Washburn University

Predicting Stock Prices using a Recurrent Neural Network

and Long Short-Term Memory cells

Ryan Glogau

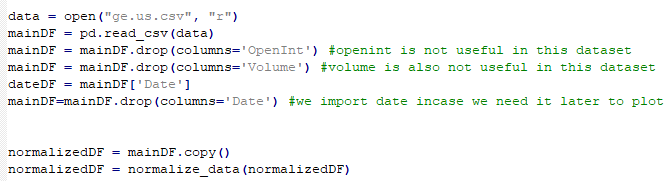
CM 332 – Data Mining

Dr. Cecil Schmidt

3 December 2020

Predicting Stock Price

For my project, I followed a guide by “Raoul” at <https://www.kaggle.com/raoulma/ny-stock-price-prediction-rnn-lstm-gru?select=prices-split-adjusted.csv> to implement a RNN using LSTM cells to predict stock price. Rather than using the stock data that he provided with his guide, I found my own set of US stock data from Kaggle that contained hundreds of stock symbols, each with at least 40 days’ worth of historical stock price data. I chose a file that contained GE (General Electric) stock data that spans from 1/2/1962 through 11/10/2017. The file is set up in the following order: Date,Open,High,Low,Close,Volume,OpenInt. Open and Close are the stocks opening and closing prices on that day, the High and Low are the stocks high and low prices for that day, the volume is the total number of stocks traded on that day, and the OpenInt is open interest which is the interest associated with the stock at the open of the market.

I implemented this project in python 3 and I used the libraries NumPy, pandas, matplotlib, and TensorFlow 2. NumPy and pandas are very useful libraries that I have used in almost every project I have made in python. They both are primarily used for easy data collection, alteration, and organization. Matplotlib is used to create graphs and for this project I created a time series graph, but there are many others that you can make with this library like bar graphs, scatterplots, etc. TensorFlow 2 is used to implement the actual neural network itself and has many other applications other than Recurrent Neural Networks.Figure 1.

I first start the program by importing my data from a CSV and storing it in a pandas data frame (which is like a fancy array and hash map put together). I dropped the column that contained OpenInt because it was all 0’s and won’t be used for this and, since it isn’t helpful in this case, I dropped the volume column as well, this is all shown in figure 1. I then made a copy of the dataframe and ran the original data frame through a normalization function called normalize\_data() as shown in figure 2. What normalize data does, is it goes through each value in the data frame passed to it and converts the value into a new value between 0 and 1. This is crucial in running data through any neural network. There is a library in python that automatically does this for you called sklearn, but I implemented this function before I found that out. I also looked at the documentation and my function is nearly identical to the function that library uses.

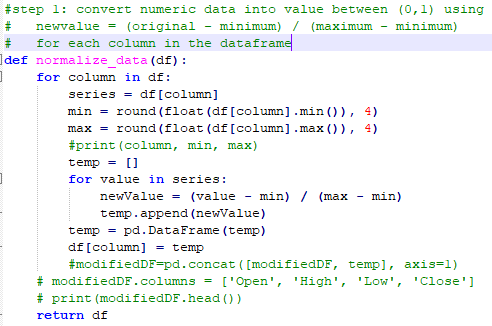


Figure 2.

After the data is normalized, I create the 3 different sets of data that I will be using for the neural network. I pass the normalized dataframe into the function create\_sets() (Figure 3), which splits the dataframe into three separate dataframes: the training data, the validation data, and the test data. The data is split up based on percentage, the validation and test data need to both be of equal size while the training data is the entire dataset minus the validation and test data. For this project, I used three different splits of data: the first one where validation and test data are each 10% (20% collectively) of the data set and the training data is 80%, the second one has validation and test data each being 15% (30% collectively) and the training was 70%, and the final one has validation and test data each being 20% (40% collectively) and the training was 60%.

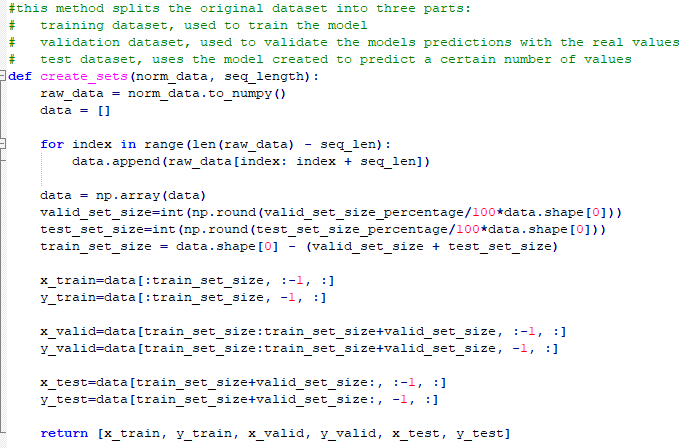


Figure 3.

The next part of the project is implementing the neural network. Trying to create a neural network of any kind from scratch in python is probably really hard, but the TensorFlow library makes it easier. The first thing I did was set up the parameters that the neural network will use. This includes number of inputs, neurons, outputs, and layers which have values of 4, 200, 4, and 2, respectively. The next set of parameters are learning rate, batch size, and number of epochs. Learning rate controls how much the network changes in response to the estimated error every time that the network weights are updated. Learning rate is very important because if it is too low, the network could get stuck and if it is too high the network could learn too fast which makes inaccurate results and weights. For this purpose, I didn’t change the value that I found in the guide which was 0.001. Batch size and number of epochs are the two variables that I tweaked to see if I could generate more accurate results for each model (by model I mean the split of training, validation, and test sets which I mentioned earlier).

