Machine Learning Engineer Nanodegree

Capstone Project

August 23rd, 2019
Justin Comino

I. Definition

Project Overview

Although predicting the stock market has long been thought to be impossible, up to 90% of trading today is done automatically according to research by JPMorgan [1]. Quant trading, as it's called, uses complicated mathematical algorithms to gain a competitive edge in buying & selling stock. IBM kick-started this type of trading in 2001 when they published their paper "High-Performance Bidding Agents for the Continuous Double Auction" [2]. Since then, stock exchanges that used to be crowded with investors slowly began to transform into server farms.

Quant trading is closely related to Artificial Intelligence, using handwritten rules to buy and sell stock automatically. With recent advances in computing capacity, however, these algorithms are in part being replaced with Machine Learning techniques. In this project, we will apply modern Machine Learning to a dataset containing stock information on about 500 companies. We can then compare our results to a more "classical" approach and see if any improvements have been made.

Problem Statement

Stock markets are volatile, changing unexpectedly due to any combination of political, social, or financial pressures. Being able to predict these changes would obviously be very valuable, which is part of what quant trading sets out to do. With ever faster computers, I believe that this field of study has a lot to benefit from Machine Learning. A neural network is able to pick up on subtle patterns and account for millions of data points in ways that a mathematical formula cannot.

For this project, I'd like to build a Machine Learning Model that can reliably predict short-term movements in the stock market. Ideally, this should perform as well or better than a "classical" algorithm. To achieve this we will be using the NYSE dataset on

Kaggle, which features 500 companies in over 800k rows of data. Here is an outline of the steps I plan to take,

- Download dataset, using the *prices-split-adjusted.csv* file
  This accounts for stock-splits in the data
- Explore data, looking for trends or abnormalities
- Pre-process data, normalize data, one-hot encode if needed
- Train benchmark model using a CNN architecture
- Train LSTM model on the same data
- Compare & visualize results

Metrics

The metric that we will use most in this project is the Mean Absolute Error. Since we are predicting the price of a stock, we need the *absolute* difference in our predictions. This should show us the difference whether we overestimate or underestimate the price. Why aren't we using something more popular like Mean Squared Error? For one, MAE is less sensitive to outliers than MSE. For example, if we overestimate a stock by $10, and again by $20, our $20 prediction is not exponentially worse than our $10 prediction. It is simply twice as incorrect. Giving more weight to outliers with something like MSE would potentially drag our prediction from following a very specific line, making the model harder to train. We would rather generally predict price than cater to outliers.

Secondly, Mean Absolute Error is more interpretable than Mean Squared Error. Since we're working directly with prices, assuming our data is not scaled, we can read the MAE as the literal price difference. If the MAE is 5.98 we know our model is off by $5.98. This makes it easier to read and understand the model.

II. Analysis

Data Exploration

Within the New York Stock Exchange zip file, there are several CSV datasets to choose from. The first one, titled "fundamentals.csv", contains financial information such as *cash reserves, cost of goods sold, earnings*, and more for the 500 listed companies. While Machine Learning would be an excellent choice for fundamental and technical analysis of this data, our focus will be on detecting patterns and predicting prices on price data alone.

With that in mind, we are using the **prices-split-adjusted.csv file**. As the name suggests, the prices within this file are adjusted for whenever a company issues more stock.

```
There are  501 stocks in our dataset
Spread over  1762 days, or 4.83  years


(851264, 7)
```

We have over **850k rows** of data, with **7 columns**.

The first two columns are the *date* and *stock ticker* which identify the company. The next 5 columns break down stock prices with *open*, *close*, *low*, *high*, and *volume*. All price columns are represented in USD. All dates are in the New York timezone.

	date	symbol	open	...	low	high	volume
216133	2011-10-27	WU	17.930000	...	17.590000	18.100000	7731000.0
316870	2012-09-05	ED	60.990002	...	60.299999	60.990002	1139200.0
660155	2015-06-25	UAA	84.830002	...	84.300003	85.290001	2892600.0
662471	2015-07-02	MA	94.489998	...	93.900002	94.760002	3034700.0
762310	2016-04-19	YHOO	36.459999	...	36.110001	36.730000	20460600.0
818347	2016-09-28	CFG	24.309999	...	23.889999	24.469999	8700300.0
334435	2012-10-26	NTRS	47.459999	...	46.820000	47.599998	949100.0
220681	2011-11-10	MYL	18.309999	...	17.959999	18.440001	4389500.0
120265	2011-01-06	YUM	35.190510	...	34.888569	35.269590	5696600.0
480473	2014-01-13	CLX	89.250000	...	88.610001	90.029999	1577200.0

In all, we have over 4 years of stock data to work with from 2010 to 2016. One thing to note is that *all* of our data is from after the 2008 Global Financial Crisis. Because of this, most stock prices contained gradually go up. The entire duration of our data is 1,762 days. For newer companies like Facebook, which went public in 2012, we only have 1,008 days of stock prices.

