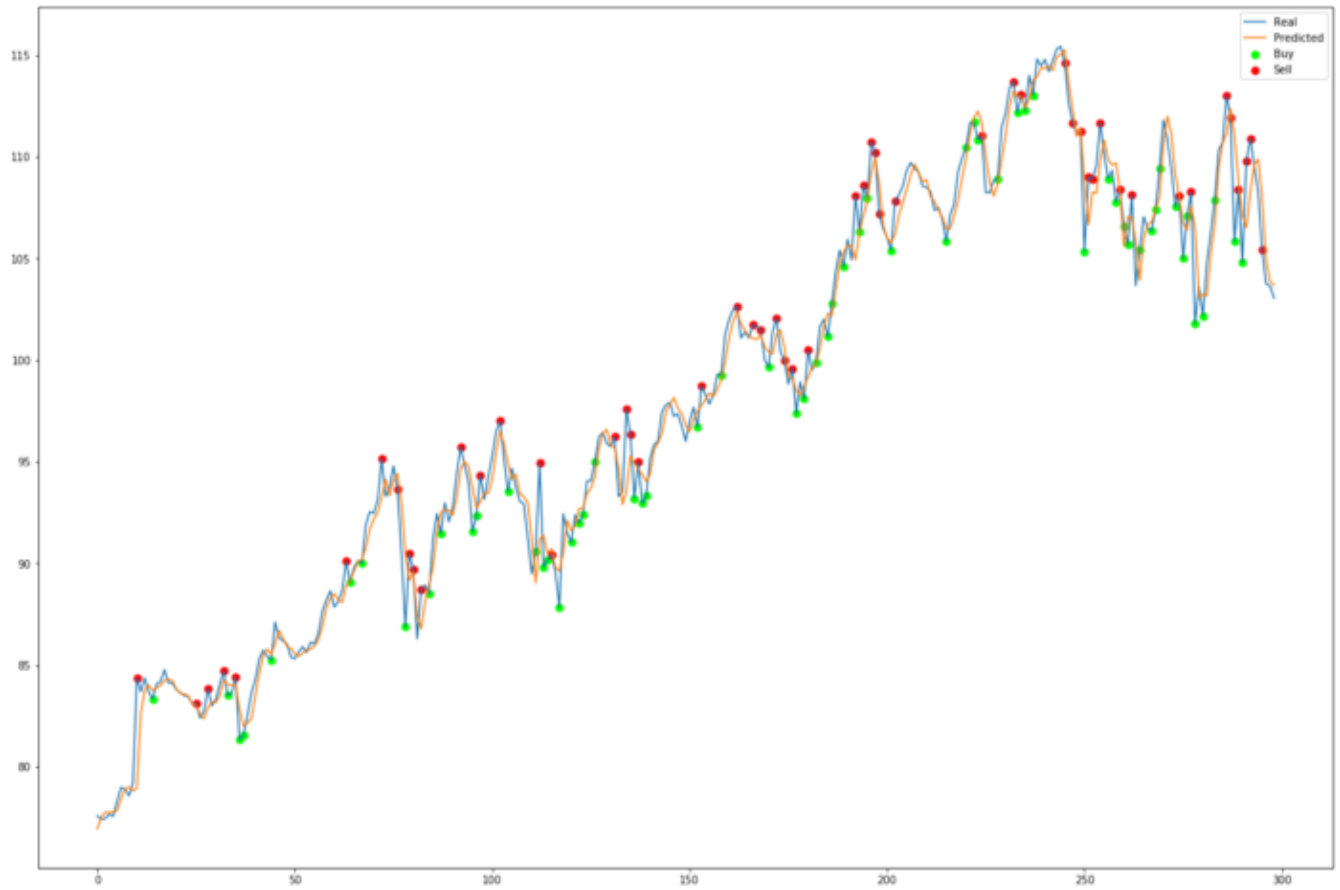
Published in Towards Data Science

Yacoub Ahmed Follow

Oct 11, 2019 · 12 min read ★ · Listen

Predicting stock prices using deep learning

If a human investor can be successful, why can't a machine?