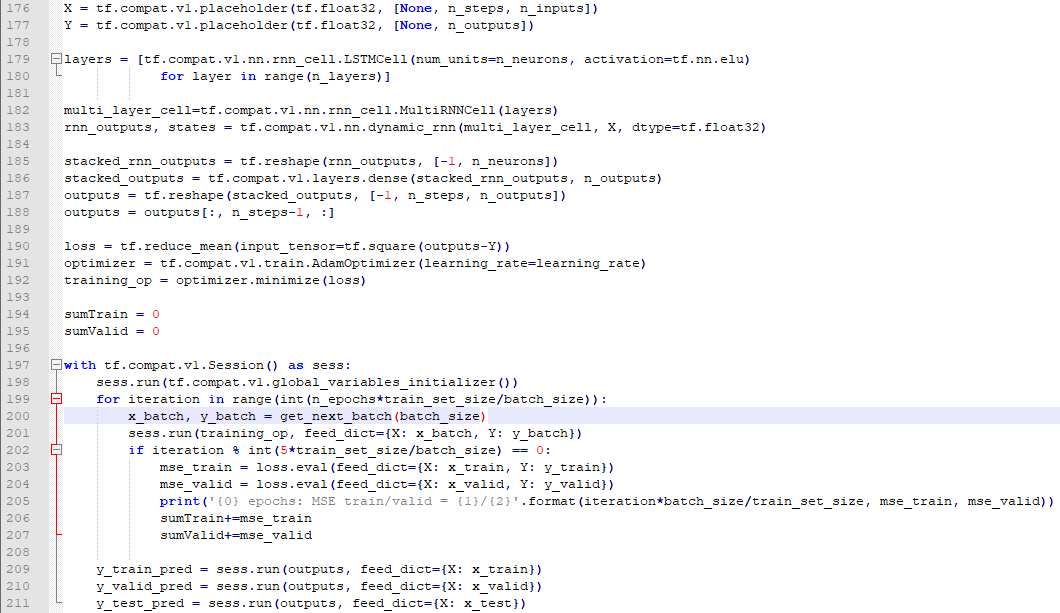


Figure 4.

Figure 4 shows all the code required to create and run the neural network. Starting at line 176, X and Y are placeholder variables that get allocated into memory but don’t have any current values. They get their values when feed\_dict is called on lines 201, 203, 204, and 209-211. On line 179, I declare what type of cell this RNN will use, for this project I used LSTM (Long Short-Term Memory) cells, but there are other options like a basic RNN cell, a LSTM cell with peephole connections, or a GRU (Gated Recurrent Unit). On line 182, I create the RNN cell out of the LSTM cell “layers” declared above it and then create the actual RNN on line 183 out of the RNN cell. Lines 185-188 format the output of the network using multiple instances of the reshape(x, y) method which changes a given tensor, *x*, into the given format, *y*. Loss, optimizer, and training\_op are all variables used in the network that optimize the output of the network according to a given “optimizer”. The loop that starts on line 199 is where the actual calculations are being done. Starting with “for each iteration in range”, it runs through the RNN with sess.run(). It then calculates mean squared error of both the training and validation set, and then runs again for however many epochs you tell it to run through.

Output

After the network has fully ran through however many epochs you tell it to, it generates a graph (Figure 5) using matplotlib. You can interact with either plot (this is comprised of two subplots, one that shows the entire dataset and one that shows just the test predictions) by zooming in or out and panning around the graph. To determine accuracy, I zoomed in on a particular portion of the test value subplot that I thought best represented accuracy. I believe from day ~11980 to ~12060 there is enough variance in values to show if the network can accurately predict the data.

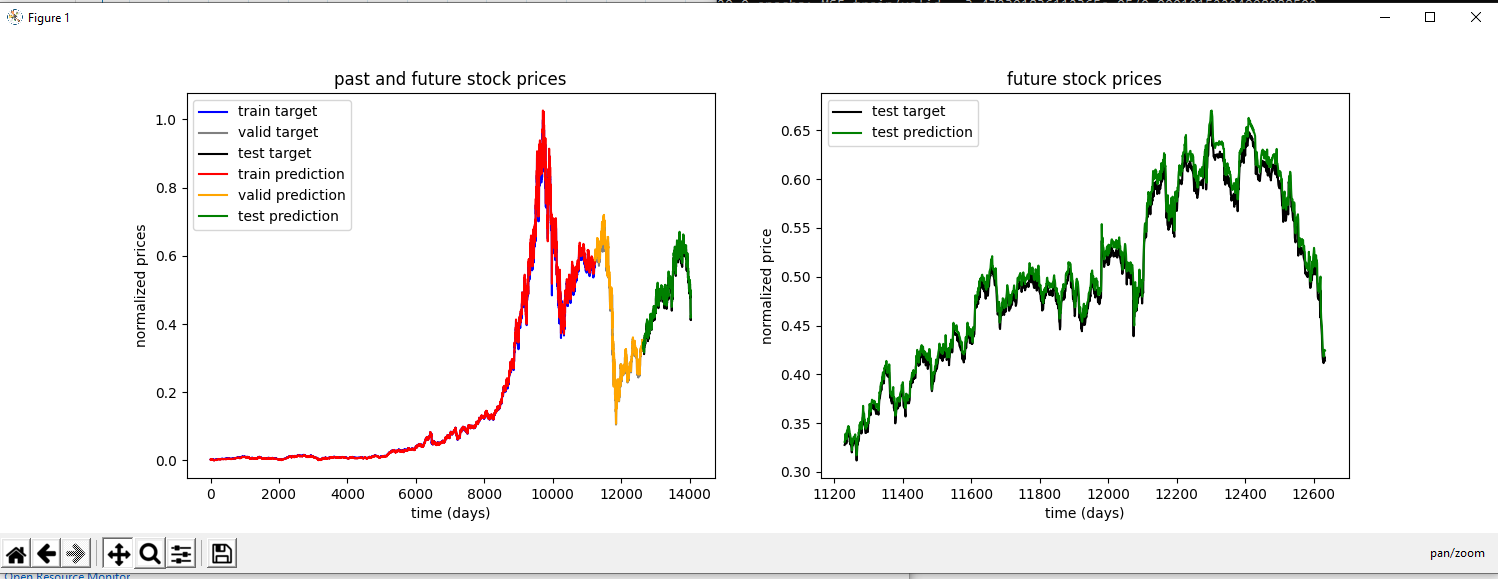


Figure 5.

Results

For the testing of this network, as I mentioned earlier, I broke it into three sub models that each break the complete dataset into different portions. For each sub model I ran the python script 5 times. The first time had 100 epochs and a batch size of 75, the second run had 100 epochs and a batch size of 25, the third run (which I used as sort of a benchmark because the values are in the middle of the other values) had 100 epochs and a batch size of 50, the fourth run had 75 epochs and a batch size of 50, and the fifth run had 125 epochs and a batch size of 50. For each test I also recorded the average mean squared error for each group of 5 epochs and the time it took (in minutes) to run through all the epochs. I provide the table that has all of the results on the following page.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Trial: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Number of Epochs | 100 | 100 | 100 | 75 | 125 | 100 | 100 | 100 | 75 | 125 | 100 | 100 | 100 | 75 | 125 |
| Batch Size | 75 | 25 | 50 | 50 | 50 | 75 | 25 | 50 | 50 | 50 | 75 | 25 | 50 | 50 | 50 |
| Time (minutes) | 10 | 15 | 13 | 10 | 16 | 9 | 14 | 11 | 8 | 14 | 8 | 11 | 10 | 7 | 12 |
| avg mse for train | 0.20% | 0.20% | 0.18% | 0.16% | 0.22% | 0.09% | 0.13% | 0.13% | 0.12% | 0.09% | 0.004% | 0.005% | 0.003% | 0.004% | 0.005% |
| avg mse for validate | 0.44% | 0.42% | 0.40% | 0.35% | 0.48% | 0.65% | 0.89% | 0.91% | 0.85% | 0.65% | 0.77% | 0.90% | 0.71% | 0.86% | 0.91% |
| training set (%) | 80 | 80 | 80 | 80 | 80 | 70 | 70 | 70 | 70 | 70 | 60 | 60 | 60 | 60 | 60 |
| validate set (%) | 10 | 10 | 10 | 10 | 10 | 15 | 15 | 15 | 15 | 15 | 20 | 20 | 20 | 20 | 20 |
| test set (%) | 10 | 10 | 10 | 10 | 10 | 15 | 15 | 15 | 15 | 15 | 20 | 20 | 20 | 20 | 20 |