Input Data: Data leakage occurs when the data you use to train your model is somehow related to what you are trying to predict. For our purposes the only input value will be the *close* column, which we'll use to predict the future price of stock. Although it seems like we could use any other columns, some of them would result in data leakage. For example, the *open* price is too similar to the *close* price. If we used it to train our model, we would first appear to be making good predictions, but would simply be giving a prediction in close proximity to the value it was trained with. To

further this point, I have manually calculated the Mean Absolute Error between the *open* and *close* price of Microsoft stock.

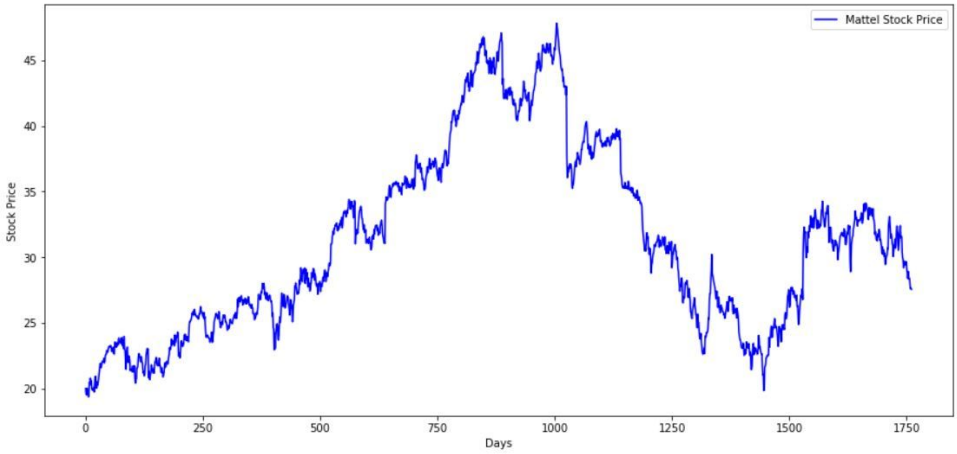
```
# calculates the Mean Absolute Error between Open and Close prices

def calculate_mae_manually(column1, column2):
  rows = len(column1)
  difference = column1 - column2
  difference = difference.abs()
  difference = difference.sum()
  difference = difference / rows
  return difference
```

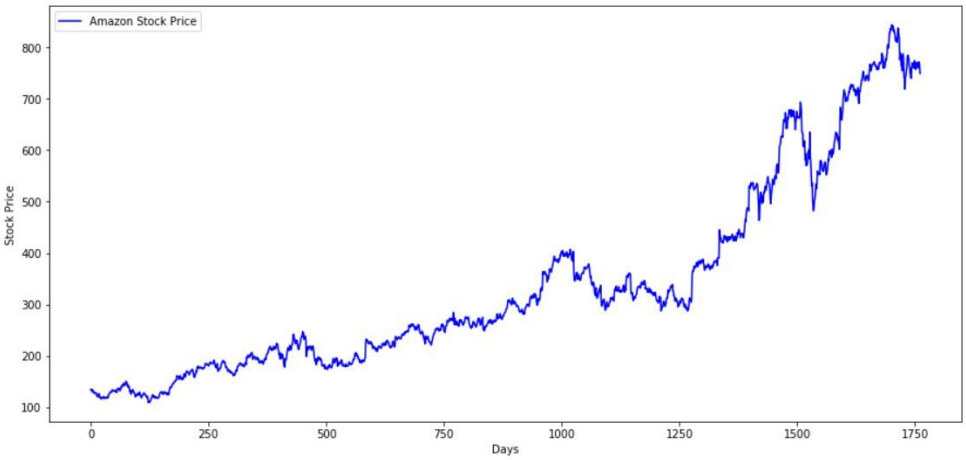
```
MAE between open and close is:  0.31
```

While 0.31 isn't a particularly good score, it is enough to tell us how similar the *open* and *close* columns are (giving us reason to exclude it in our training).

The 800k rows of data are split between about **500 companies**. The only similarity between these companies is the stock exchange they're listed on, as they come from a number of industries. When sorting by stock ticker, there is a wide variety in data between each company. For example, some companies show high seasonality. Other companies from sectors in decline may show an accompanying decline in stock value, while tech stocks grow in contrast. Following are stock prices to show this contrast,



Mattel, a retail toy maker sensitive to e-commerce



Amazon, the e-commerce competitor

The most important aspect of our data is so subtle you might have missed it!
It is that we have a dataset of **time-series data**. That is, our data is in a specific order, and this order is relevant in deciding the architecture we use. Consequently, common train / test split functions cannot be used as they will shuffle the order of our data.