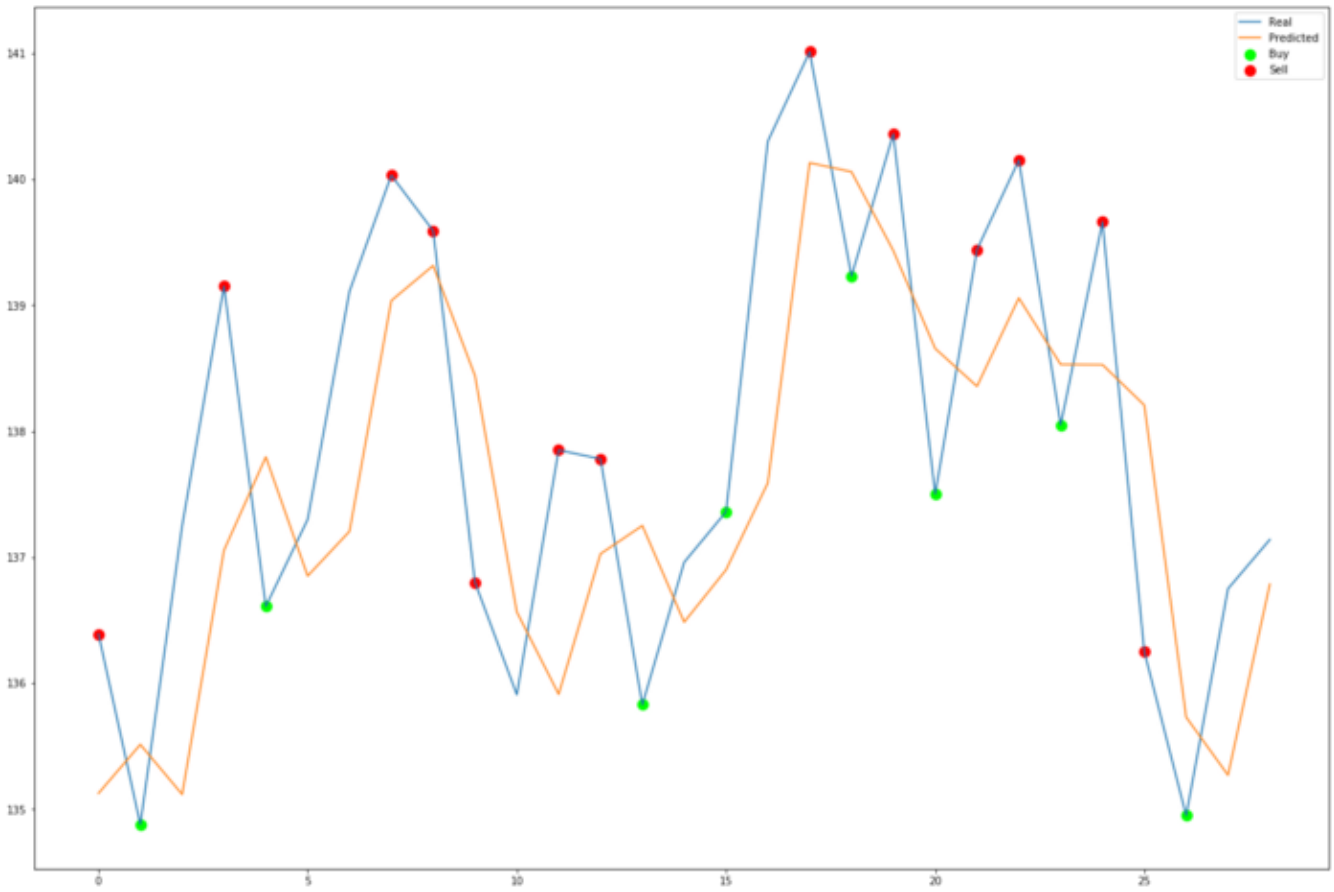


I would just like to add a disclaimer — this project is entirely intended for research purposes! I'm just a student learning about deep learning, and the project is a work in

Algorithmic trading has revolutionised the stock market and its surrounding industry. Over 70% of all trades happening in the US right now are being handled by bots[1]. Gone are the days of the packed stock exchange with suited people waving sheets of paper shouting into telephones.

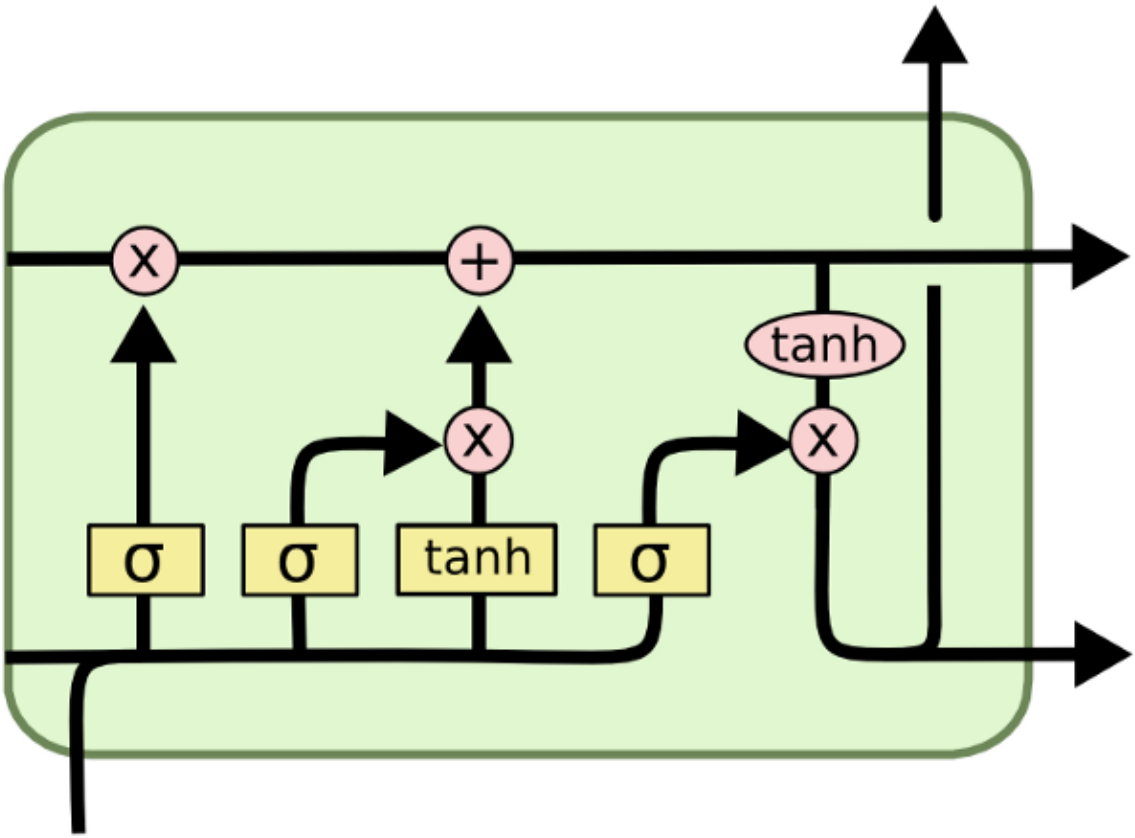
This got me thinking of how I could develop my own algorithm for trading stocks, or at least try to accurately predict them.

And the results weren't bad!



My trading algorithm for the MSFT stock September — October 2019

I've learned a lot about neural networks and machine learning over the summer and one of the most recent and applicable ML technologies I learnt about is the LSTM cell [2].



An LSTM cell. Credit: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory cells are like mini neural networks designed to allow for memory in a larger neural network. This is achieved through the use of a recurrent node inside the LSTM cell. This node has an edge looping back on itself with a weight of one, meaning at every feedfoward iteration the cell can hold onto information from the previous step, as well as all previous steps. Since the looping connection's weight is one, old memories wont fade over time like they would in traditional RNNs.

LTSMs and recurrent neural networks are as a result good at working with time series data thanks to their ability to remember the past. By storing some of the old state in these recurrent nodes, RNNs and LSTMs can reason about current information as well as information the network had seen one, ten or a thousand steps ago. Even better, I don't have to write my own implementation of an LSTM cell; they're a default layer in Tensorflow's Keras.

