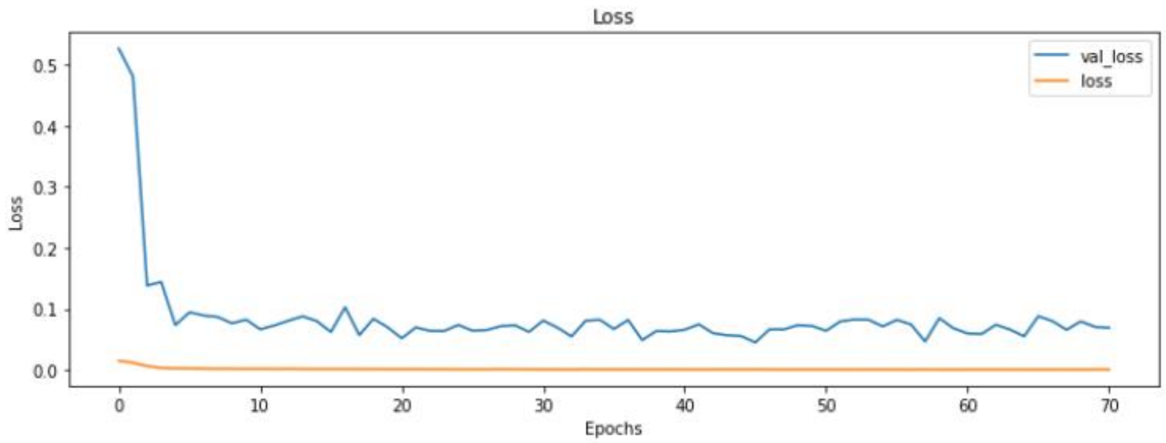
- Dropout is a layer added to use only a portion of the nodes of the next layer and drop-out the others; if dropout is 0.2, 20% of the nodes of the next layer will be ignored.
- Early stopping halts the training when a monitored metric has stopped improving. In this case, the metric we want to minimize is 'val_loss', and a patience of 25 which is the number of epochs with no improvement after which training will be stopped.

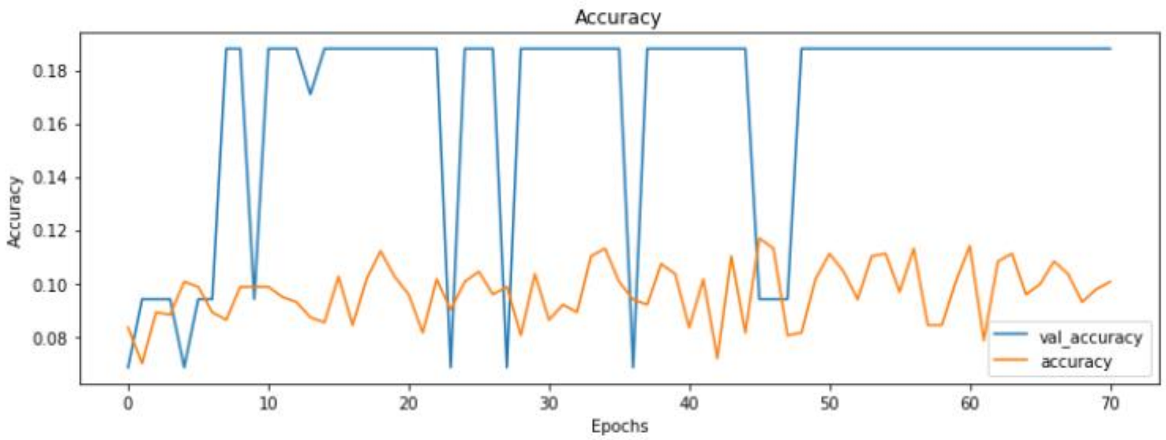
Model Evaluation

In order to evaluate the model's performance, the following analysis were done:

Loss and Accuracy

Plot the Loss and Accuracy with their corresponding validation sets defined as 0.1 (10%) in the 'model.fit' method.