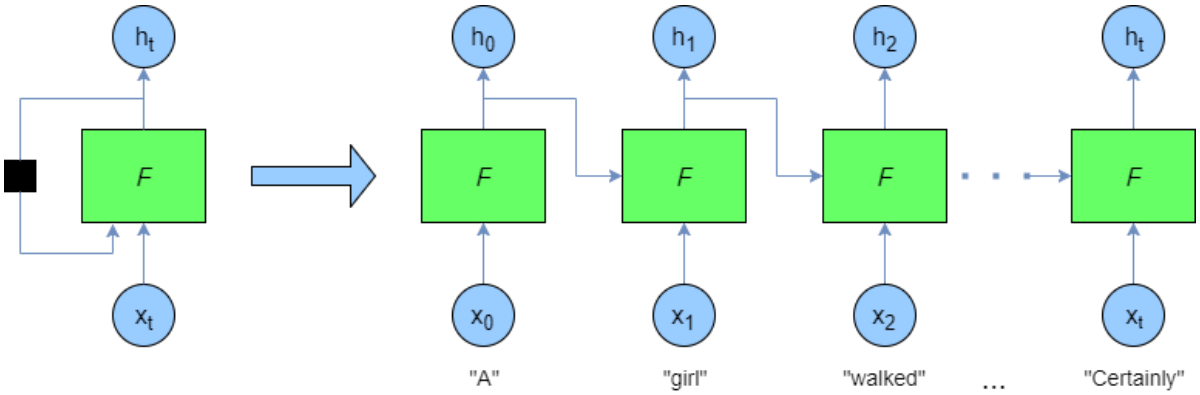
Upon closer inspection, you'll notice we are also missing stock prices for weekends and holidays. This is another caveat to the data, as stock markets are only open on business days. Using feature engineering, you could actually use this observation to create a latent feature column called "end of week" or "before holiday".

The data is said to have a daily "periodicity" because observations (recording of stock prices) are done once a day. If we were to pre-process our data and combine Monday - Friday stock prices, the data would be said to have a weekly periodicity.
LSTM

Consider a classic Neural Network. You have a specific pattern in your training data, and through various convolutions, pooling, and activations you train your model to recognize this pattern. When predicting, your model has a sort of tunnel-vision where it is only able to consider the current set of circumstances.

By comparison, our data is a series of observations that are equally spaced apart in time. This is also called a time-series. The concept of time adds another dimension to the data, and calls for a type of Neural Network that is able to take advantage of this dimensionality. Although ML can be used to learn anything, I'll show you why some architectures are better suited to the problem than others.

In a typical feedforward Neural Network, training data is sent through the various nodes only once. As it filters it's way through the model, the model updates it's weights before taking in the next training sample and repeating the process. In a Recurrent Neural Network, however, the model takes the resulting output of a node and feeds it back into the input. This loop happens as many times as is specified. In this way we have a sliding "window" that is not only considering the current example but those that came before and after it. A typical NN will only be able to recognize one pattern at a time, whereas an RNN can recognize a *set* of patterns. Here is what an RNN cell would look like "unrolled", each separate cell being a different time step.



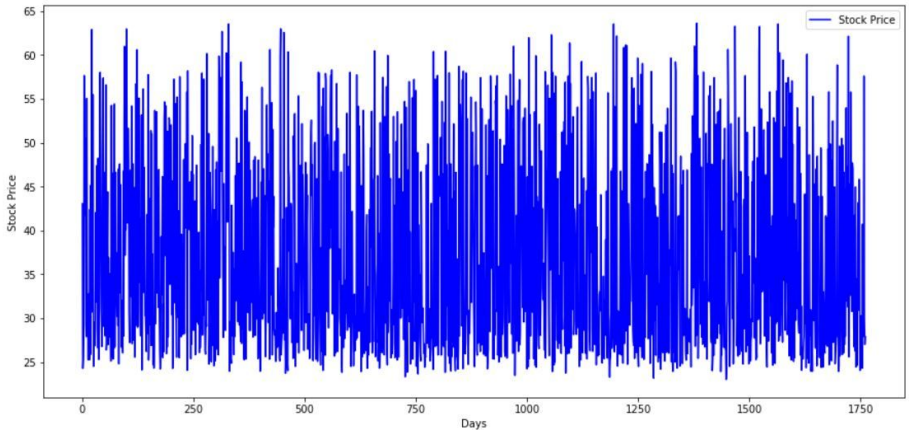
[3]

Long Short-Term Memory networks are a type of RNN that are able to handle much longer cause-and-effect relationships. They are widely used in speech recognition and translation where context is key. For stock data, an LSTM will be able to effectively recognize the patterns defining bull runs or bearish downturns in price. Time-steps will be represented in days, at 7 timesteps we can predict stocks a week in advance.