The results show that as the amount of data that the network has to train itself on decreases, the mse for the validation (and testing) will increase. Interestingly, the graphs do not necessarily reflect this. For example, Figure 6 is the graph of the “best” performing run according to the mse of the validation set (trial 4), and Figure 7 is the graph of one of “worst” performing runs according to the avg mse of the validation (trial 15).

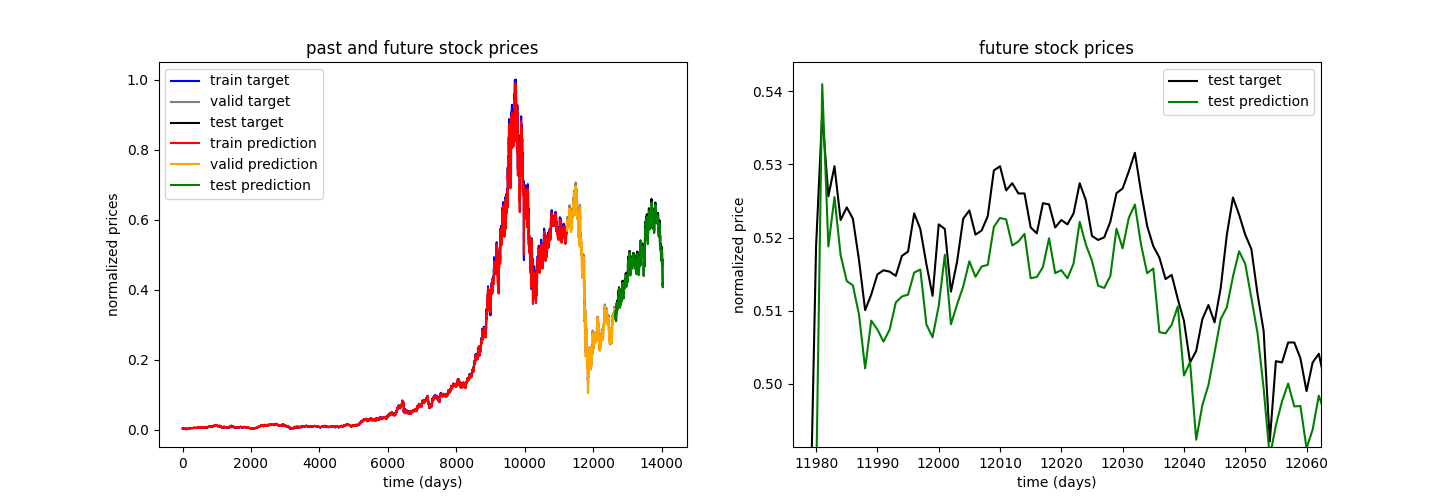


Figure 6

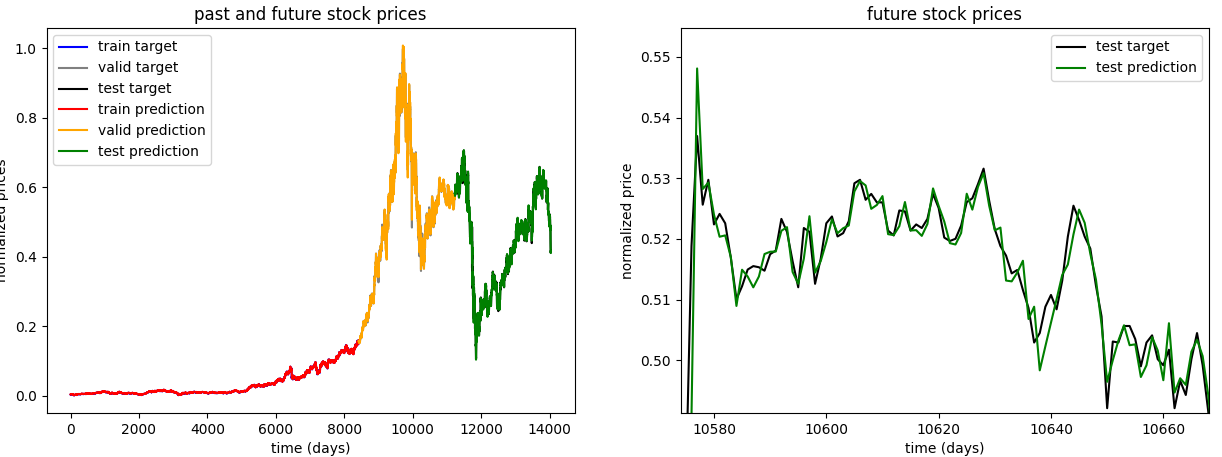


Figure 7

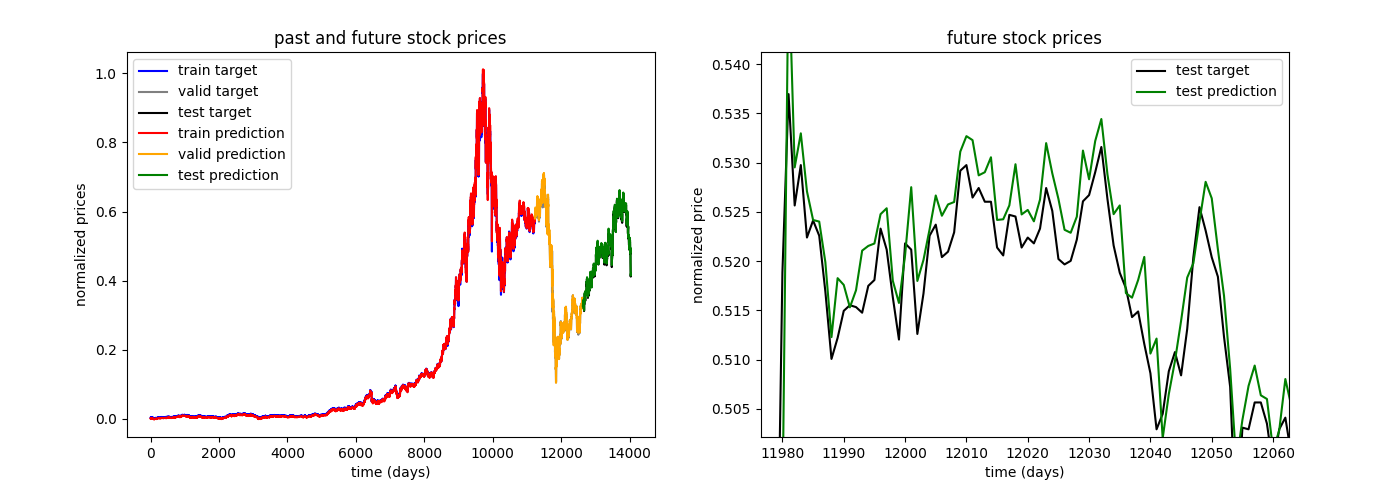
As you can see, there is a stark difference in the predictions of the “best” performing trial and one of the “worst” performing trials. Additionally, the set of predictions that came from the model with a training set size of 70% were some of the most consistently poor predictions, while the training set size of 60% had seemingly the most accurate set of predictions.

The results bring into question the accuracy of this neural network which is a very complicated question because it depends on the definition of “accurate”. Accuracy typically means how consistently an archer can hit a target in the bullseye or how often a basketball player can make a shot; however, in this situation it is not that simple. If you determine accuracy of this network based on how consistently the test predictions are almost exactly aligned with the test target, you’ll be disappointed. However, if you determine accuracy of this network based on how consistently the test prediction follows the general direction of the test target, you’ll be satisfied. The latter of which could be more useful in the real world when trying to predict stock prices and maximize profit.