If you want to jump straight into the code you can check out the GitHub repo :)

Note from Towards Data Science's editors: *While we allow independent authors to publish articles in accordance with our rules and guidelines, we do not endorse each author's contribution. You should not rely on an author's works without seeking professional advice. See our Reader Terms for details.*

The Dataset

The good thing about stock price history is that it's basically a well labelled pre formed dataset. After some googling I found a service called AlphaVantage. They offered the daily price history of NASDAQ stocks for the past 20 years. This included the open, high, low, close and volume of trades for each day, from today all the way back up to 1999. Even better, a python wrapper exists for the service. I got my free API key from the website and downloaded Microsofts daily stock history.

```
from alpha_vantage.timeseries import TimeSeries
import json


def save_dataset(symbol):
    credentials = json.load(open('creds.json', 'r'))
    api_key = credentials['av_api_key']

    ts = TimeSeries(key=api_key, output_format='pandas')
      data, meta_data = ts.get_daily(symbol, outputsize='full')

    data.to_csv(f'./{symbol}_daily.csv')
```

save_data_to_csv.py hosted with ❤ by GitHub view raw

Since AlphaVantage's free API only allows for 5 calls per minute (and up to 500 calls a day) I opted to download the datasets and save them in CSV format so I could use them

volume value also affected how other volume values in the dataset were scaled when normalising the data, so I opted to drop the oldest data points out of every set. I also drop the date since the model doesn't need to know anything about *when* trades happened, all it needs is well ordered time series data.

```
import pandas as pd
from sklearn import preprocessing
import numpy as np

history_points = 50

def csv_to_dataset(csv_path):
    data = pd.read_csv(csv_path)
    data = data.drop('date', axis=1)
    data = data.drop(0, axis=0)
```

dataset_0.py hosted with ❤ by GitHub view raw

I also keep track of the number of {*history_points*} we want to use; the number of days of stock history the model gets to base its predictions off of. So if history_points is set to 50, the model will train on and require the past 50 days of stock history to make a prediction about just the next day.

Now we have to normalise the data — scale it between 0 and 1 — to improve how quickly our network converges[3] . Sklearn has a great *preprocessing* library capable of doing this.

```
    data_normaliser = preprocessing.MinMaxScaler()
    data_normalised = data_normaliser.fit_transform(data)
```

dataset_1.py hosted with ❤ by GitHub view raw

data_normalised now contains the normalised stock prices.

Now to get the dataset ready for model consumption.

```
    # using the last {history_points} open high low close volume data points, predict the next
    ohlcv_histories_normalised =      np.array([data_normalised[i  : i + history_points].copy()
```


```
8
9       y_normaliser = preprocessing.MinMaxScaler()
10      y_normaliser.fit(np.expand_dims( next_day_open_values ))
```

dataset_2.py hosted with ❤ by GitHub view raw

The *ohlcv_histories* list will be our x parameter when training the neural network. Each value in the list is a numpy array containing 50 open, high, low, close, volume values, going from oldest to newest. This is controlled by the *history_points* parameter, as seen inside the slice operation.

So for each x value we are getting the [i : i + history_points] stock prices (remember that numpy slicing is [inclusive:exclusive]), then the y value must be the singular [i + history_points] stock price; the stock price for the very next day.

Here we also have to choose what value we are intending on predicting. I decided to predict the **open** value for the next day, so we need to get the 0-th element of every ohlcv value in the data, hence *data_scaled[:,0]*.

There's also a variable called y_normaliser to hold on to. This is used at the end of a prediction, where the model will spit out a *normalised* number between 0 and 1, we want to apply the reverse of the dataset normalisation to scale it back up to real world values. In the future we will also use this to compute the real world (un-normalised) error of our model.

Then to get the data working with Keras I make the y array 2-dimensional by way of *np.expand_dims()*. And finally I keep hold of the unscaled next day open values for plotting results later.

Just before returning the data we check that the number of x's == the number of y's.

```
1       assert ohlcv_histories_normalised.shape[0] == next_day_open_values_normalised.shape[0]
2       return ohlcv_histories_normalised, next_day_open_values_normalised, next_day_open_values, y_
```

dataset_3.py hosted with ❤ by GitHub view raw

Now you can get any dataset you have the csv file for by running:

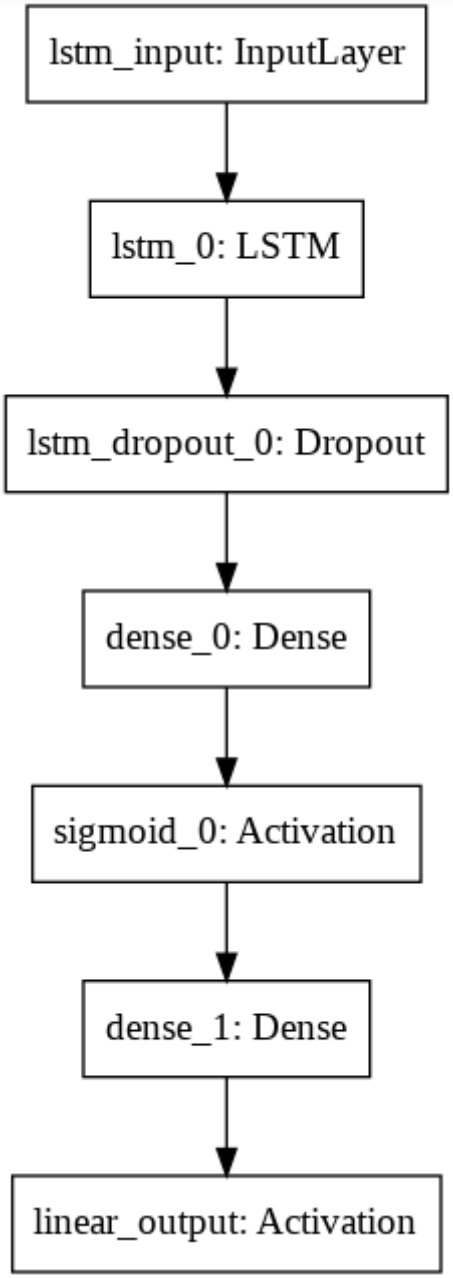
And that's it! Our dataset is ready.

The Model

I started this project only knowing how to write sequential Keras code, but I ended up learning it's functional API since I wanted a more complex network structure, eventually featuring two inputs with different layer types in each branch.

I'll go over the most basic model that I came up with first.

Which gives us a model that looks like:

Basic model architecture

The input layer has shape (history_points, 5), since each input data point is an array shaped like [*history_points* × OHLCV]. The model has 50 LSTM cells in the first layer, a dropout layer to prevent overfitting and then some dense layers to bring all of the LSTM data together.

An important feature of this network is the linear output activation, allowing the model to tune its penultimate weights accurately.