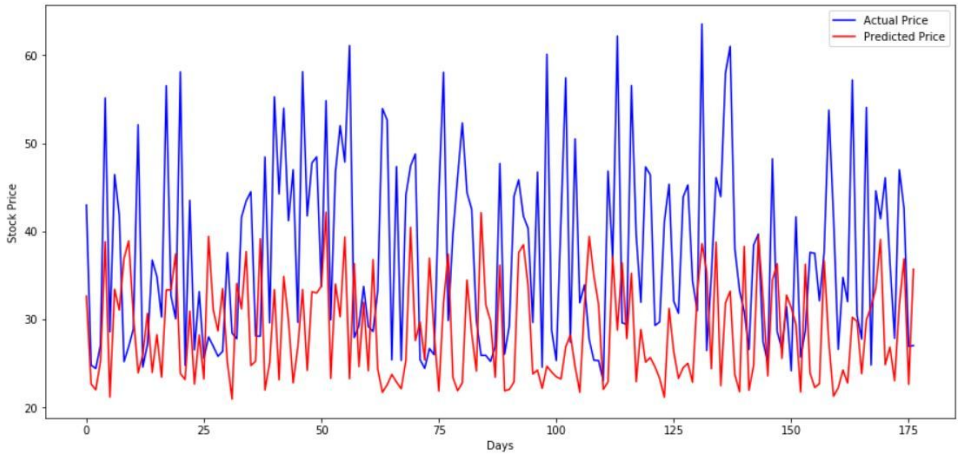
Benchmark

For our benchmark, we are going to use a Convolutional Neural Network. Both LSTM and CNNs are Neural Networks, but a CNN does not have the ability to see more than one sample at a time. This should reinforce our decision to match time-series data with a time-series capable model further on.

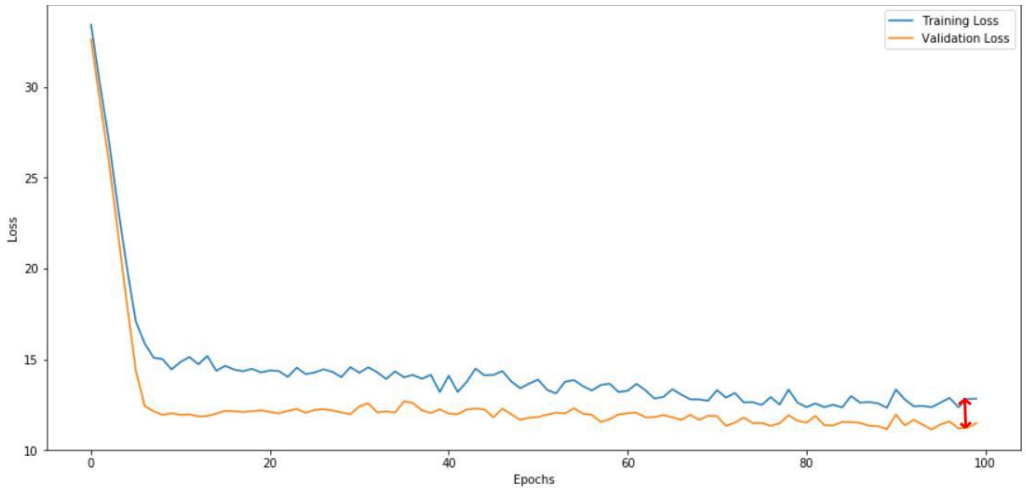
Both networks will be trained for **100 epochs**. Both will use **Adam** as the optimizer and **Mean Absolute Error** as the metric to measure loss. To visually confirm that we are removing any kind of implicit time ordering, I have used `np.random.shuffle()` to shuffle the index value of each row in place. Half of the values will be our X, and half will be our Y. Here we can see the result of this shuffle,



After 100 epochs, we can see how well our model did on a prediction of the Test set. It seems unable to fit the data very well, almost guessing at the next predicted price.



This is confirmed with our graph on Training and Validation loss. You can see how the model fails to perform well on either set, there remains a significant gap between Validation and Training. This is a classic case of underfitting.



The Mean Absolute Error on both Training and Validation sets is almost the same, with our benchmark model off by $11 to $12. This is significant for a stock that costs as little as $25 at some points.