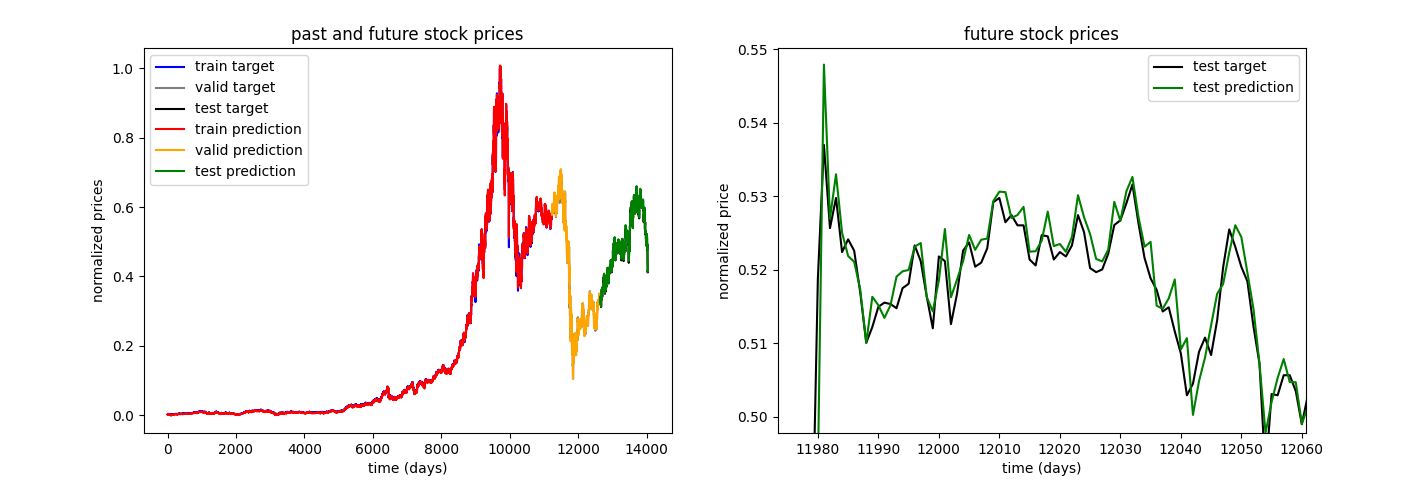
Conclusion

So, could this network be used in the real world to predict stock prices? Well, probably not. Although some tests on this neural network showed very accurate test predictions, this network predicts price based on historical data. In the real world, historical data related to stock prices is not what you would want to base a decision to sell or buy a stock on. What’s more helpful is whether or not the company met its earnings goals in the previous quarter, whether the executives of the company are doing a good job to propel the company to more success, or whether there is even a market for the product the company offers. Unfortunately, this neural network doesn’t look at factors like earnings or executive leadership, which would truly determine the price of a stock. What this network would be more useful for is showing trends on whether the stock price will go up or down which could help determine when to buy or sell the stock. This would be a much better use because even the inaccurate predictions like those in figure 6 show and match general trends: when the test target increases, the test prediction increases, when the test target decreases, the test prediction decreases. Trends can be more valuable than getting inaccurate “exact” prices in the stock market. If you knew a stock you were going to buy would increase by X-percent, +/- a few percent, in a given time period, you would probably be more willing to buy it than if you had no foresight.

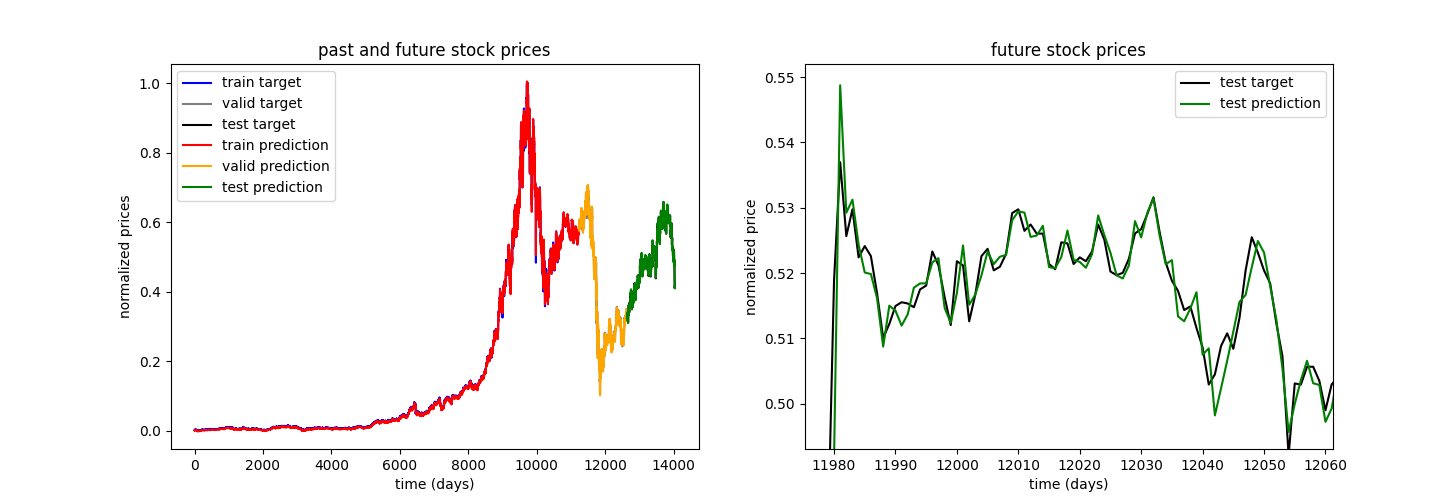
Outputs (epochs, batch size, train %, validate %, test %)