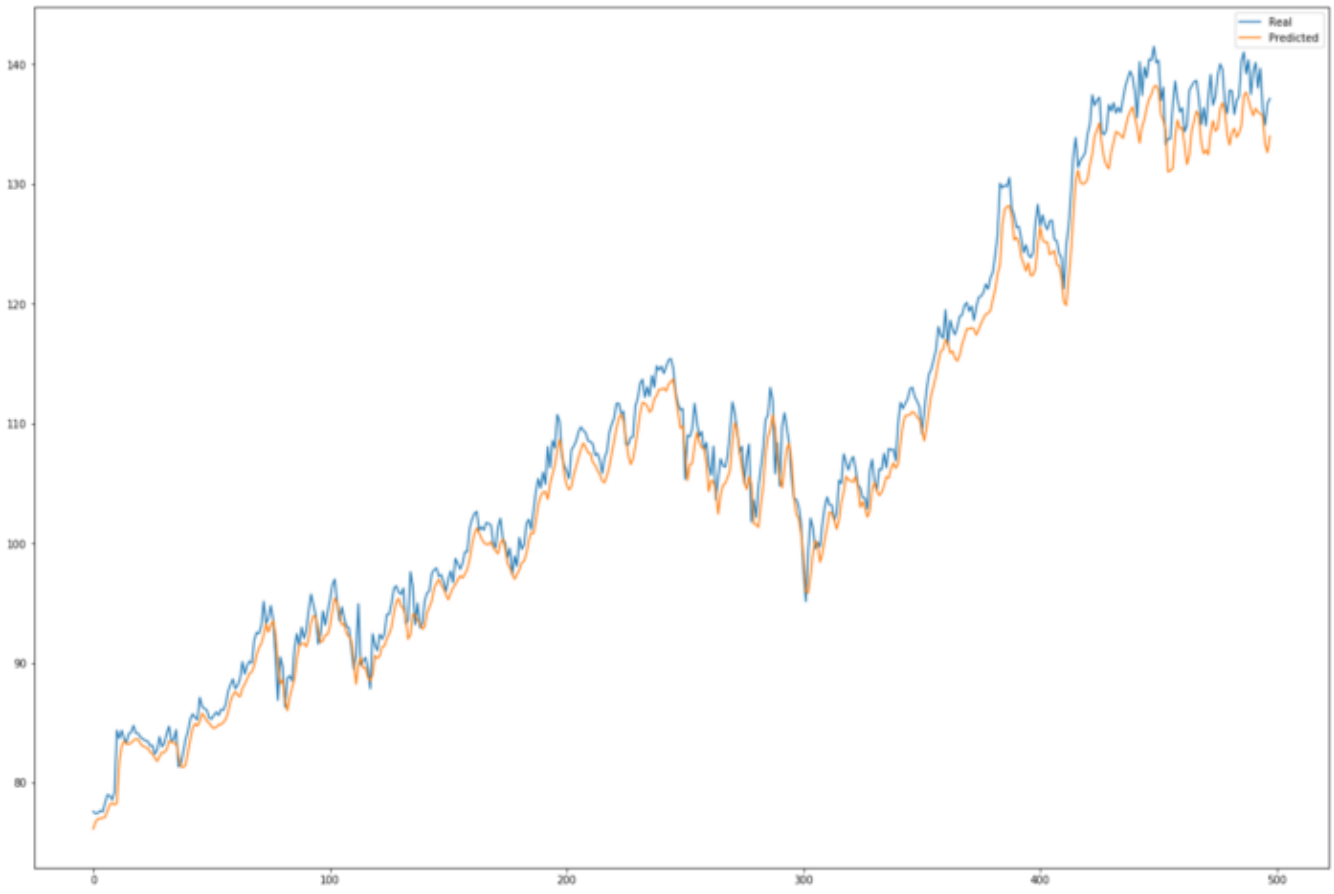
The Training

This is why I love Keras. Literally 3 lines of code and you instantly know how well your model is doing on a dataset. I got a final evaluation score of 0.00029, which seems super low but remember that this is the mean squared error of the *normalised* data. After scaling this value will go up significantly, so it's not a great metric for loss.

The Evaluation

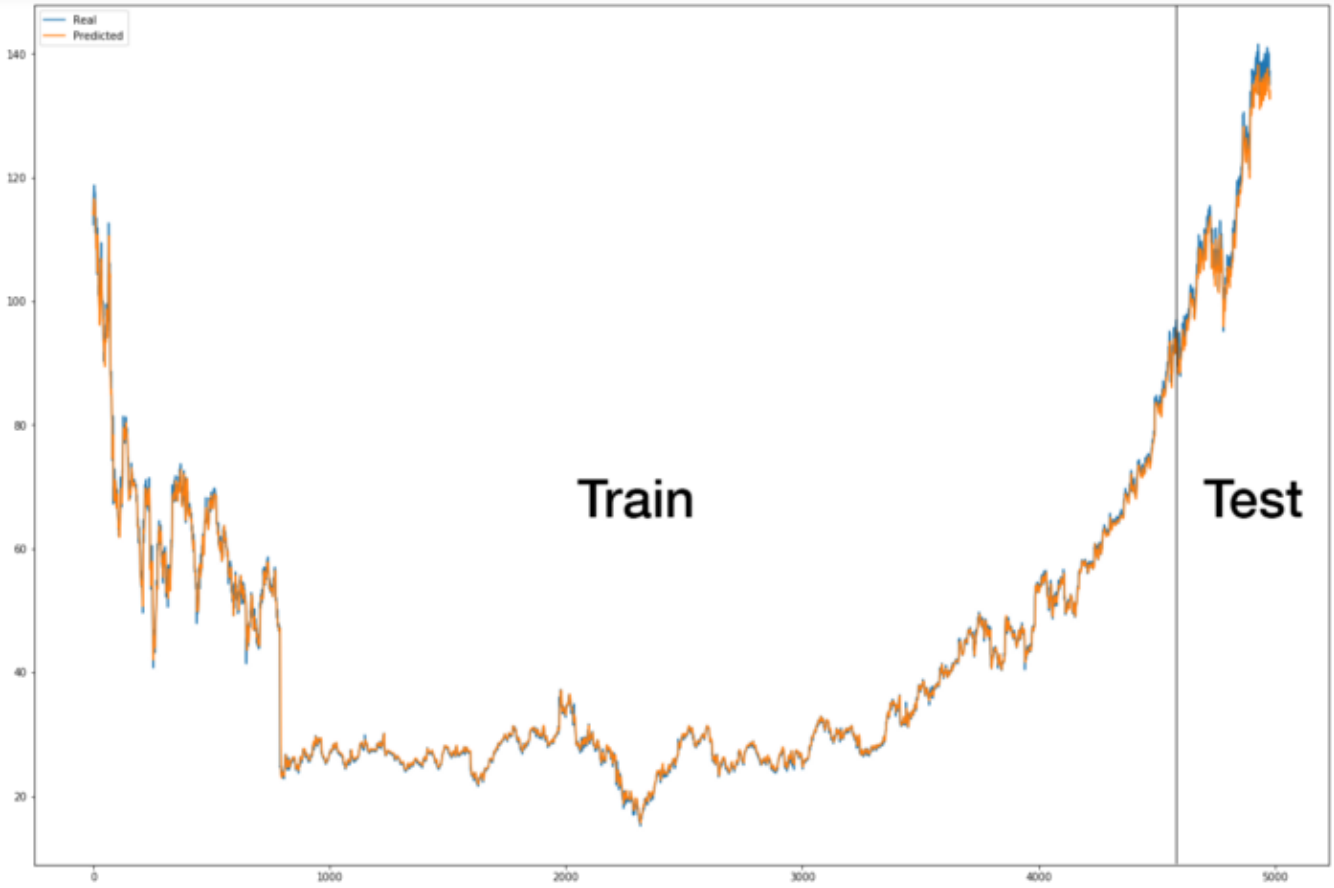
To more accurately evaluate the model, let's see how it predicts the test set in comparison with the real values. First we scale the predicted values up, then we compute the mean squared error, but then to make the error relative to the dataset we divide it by the 'spread' of the test data — the max test data point minus the min.