III. Methodology

Data Preprocessing

Because we're working with time-series data, the first step in preprocessing is actually dividing it into a Train and Test split. This isn't done the usual way. Since the order the data is in will be used to predict its future value, it's important that we split the data without losing this order. For this reason we can't use something like SciKit's `train_test_split()` because it will shuffle the data. Instead we have to manually divide it. The following function splits the dataset and returns Train / Test portions while leaving the order in place.

```
def split_data(dataset):
  train_length = int(len(dataset) * 0.80)
  test_length  = len(dataset) - train_length
  train = dataset[0:train_length]
  test  = dataset[train_length:len(dataset)]
  return train, test
```

The next step is crucial when working with a time-series, and is called **lagging** the data. This is essentially rearranging the data so X can be used to predict X+1. This is easily the biggest difference between classic Machine Learning models and time sensitive models such as LSTM and RNN. When preparing our data we have no Y value, rather, we copy the X, shift those values forward, and use them in place of Y. This can be done with something as simple as Pandas `shift()`. Here is our implementation,

```
[ ]  # splits an ordered dataset into X and Y
     # eg. for a sequence of observations as the X value,
     # provides the next observation as the Y value

     def create_xy(dataset, time_span = 1):
       X, Y = [], []
       for i in range(len(dataset) - time_span - 1):
         value = dataset[i:(i + time_span), 0]
         X.append(value)
         Y.append(dataset[i + time_span, 0])
       return np.array(X), np.array(Y)

     '''
```

Notice the `time_span` argument, which can be set to any arbitrary length of time.
If each time step is a day, a `time_span` of 7 would result in sample of a week.
If each time step is a minute, a `time_span` of 60 would let you predict an hour ahead.

By performing this "rolling window" transformation on your data, you are giving context to the order that each row is in. This allows a model to recognize the pattern that a *series* of rows has, instead of one at a time. Where a typical neural network may only be able to process a photo, an LSTM can understand a video by looking at each frame. Preprocessing your time-series data in this way is crucial to building a working model.

Implementation

Here are the steps we've seen so far,

1. Split data into Train / Test while retaining the order
2. Lag the X values to create an artificial X, Y dataset

Even after splitting and lagging the data, we're not done creating our Long Short Term Memory model. In order to use an LSTM your input data must have 3 dimensions,

Samples, Time Steps, Features

In a typical 2D array each column is a feature, and each row is a time step. If you were to combine rows 1-3 into one row, then 2-4 into one row, 3-5 into one row, etc you would have 3 samples. Most of this was done for us when we lagged our data using the `create_xy()` function. The final step is to reshape the data to 3 dimensions.

```
# shape for LSTM
# VERY IMPORTANT TO DO CORRECTLY
trainX = trainX.reshape(trainX.shape[0] , 1 ,trainX.shape[1])
testX = testX.reshape(testX.shape[0] , 1 ,testX.shape[1])
```

This follows the **samples, time steps, features** format and is finally ready for implementation with our LSTM layer,

```
LSTMmodel.add(LSTM(128 , input_shape = (1 , time_span) , return_sequences=True))
```

Refinement

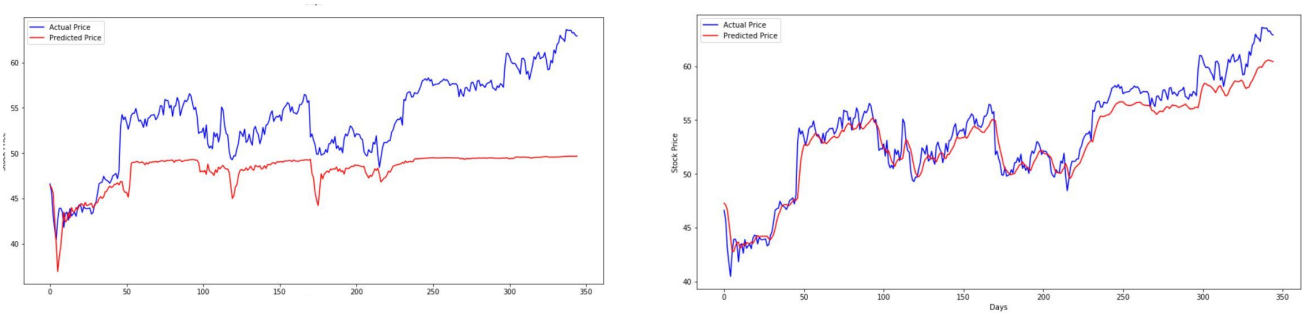
```
_________________________________________________________________
Layer (type)                 Output Shape
=================================================================
lstm_79 (LSTM)               (None, 1, 8)
_________________________________________________________________
lstm_80 (LSTM)               (None, 1)
=================================================================
Total params: 552
Trainable params: 552
Non-trainable params: 0
_________________________________________________________________
```

When refining a model, it's important to limit changes to one variable at a time. This allows you to be confident that any results gained are from the change you made as opposed to some combination of modifications. To keep things consistent, my goal was to build a capable model within **50 epochs**.