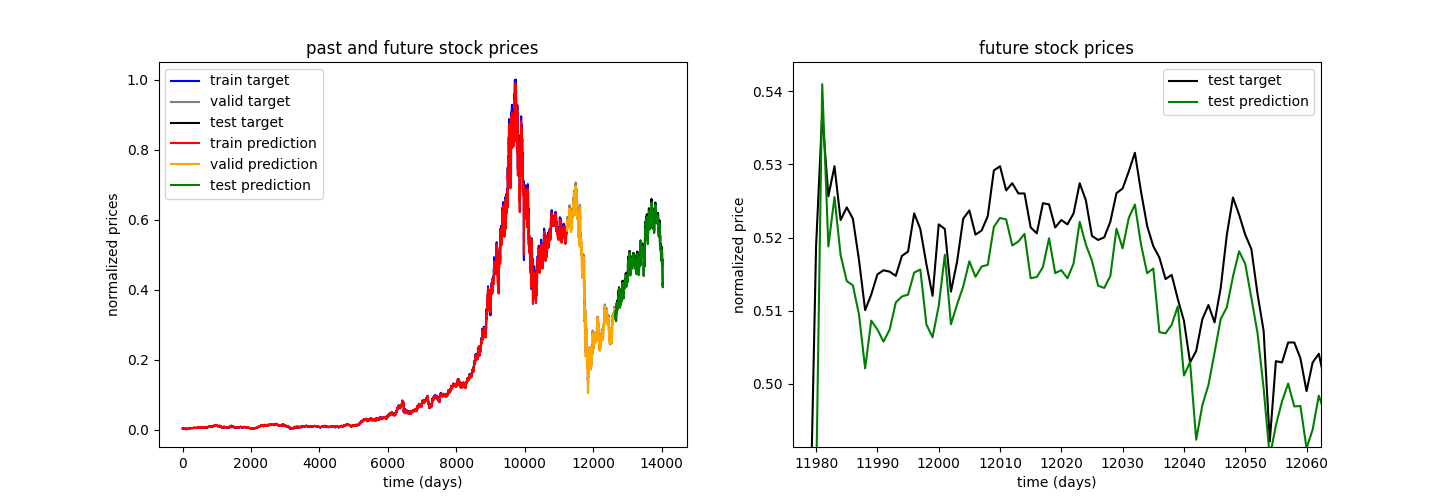
100 epochs, 25 batch size, 80/10/10 percent train, validate, test



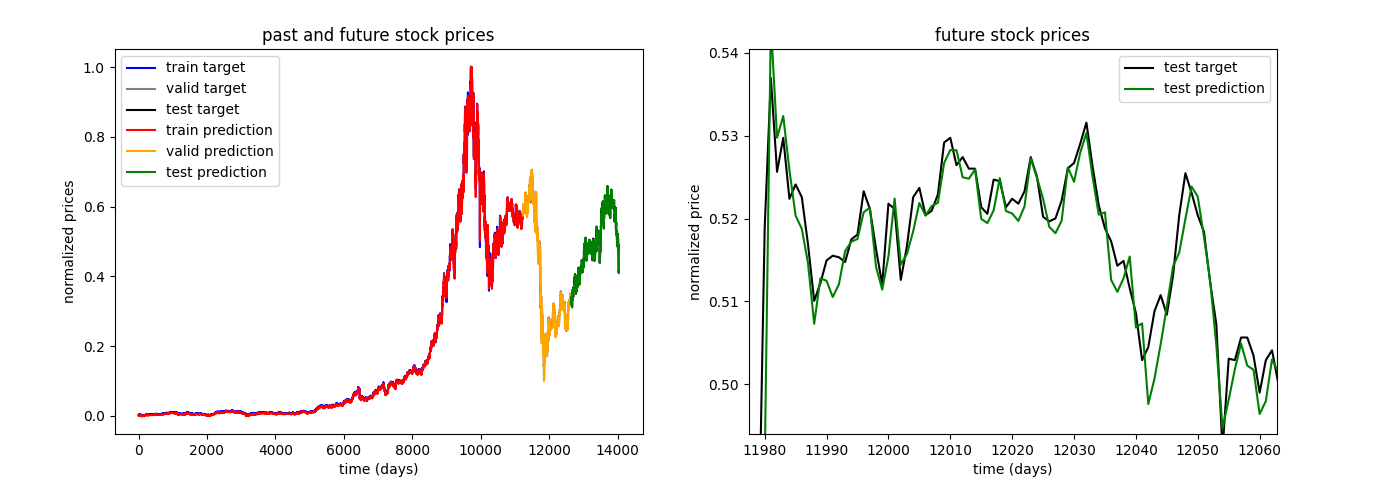
100 epochs, 50 batch size, 80/10/10 percent train, validate, test