This gives us an adjusted mean squared error of **7.17**. Is that good? It's not amazing, it means on average the predicted line deviates over 7% from the actual. Let's see how it looks on a graph.



Real and predicted daily opening stock price of MSFT from the last 500 days, basic network

But not bad! I'm not sure why the predicted value is consistently lower than the actual value, maybe it's something to do with the way the test and train sets are split. And the



Real and predicted daily opening stock price of MSFT since 1999, basic network

It's hard to tell how well the algorithm is performing across this whole graph but you can definitely see a tighter fit across the train set, as we would expect.

The Improvements

We could try to make our model more complex, and also increase the size of the dataset. Let's start with trying to create a more complex model.

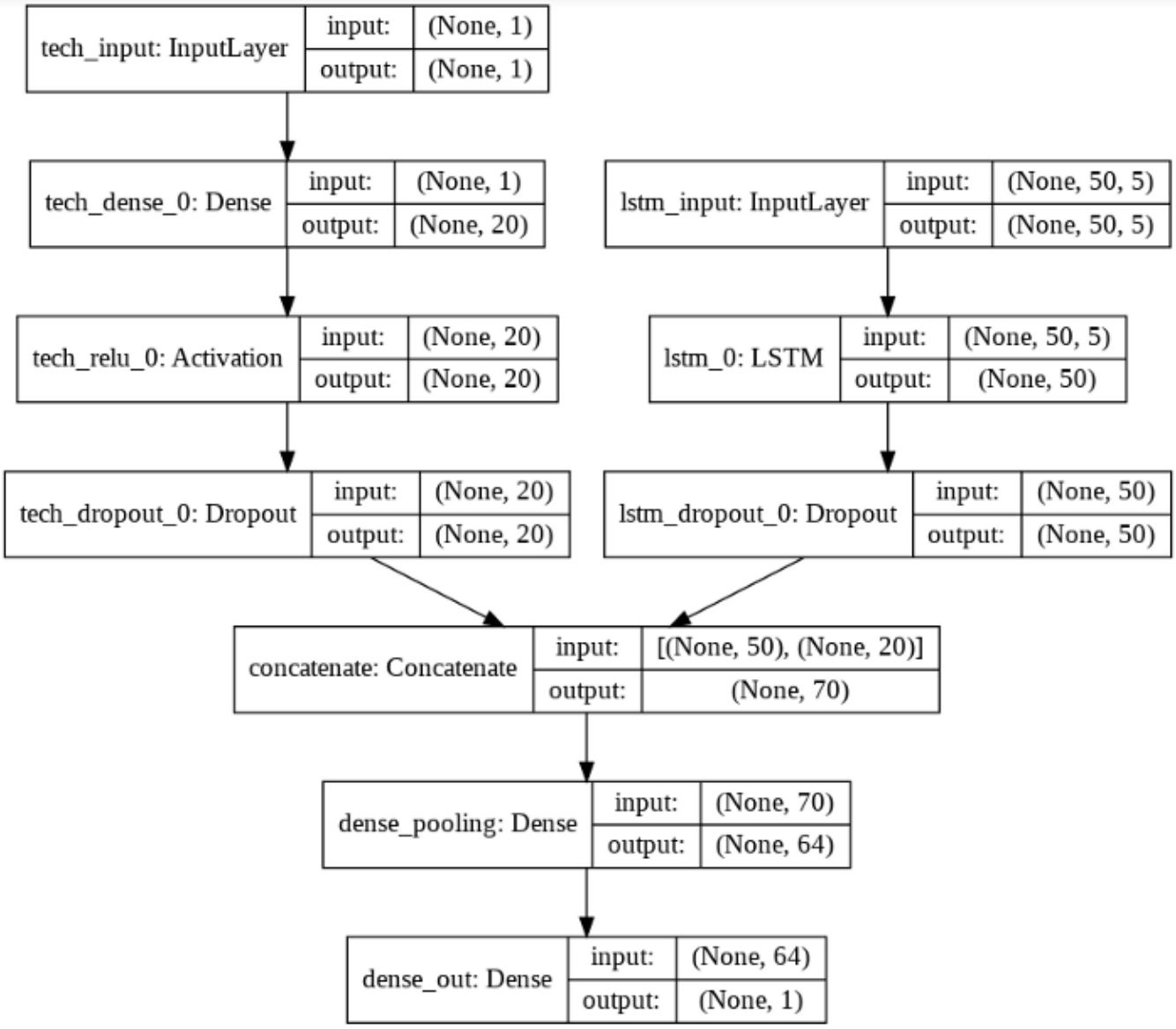
A common metric used by stock market analysts are *technical indicators[4]*. Technical indicators are math operations done on stock price history, and are traditionally used as visual aids to help identify the direction the market is going to change in. We can augment our model to accept these technical indicators through a secondary input branch.

For now let's use only the simple moving average SMA indicator as an extra input into our network.

This happens just after we have defined the ohlcv_histories and next_day_open_values arrays. We loop across every 50-price block of data and calculate the mean of the 3rd column, the closing price, and add that value to a *technical_indicators* list. The list then goes through the same transformations as the rest of the data, being scaled to fit within the values 0 to 1. We then change the return statement to return the technical indicators, as well as the other stuff we returned from before.

Now to augment the model to match this new dataset. We want to be able to use our old LSTM structure, but incorporate the SMA technical indicator somewhere in the

it into the penultimate 64-node dense layer. So we will need a model with two inputs, a concatenation layer and one output.