Starting with only 2 LSTM layers and less than 1000 parameters, my first model performed surprisingly well. Here were my initial results,

Time Span: 7 days
Batch Size: 64
Optimizer: Adam
Metric: MAE
Training Loss: 0.013
Validation Loss: 0.087

Up to this point I had not done any scaling in favor of better model interpretability. After using Sklearn's `MixMaxScaler()`, it was apparent how much training speed would benefit from scaling. Here is an example showing both models at **45 epochs**, the first without scaling and the second with. This is on the test set.



Notice in the first example how the model is able to predict the various movements in stock price, but struggles to match the upward trend.

This is particularly interesting because we didn't shuffle our data. Since we're working through the data in order, it makes sense that the LSTM adjusts weights for the model from left to right.

One variable LSTMs are sensitive to is batch size. Reducing the batch size from 64 to match the time span of 7 slowed down training, but lowered validation loss to **0.073**

Using `mean_squared_error` instead of `mean_absolute_error` provided the biggest improvement in refining my model. As shown in the previous figure, despite imitating stock movement the model had trouble figuring out *where* to put it's predictions. Using MSE resulted in a noticeably faster adaptation to the stocks approximate location. Intuition says that one incorrect prediction is not *exponentially* worse than any other prediction, but due to the way the LSTM fits from left to right, MSE closed the gap much faster.

One side effect of using MSE was that the model was slightly more likely to overfit. This was addressed by adding several Dense and Dropout layers to add model capability, as well as increasing the batch size to combat overfitting.

Final Solution

```
_________________________________________________________________
Layer (type)                 Output Shape
=================================================================
lstm_7 (LSTM)                (None, 1, 128)
_________________________________________________________________
dropout_3 (Dropout)          (None, 1, 128)
_________________________________________________________________
lstm_8 (LSTM)                (None, 128)
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)
_________________________________________________________________
dense_4 (Dense)              (None, 16)
_________________________________________________________________
dense_5 (Dense)              (None, 8)
_________________________________________________________________
dense_6 (Dense)              (None, 1)
=================================================================
Total params: 203,425
Trainable params: 203,425
Non-trainable params: 0
```

Time Span: 7 days
Batch Size: 64
Optimizer: Adam
Metric: MSE
Training Loss: 0.00035
Validation Loss: 0.0013

IV. Results

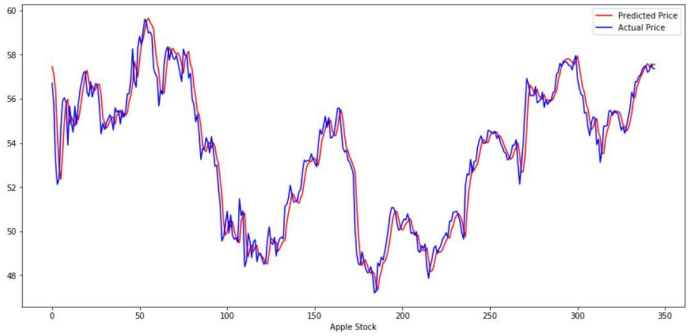
Model Evaluation and Validation

Choosing Long Short-Term Memory for use in the final model reflected well on our results. The underlying architecture of the LSTM effectively worked with time series data, helping us confidently predict future price based on prior performance.

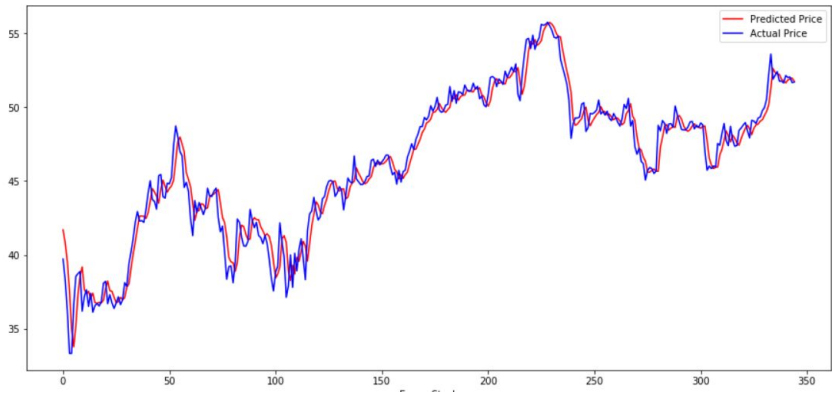
LSTMs are great at recognizing repeating patterns at a macro level. This led me to wonder,

Will the same patterns learned from one stock apply to another?

To test this, I used the same model that was trained on Microsoft stock (MSFT), to predict the price of Apple stock (AAPL) , and printed the MSE.



The result was an MSE of 0.0005. This could be because Microsoft and Apple are closely related technology companies. A trend in one may very well correlate positively or negatively with another. To rule this out, I compared the Microsoft trained model to one of the largest oil companies in the world, Exxon Mobil (XOM). Interestingly, the MSE remained at 0.0005.