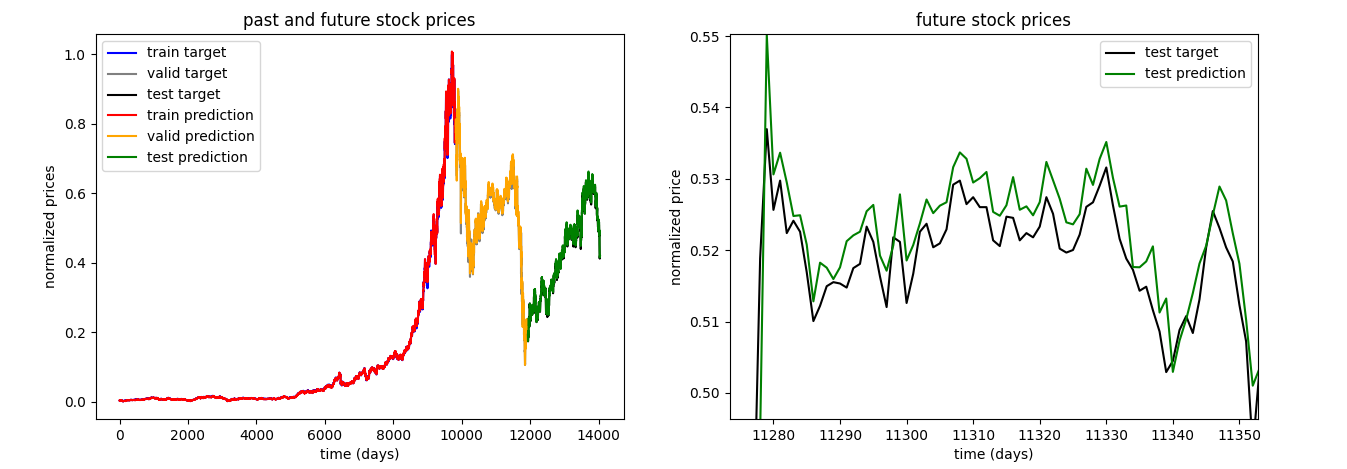
100 epochs, 75 batch size, 80/10/10



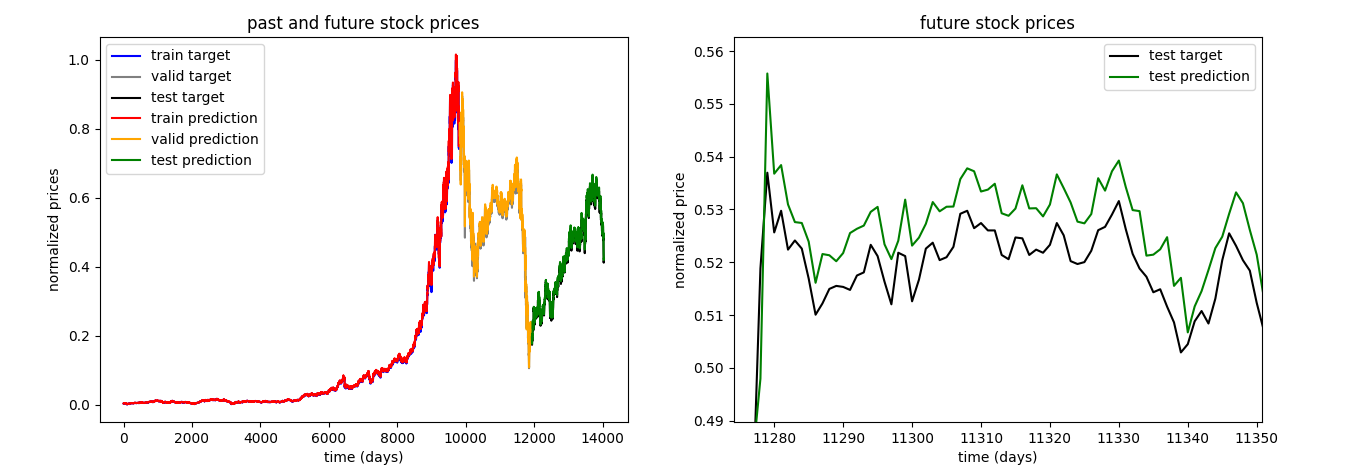
75 epochs, 50 batch size, 80/10/10



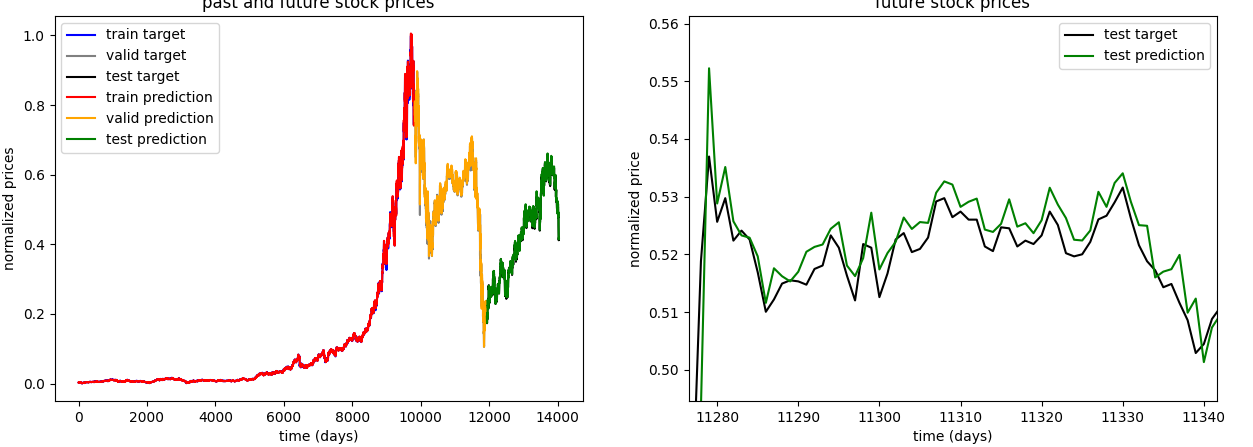
125 epochs, 50 batch size, 80/10/10



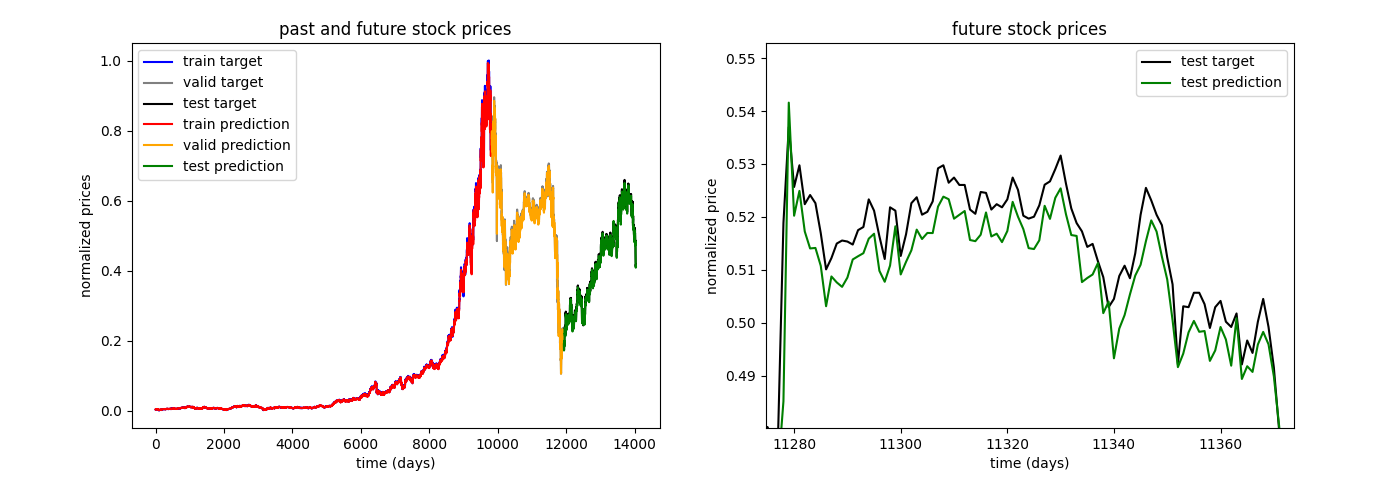
100 epochs, 25 batch size, 70/15/15