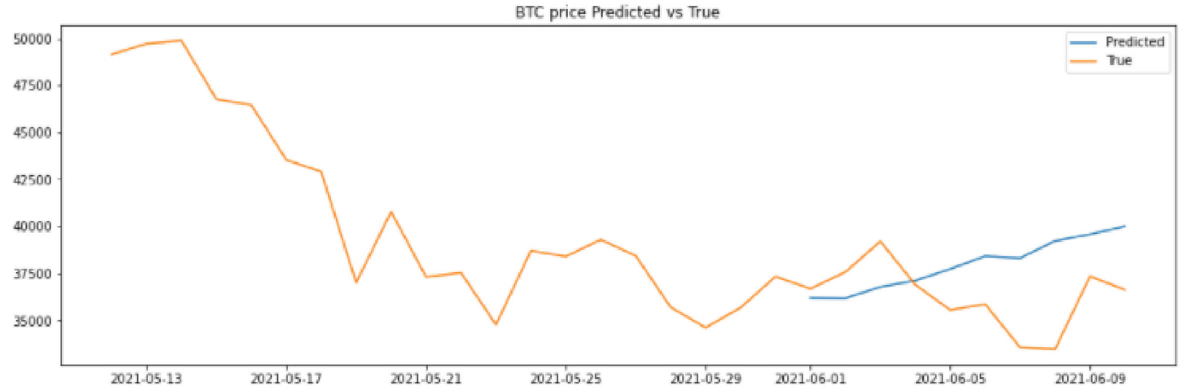




Ideally, we want to see both of them converging as the number of epochs increases. If both of them diverge from one another, it is a sign of overfitting. In this case, we can see they converge up to a point due to Dropout and Early Stopping. Without them, both will converge and then diverge once the model starts overfitting.

Predictions with real values

Once the training is complete and we are satisfied with how the Loss and Accuracy converged, we test our model against actual data to see how well it performs. We can do this by simply visualizing the model predictions and the actual values:

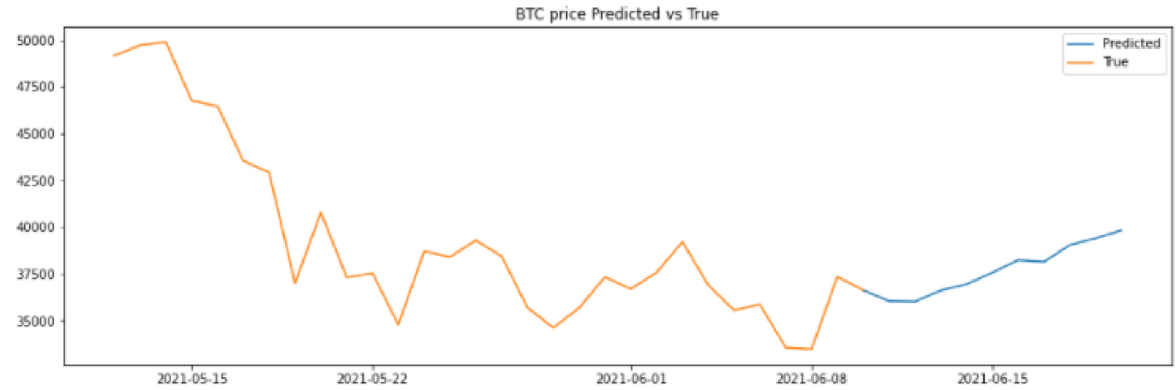


The results do not look very promising.

Trying with different parameters and RNN architecture in the allocated time, it was not possible to further improve the model performance. This is attributable to the high volatility of BTC Price, which is highly dependent on external factors rather than its own price and volume. For example, Elon Musk's tweets, countries releasing approval/restrictions, companies adopting BTC (VISA, PayPal, Tesla, etc).

Model Forecasting

Finally, I will predict and plot the Bitcoin's price for the next 10 days.



Parameter Tunning

In the beginning of the notebook, you may find a cell where the main parameters were extrapolated to do further parameter tunning.

Further improvements

Due to time contraints, I was not able to complete the project as I'd like, but here I will outline few ideas that can be further developed:

- Include other models to compare the results, such as, ARIMA, SARIMA, Facebook Prophet, among others. We could also try GRU cells in stead of LSTM for the RNN.
- Include more features that influence BTC price. E.g.: some other stock exchange price, Twitter data, Whale data, altcoins data, active BTC addresses, etc.
- Increase the number of datapoints by reducing the time intervals. For this notebook, I used daily prices, but we could try with hourly prices for example.

Documentation and Code Repository

Please have a look at my GitHub repository for the complete code explained in a Jupyter Notebook.

Feel free to comment or contact me in case you have any doubts or suggestions to improve the project.

If you want further documentation on LSTM, Bitcoin or other related topics, I have included a list with links that were useful while I was doing my project, check it out!

Thanks for reading!