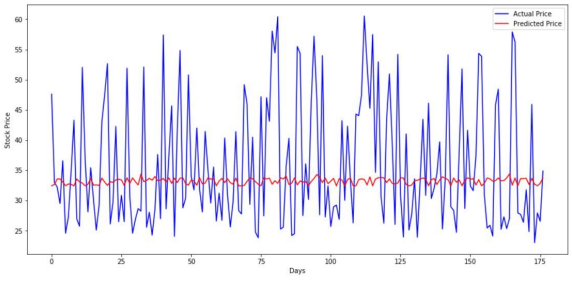
Can this model be trusted?

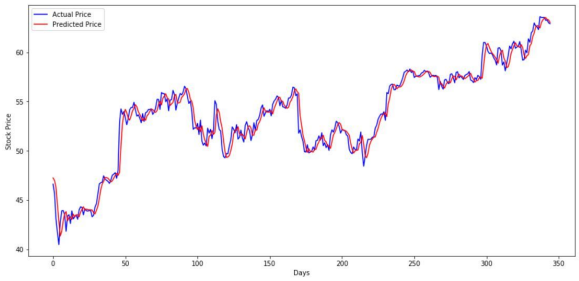
Without an understanding of the data we're analyzing, it's easy to come to the conclusion that we've built a money making machine! The truth of course, is much less exciting. According to random walk theory, stock prices are completely random and unable to be predicted. Stock prices are said to actually be a type of gaussian noise.

The reason our model performs well on both the test set and other stocks it has not seen before, is because it is predicting only 1 day at a time. We use the previous 7 days of data to predict the 8th day. This rolling window moves forward only to predict the next day. Moreover, investors are not necessarily trying to predict the price, but are trying to predict *returns.* Due to the random nature of stock, it's impossible to know the ideal place to enter & exit a stock to make a profitable return when you're only predicting one day at a time.

Justification

Although dramatic, we can see the effect that removing any kind of ordering / time information has on an otherwise capable Convolutional Neural Network. Both graphs reflect Test set results. Removing time information severely reduces the effectiveness of the model.





Benchmark MSE: 0.0669 **Final Solution MSE:** 0.0012

An LSTM or any kind of Recurrent Neural Network such as GRU proved to work well with our dataset. This is a very powerful architecture that has been responsible for many of the latest innovations in Machine Learning.

V. Conclusion

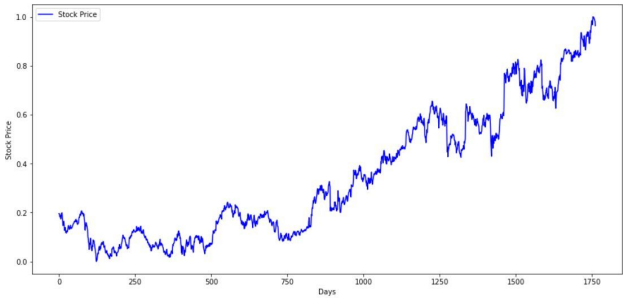
Free-Form Visualization

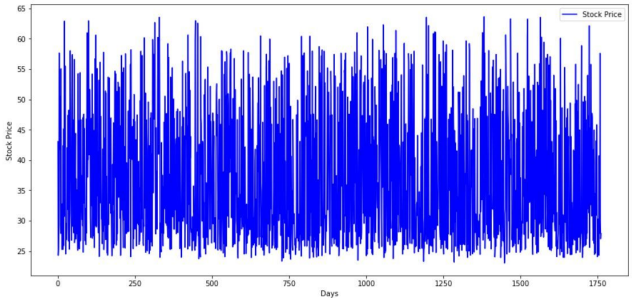
As shown prior, the LSTM model was able to roughly mimic movements in training and test data but had trouble placing those predictions on the Y axis. This is because our data is not stationary. In other words, our stock data shows a seasonal or cyclical trend. Since this dataset is post 2008, most stock contained trends upwards.

AutoCorrelation
A set of values with an identifiable trend is said to have strong autocorrelation. This literally means being “highly related to itself”. When graphing autocorrelation, we are seeing likely a variable is to be in the same place as the previous variable. For example, a stock price of $10 over 100 days would have a high correlation. There is a high chance that the next days price will be similar to the previous days price. Seasonal data can also show autocorrelation, where the price one year is very similar to the price at the same time next year.

Pearson's Correlation Coefficient is what we use to illustrate how similar these values are. It is a number between 1 and -1, with 1 being positive correlation (a repeating value) and -1 being negative correlation (the value is the opposite). A coefficient of 0 indicates no correlation (two data points are not related).