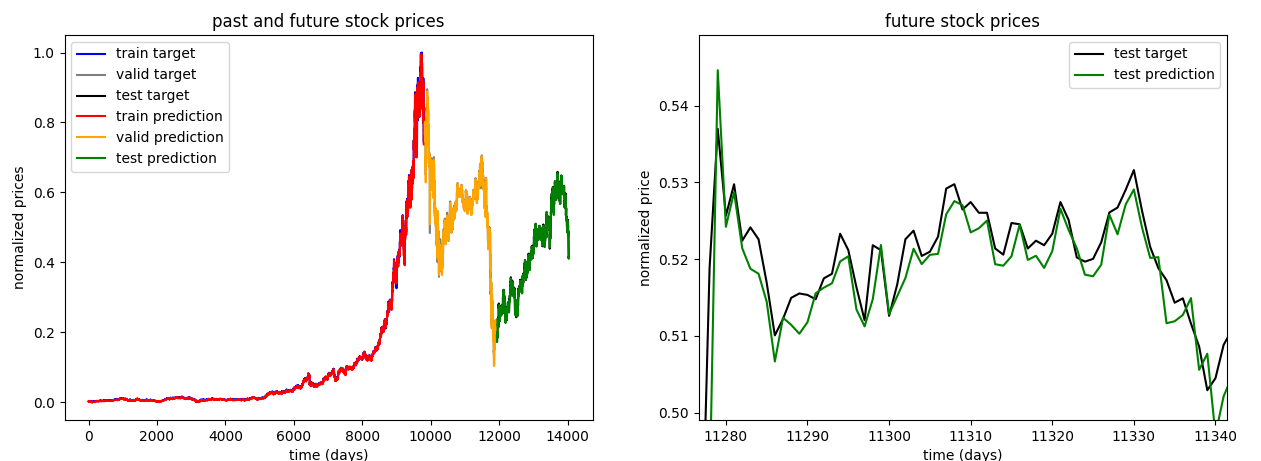
100 epochs, 50 batch sizes, 70/15/15



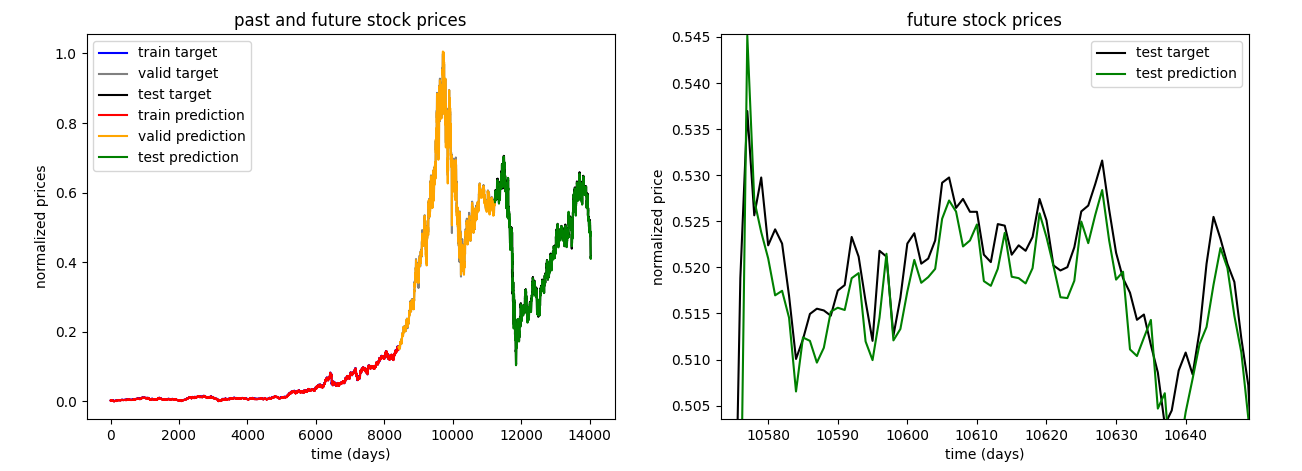
100 epochs, 75 batch size, 70/15/15

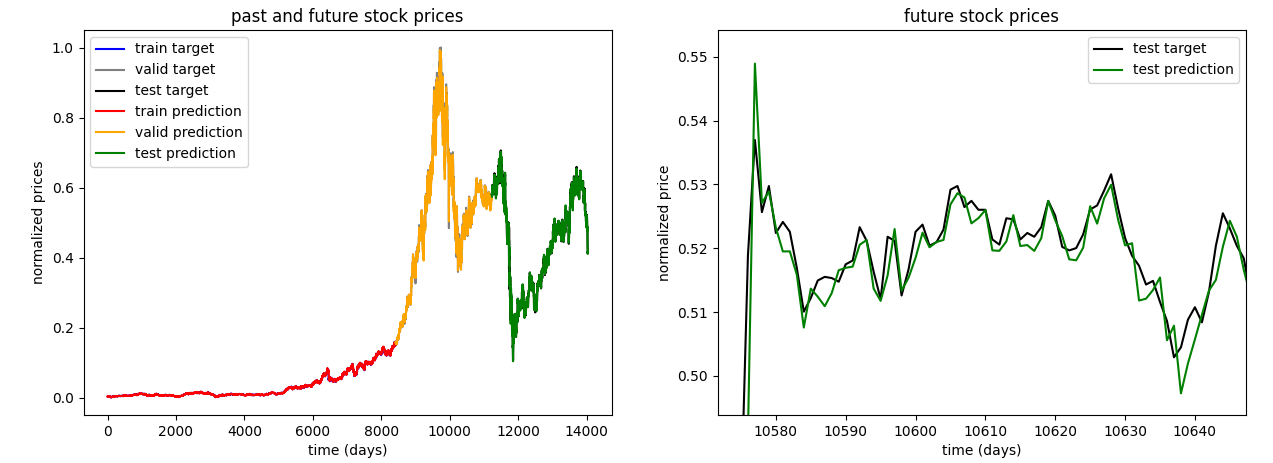


75 epochs, 50 batch size, 70/15/15

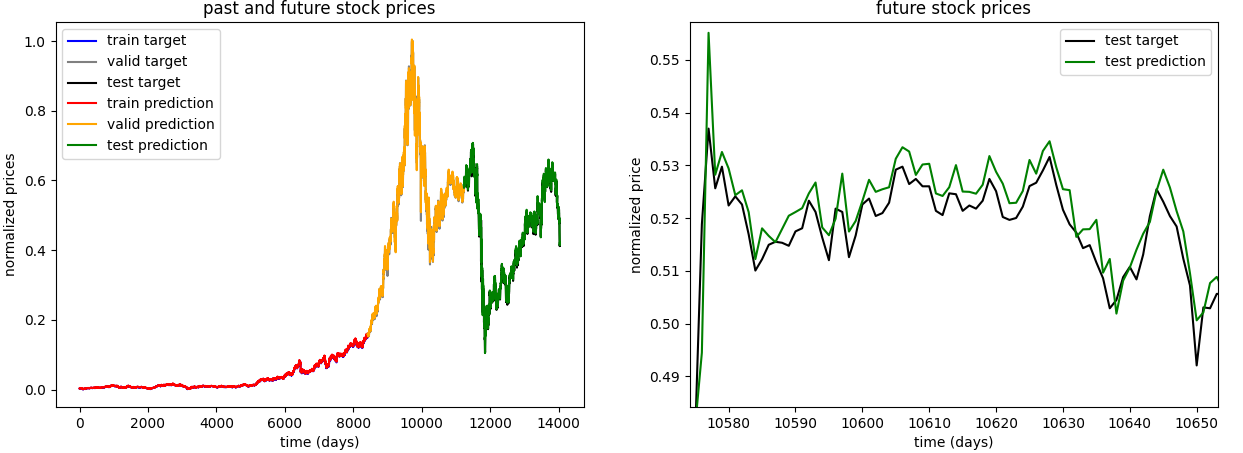