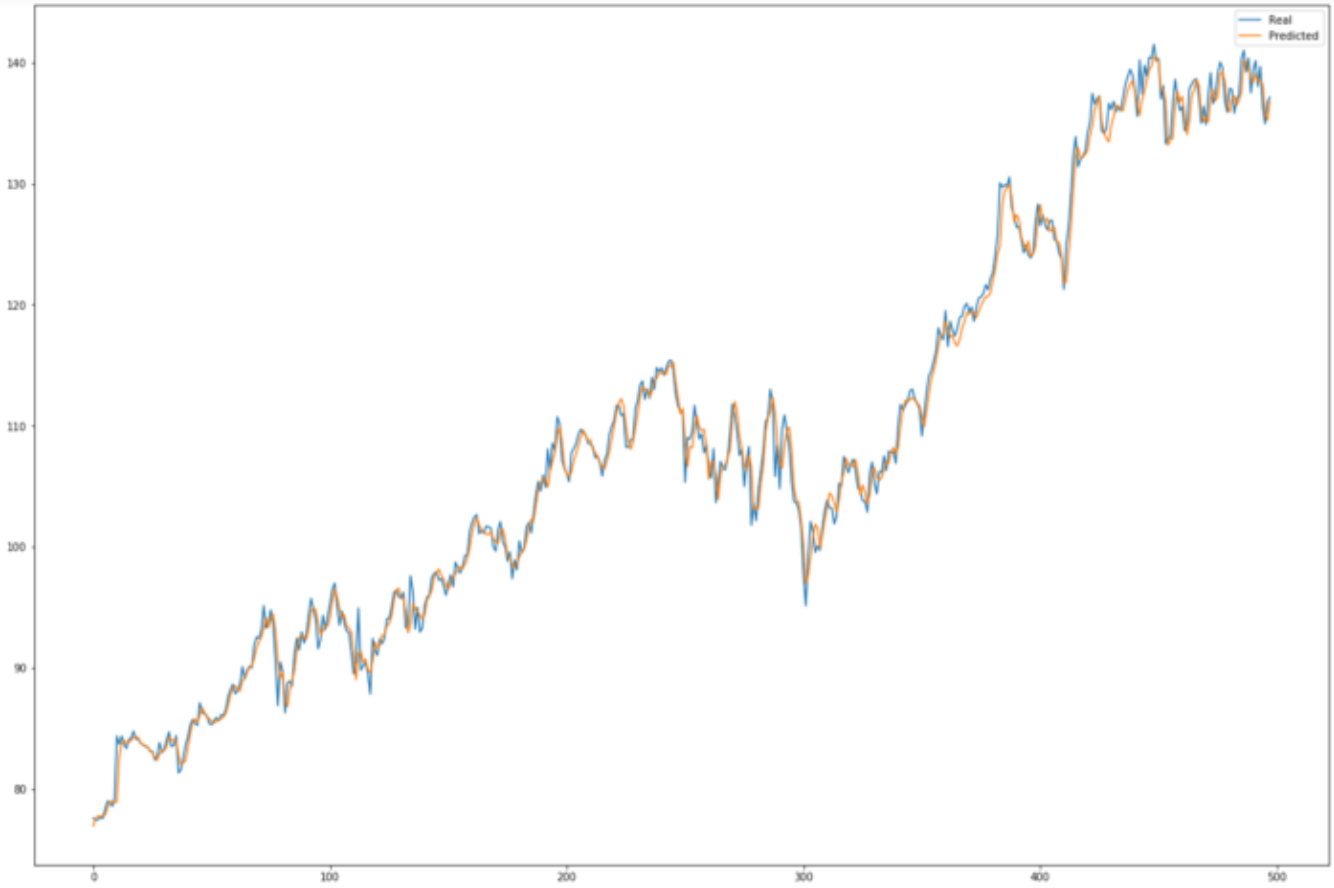
Model architecture after augmentation to use technical indicators

Note how we used technical_indicators.shape[1] as the input shape for the tech_input layer. This means that any new technical indicators we add will fit in just fine when we recompile the model.

The evaluation code has to be changed to match this dataset change as well.

We pass in a list of [ohlcv, technical_indicators] as the input to our model. This order matches the way we defined our model's input.

And we get an adjusted mean squared error of **2.25**! Much lower, and the prediction appears to fit significantly closer to the test set when plotted;



Real and predicted daily opening stock price of MSFT from the last 500 days, using SMA

This model appears to not suffer the previous problem of being continuously off by a fixed amount, but does seem to suffer from not catching sudden jumps as well. Like at around x-coordinate 120, a large jump and dip in the real price occurs but the model fails to capture this effectively. But it is getting better! And it seems that technical indicators could be the way forward.

Let's try including a more advanced technical indicator: the Moving Average Convergence Divergence. The MACD is calculated by subtracting the 26-period Exponential Moving Average from the 12-period EMA[6]. The EMA is calculated[7] using the formula:

$$EMA = \text{Price}(t) \times k + EMA(y) \times (1 - k)$$

where:

$t =$ today

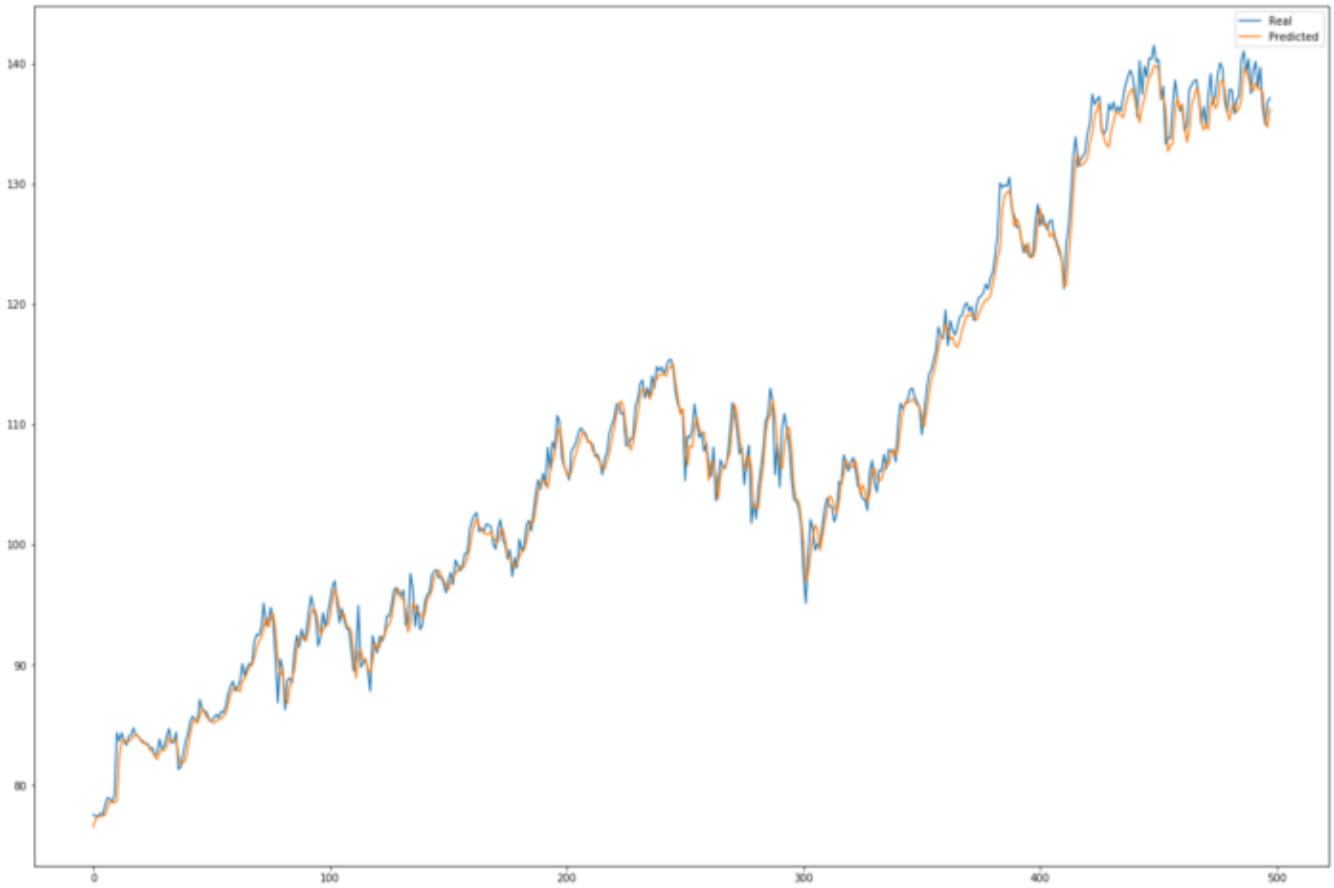
$y =$ yesterday

$N =$ number of days in EMA

$k = 2 \div (N + 1)$

To update our technical indicators loop to include the MACD indicator:

With the SMA and MACD indicators, our model achieves an adjusted mean squared error of 2.60. Slightly higher than when using just the SMA and that is reflected in the graph.



Real and predicted daily opening stock price of MSFT from the last 500 days, using SMA and MACD

You can see that it doesn't fit as accurately as when using just the SMA. Maybe since we're giving the model more data it needs more layers to make sense of it, but for now I'm just going to omit the use of the MACD technical indicator.

We can also experiment with using a larger dataset. If we assume that the techniques applied to stock prediction for Microsoft's stock can be generalised to all stocks, then we could just combine the results of the csv_to_dataset() function for lots of different stock histories. As an example, we could train on the stock histories of AMZN, FB, GOOGL, MSFT, NFLX, and test the results on the AAPL stock.

First run the save_dataset method from all the way back at the start of this post for all of the different stocks you wish to use. Then, the dataset creation method would look

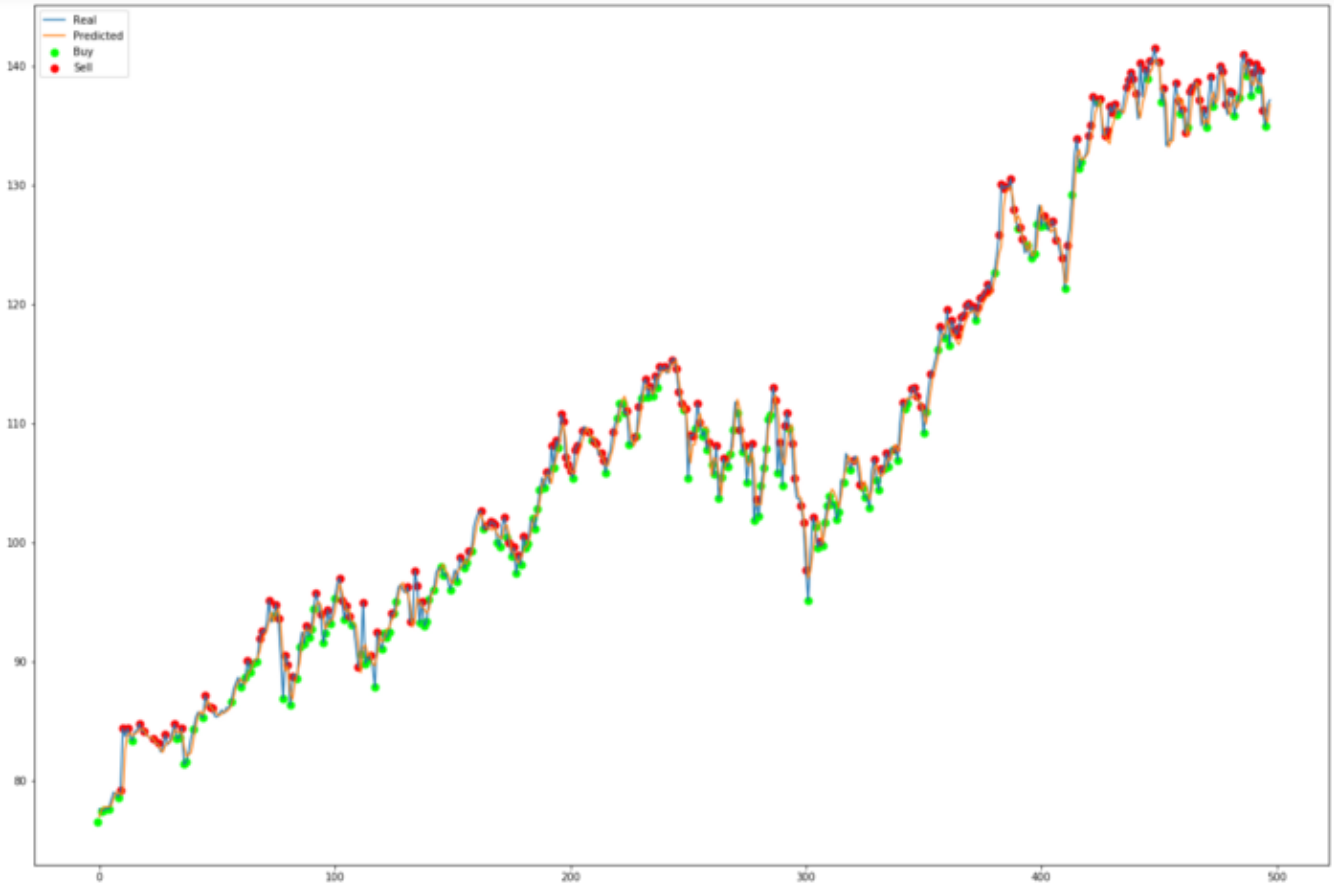
This basically says for each csv file in the current directory, if the file name is not *test_set_name*, load the dataset from it and append it to the whole dataset. Then load the *test_set_name* csv file and use that as the test dataset. Using the AMZN, NFLX, GOOGL, FB and MSFT stock prices for the train set we get 19854 train samples. Using the AAPL stock for the test set we get 4981 test samples.