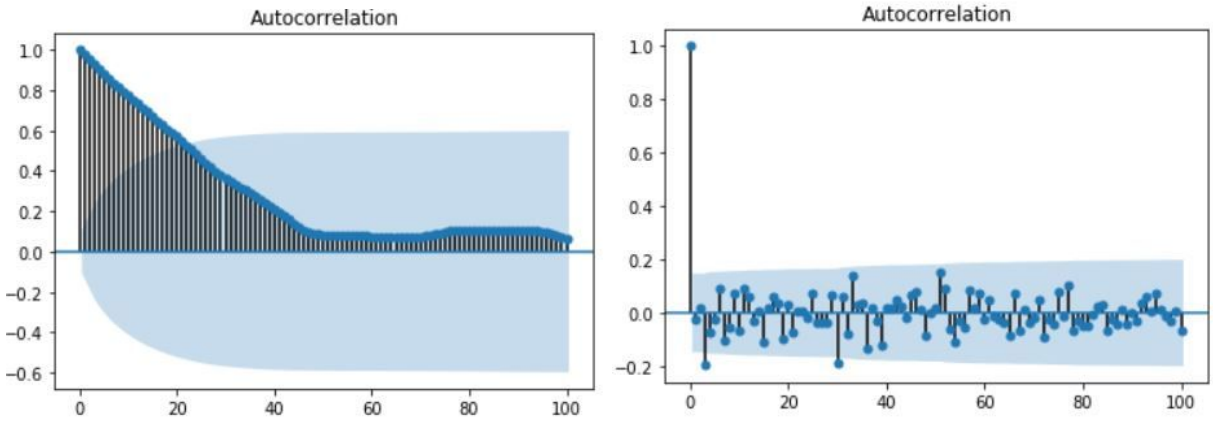
Here is the Microsoft training set, and the shuffled version as a reminder,





And here is the graph plotting the ordered Microsoft stock data’s correlation. In the first half of stock prices, there is not much movement. As a result, our autocorrelation graph shows that these values are highly correlated. As the stock trends upwards, this

correlation goes down. The blue cone is the **confidence interval** set to 95%. Anything outside of this confidence interval is highly likely to be correlated.



Compare this to the shuffled stock data we used for our benchmark. Every point is within the 95% confidence interval, showing very weak correlation with itself.
Note: The first data point always has a coefficient of 1, since it is related to itself

Reflection

Finding the best solution to this problem required a good deal of back and forth.
My first model had only 2 LSTMs and performed OK but was underfitting. I then added 3 Dense layers and switched to a MSE loss function. This would overfit at higher epochs. Finally, I added several Dropout layers at 20% that struck the right balance.

Overall, I was very impressed at how capable the LSTM model was. Although not perfect, even just a 2 layer LSTM model was able to get reasonable results. I find it incredible that what was a very complicated algorithm developed in the 90's can be implemented in Tensorflow and is still effective today.

One thing I found interesting while researching forecasting was the preference that quant traders have for using "pure" models. These are models that are trained *only* on stock price, without taking into consideration any fundamentals. Upon careful consideration, this makes sense. If stock price is truly random, adding more features would force the model to try to learn the relationship between two unrelated variables. Your model would perform the same, or more likely worse.

The final solution did well at predicting one day at a time. Forecasting further into the future is possible but would result in a less confident prediction. As such, this model would be most useful in an analysis context.

Improvement

Is it possible to use the current model as a benchmark for something better?
It depends. The model has a convincing MSE on test data and other stock. Since it is not stationary, one improvement I would make is stationarizing the data. This would likely result in a worse score as the model could not rely on the general trend. To do this, I would have to separate each row by stock ticker and stationarize the data one by one.

Another improvement that I would make is lengthening the `time_span` to a matter of years. If there was a dataset spanning 20+ years it would be easier to gauge how effective the model is in the long term. More data would have to be collected.

Conclusion

This project was a fascinating primer in exploring time series data. Much like reinforcement learning, working with time series data represents an entire field of Machine Learning that I previously was not aware of. Classical analysis of stocks has much to gain from the exciting progress being made with easier to implement and just as effective Machine Learning models.

Bibliography

[1] Cheng, E. (2017, June 14). Just 10% of trading is regular stock picking, JPMorgan estimates. Retrieved from https://www.cnbc.com/2017/06/13/death-of-the-human-investor-just-10-percent-of-trading-is-regular-stock-picking-jpmorgan-estimates.html

[2] Tesauro, G., & Das, R. (2001). High-performance bidding agents for the continuous double auction. Proceedings of the 3rd ACM conference on Electronic Commerce - EC '01. doi:10.1145/501158.501183

[3] Retrieved from https://cdn-images-1.medium.com/max/1000/0*aHo9xDRcfJ6_A8vU.png