125 epochs, 50 batch size, 70/15/15

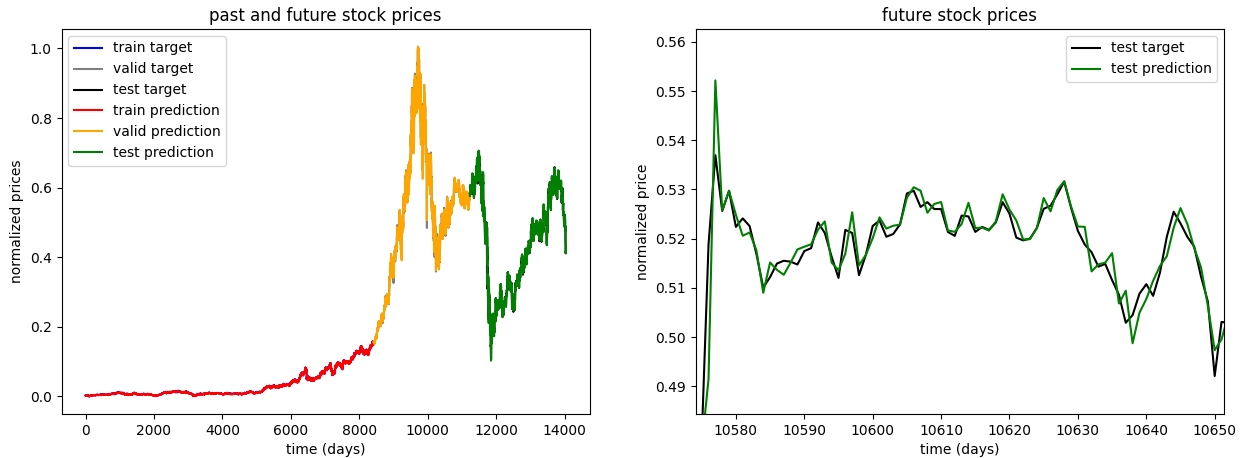
100 epochs, 25 batch size, 60/20/20



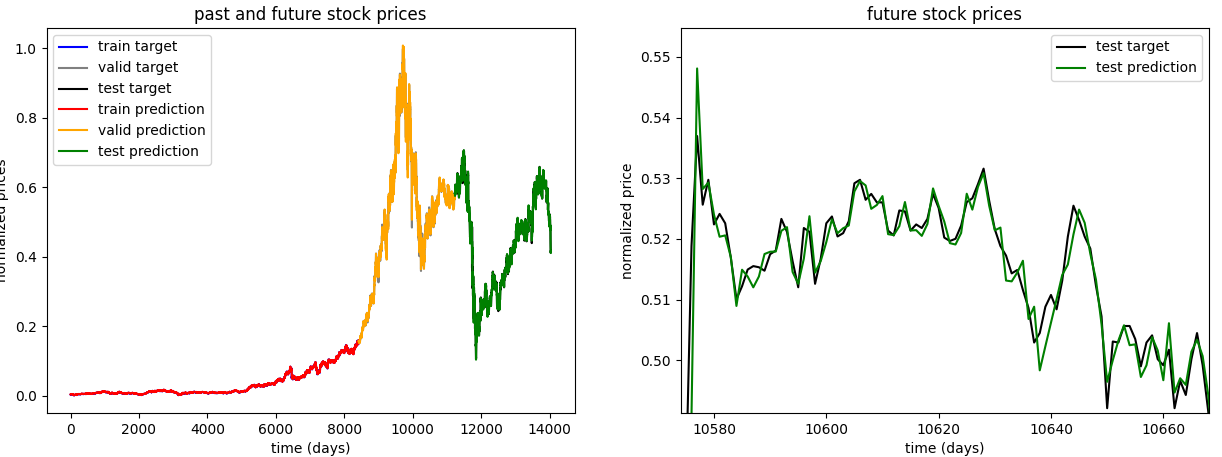
100 epochs, 50 batch size, 60/20/20



100 epochs, 75 batch size, 60/20/20



75 epochs, 50 batch size, 60/20/20



125 epochs, 50 batch size, 60/20/20