After training with this new, larger dataset for 50 epochs with the SMA indicator we get an adjusted MSE value of **12.97**, higher than when we trained on just one stock. Maybe it's the case that stock prediction is actually symbol-specific; that different stocks may behave in slightly different ways. It's clear that they all fundamentally


The Algorithm

Armed with an okay-ish stock prediction algorithm I thought of a naïve way of creating a bot to decide to buy/sell a stock today given the stock's history. In essence you just predict the opening value of the stock for the next day, and if it is beyond a threshold amount you buy the stock. If it is below another threshold amount, sell the stock. This dead simple algorithm actually seemed to work quite well — visually at least.

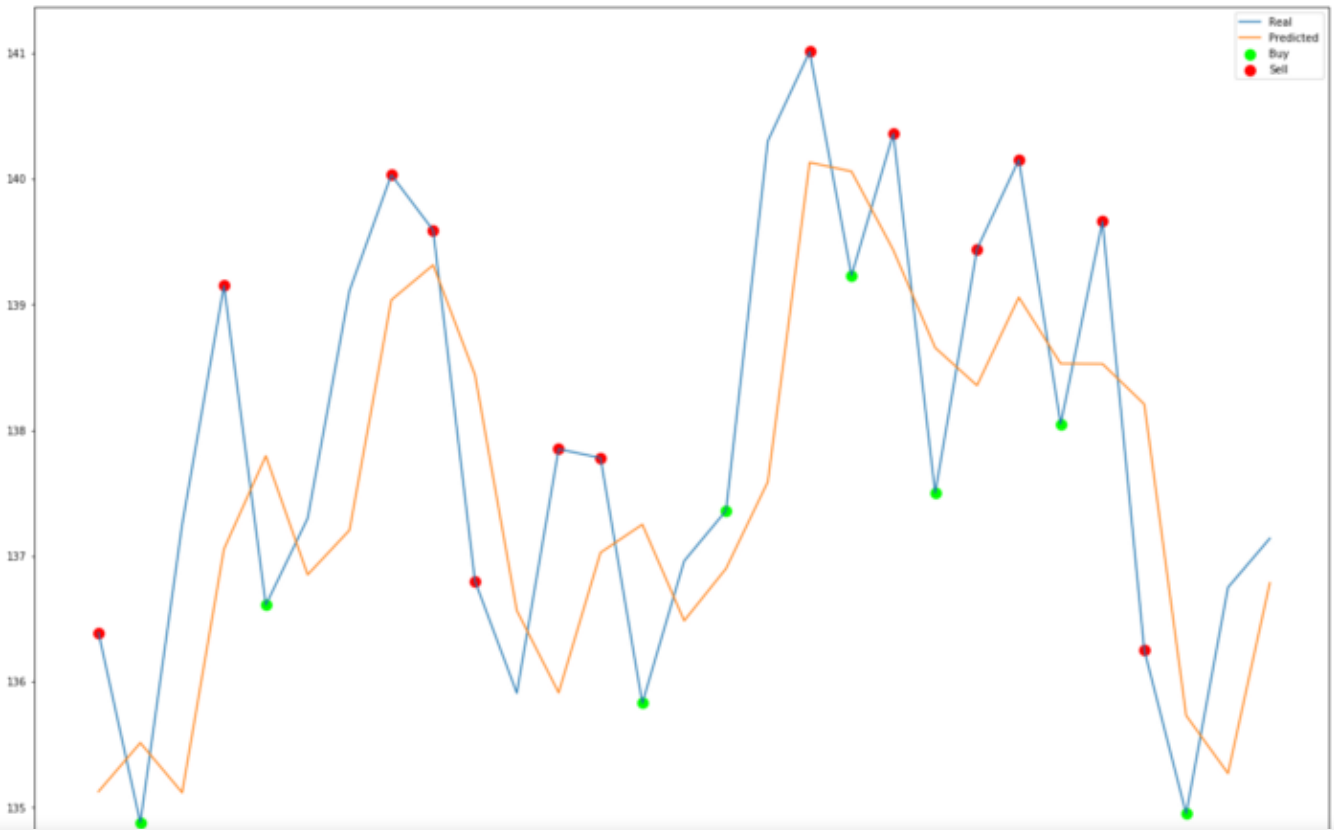
Then plot the trades.



Looking good! The algorithm appears to be correctly buying low and selling high. Remember, this is all on test data — data that the network has never seen before. There's no reason why this data couldn't have been live, and these trades actually be real!

Given these buys and sells, if we say that at each 'buy' we buy up $10 worth of the stock, and at each 'sell' we sell all of the stock, the algorithm would have earned $38.47. But bear in mind that is across 500 days. The code to calculate the earnings of the algorithm is here;

If we instead tried this algorithm over 30 days, using the same $10 purchasing amount and a threshold level of 0.2, the algorithm would have earned a measly $1.55. But that's better than nothing!



The Conclusion

I think there is still some room for improvement for the prediction algorithm. Namely, the technical indicators used, history_points hyperparameter, buy/sell algorithm/hyperparameters and model architecture are all things that I would like to optimise in the future.

I'd also like to look into giving the model more data by having more LSTM branches, one for each timestep available on AlphaVantage, so the network can make decisions based on short, medium and long term trends.

The full code for this project is available on my GitHub. Feel free to leave any feedback/improvements over on the issues page!

I do plan to expand on this project some more, to really push the limits of what can be achieved using just numerical data to predict stocks. Keep up to date with what I'm doing on my blog!

References

[1]: https://www.experfy.com/blog/the-future-of-algorithmic-trading

[2]: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[3]: https://jovianlin.io/why-is-normalization-important-in-neural-networks/

[4]: https://www.investopedia.com/terms/t/technicalindicator.asp

[5]: https://www.investopedia.com/terms/s/sma.asp

[6]: https://www.investopedia.com/terms/m/macd.asp

[7]: https://www.investopedia.com/ask/answers/122314/what-exponential-moving-average-ema-formula-and-how-ema-calculated.asp

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter