Forecasting with AI

Calib Fenoglio

CS 461, Bradley University

Professor Adam, Byerly

Fall 2020

Forecasting with Ai

Implementing machine learning and artificial intelligence to understand time series data

Abstract:

The purpose of this individual study paper is to utilize the theory from computer science and mathematics to solve the common general problem of forecasting time series data with practical analysis. Specifically Linear Regression (LR) will be applied to stock market data along with predicting on that same data with a Recurrent Neural Network (RNN). This has many applications in general analytics, electrical engineering, and even meteorology. The RNN is a stochastic process meaning it can produce a different result on two separate runs all things being the same. The RNN is more susceptible to tuning errors and could be outperformed by a simpler statistical estimator.

Table of Contents:

Background:

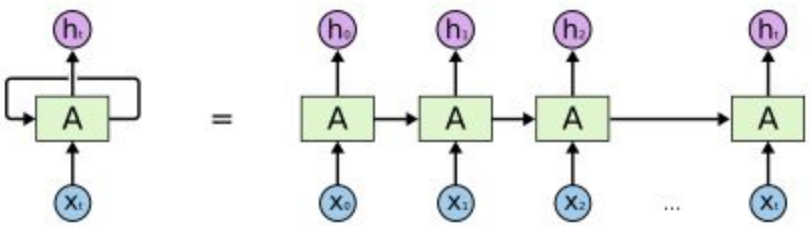
Many problems in this world can be modelled with time series, times series are inescapable. A time series is any continuous value measurement taken periodically. For example, there are applications in machine learning and AI for weather tracking, and even modelling public transit usage for dynamic allocation of resources. Speech and audio are also time series that can be modelled. Machine learning and Ai has even infiltrated the embedded tech world now that modern sensors are very sophisticated and cpu and memory on a small scale are getting increasingly cheaper to produce. Time series are also useful to model companies' stock price or the amount of visitors at a theme park. Modelling this type of data has huge implications on putting resources into the correct areas, modelling chaotic systems and providing business value. These techniques allow truly dynamic systems, but are limited in predicting too far into the future due to varying initial conditions propagating large deviations through to the end results. Some people refer to the phenomenon as the butterfly effect.

The thing about time series data is that due to the stock market we have a nearly endless amount of it to use for modelling. This data is easily accessible whether you want to download it from kaggle or another data repository site to your local machine or even a remote repository such as github. Another option is to use one of many api that can collect the data for you given a site, the stock in question and a segment of time to capture. For the purposes of this experiment the web.DataReader in pandas library for python was chosen based on familiarity with pandas and the ease of use. In addition to this an account on Tiingo was made and an api key generated to get a high quality data stream. The time series data was downloaded to google drive and uploaded to github for easy access later.

Good clean data is hard to come by and even harder to wrangle and engineer to get the desired output. Luckily the Pandas library DataReader function only needs a host site, a stock

by its ticker name with the start and end dates to pipeline stock market data right to you. For the following experiments Apple or AAPL stocks were chosen. For the data sets 5.5 years of opening and closings were chosen, this doesn't translate into 5.5 years worth of days because some days stock data is not taken.There are 7 features with ‘Open’ being our target to predict once the model has been built. To represent our test set to predict on we predicted 3 months into the future.

A huge pitfall of traditional neural networks is that they don't allow information to persist through instances. This means they cannot base new knowledge on previous knowledge. RNN’s work by predicting some vector at some time series step, they connect previous information by looping, one time step is fed to the next timestep.



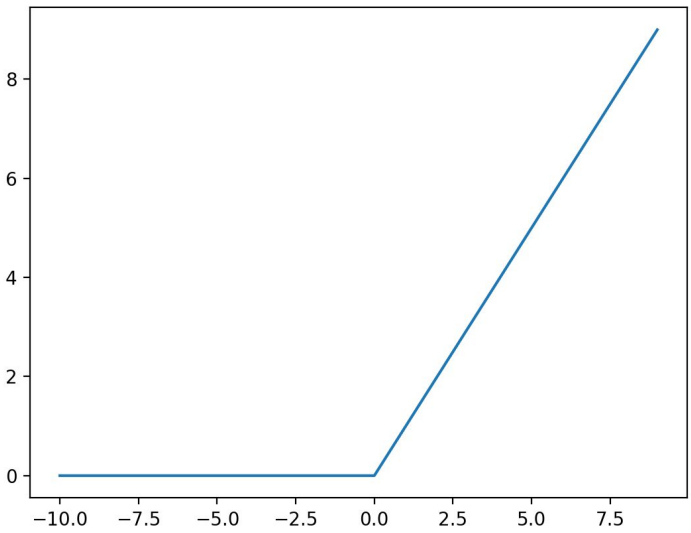
An unrolled recurrent neural network.

[6]

RNN are efficient at predicting next items in a sequence given a context. The further a context occurs from the sequence prediction the less useful that past information gain. This problem is solved by LSTM or long short term memory networks. LSTM are able to learn long term dependencies and can solve real world applications not just “toy problems”. An LSTM has a sequence architecture when unrolled it can be seen as a very deep feed forward network and they are used to improve the flow of gradient descent. The true power in LSTM is that it combats the vanishing gradient problem found in vanilla NN implementations as well as RNN and CNN models. LSTM networks are also more robust to input noise.

The Vanishing gradient Problem:

To summarize the vanishing gradient problem “The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value” https://en.wikipedia.org/wiki/Vanishing_gradient_problem. In the worst case, this may completely stop the neural network from further training. As one example of causing the problem, traditional activation functions such as the hyperbolic tangent function have gradients in the range (0, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying *n* of these small numbers to compute gradients of the "front" layers in an *n*-layer network, meaning that the gradient (error signal) decreases exponentially with *n* while the front layers train very slowly.



- graph depicting the squeezing at negative values[1]

Dropout:

Dropout is a technique that deactivates random neurons in the neural network to avoid the problem of overfitting the data. Below, in conclusions, is an accuracy of the model evaluation with and without dropout turned on. This is a simple tool, easy to implement, but

not always the correct approach. In some instances dropout can cause a decrease in accuracy but generally it just prevents overfitting and heavy reliance on any single node in the network.

Dense:

The output of an LSTM is not a softmax like RNN. The dimensionality of the output of an LSTM will generally not be the target value the experiment is trying to find. To get that target value a rotation/scaling/translation is applied in the form of a dense layer to change the dimensions of the output.

Mean Square Error :

The mean square error is a non negative number signifying the square error loss averaged across all the instances in the model. The mean square error in an estimator accounts for both the variance and the bias in the model.

The MSE of an estimator $\hat{\theta}$ with respect to an unknown parameter θ is defined as[2]

$$\mathrm{MSE}(\hat{\theta}) = \mathrm{E}_\theta\left[(\hat{\theta} - \theta)^2\right].$$

“The MSE can actually be written another way that is more indicative of the way a human perceives it. This also has the implication that when bias is equal to 0 MSE is equal to the variance.” [3]

$$\mathrm{MSE}(\hat{\theta}) = \mathrm{Var}_\theta(\hat{\theta}) + \mathrm{Bias}(\hat{\theta}, \theta)^2.$$

Section one main results, experiments, optimizations:

To build a solution to the time series problem first the data needs to be cleansed, scaled, and a train test split done on the data. This was accomplished with Python Pandas and Sklearn preprocessing libraries. The dataset is read into memory first, then described with pandas to get a feel for the dimensionality and create an intuition for the resulting steps. The data is then

scaled between 0 - 1 to produce a better fit that isnt weighted towards some arbitrary high or low pattern. At this point sklearn train_test_split is called to divide the testing and training instances.

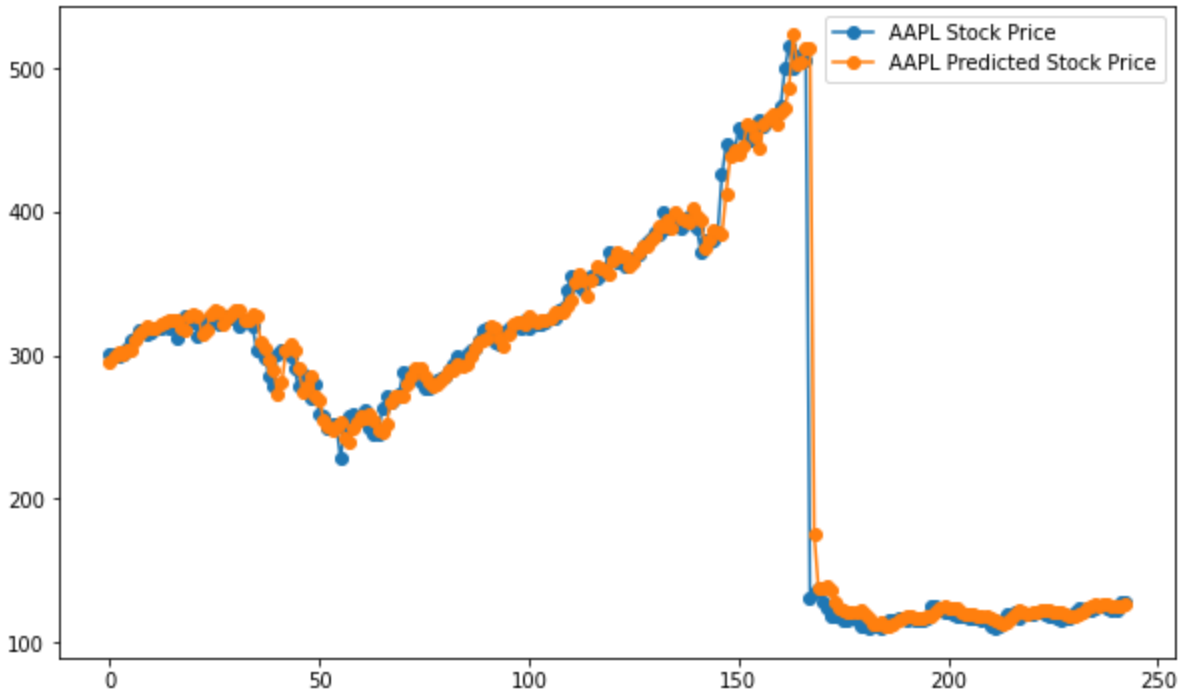
To model our stock market data a model was built utilizing Keras api. The experiment is using a sequential model since it fits the use case perfectly of each plain stack of layers having exactly one input and output tensor. The sequential model at the layer level includes 3 LSTM layers and a dense layer. The LSTM input layer is initialized to the input shape of our training data steps defined earlier in the preprocessing. The model is compiled with the ‘adam’ optimizer utilizing the mean_square_error loss function. The model goes through 100 epochs with a batch_size of 37.

```
regressor.summary()

Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_32 (LSTM)               (None, 60, 100)           40800
_________________________________________________________________
lstm_33 (LSTM)               (None, 60, 100)           80400
_________________________________________________________________
lstm_34 (LSTM)               (None, 100)               80400
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 101
=================================================================
Total params: 201,701
Trainable params: 201,701
Non-trainable params: 0
_________________________________________________________________
```

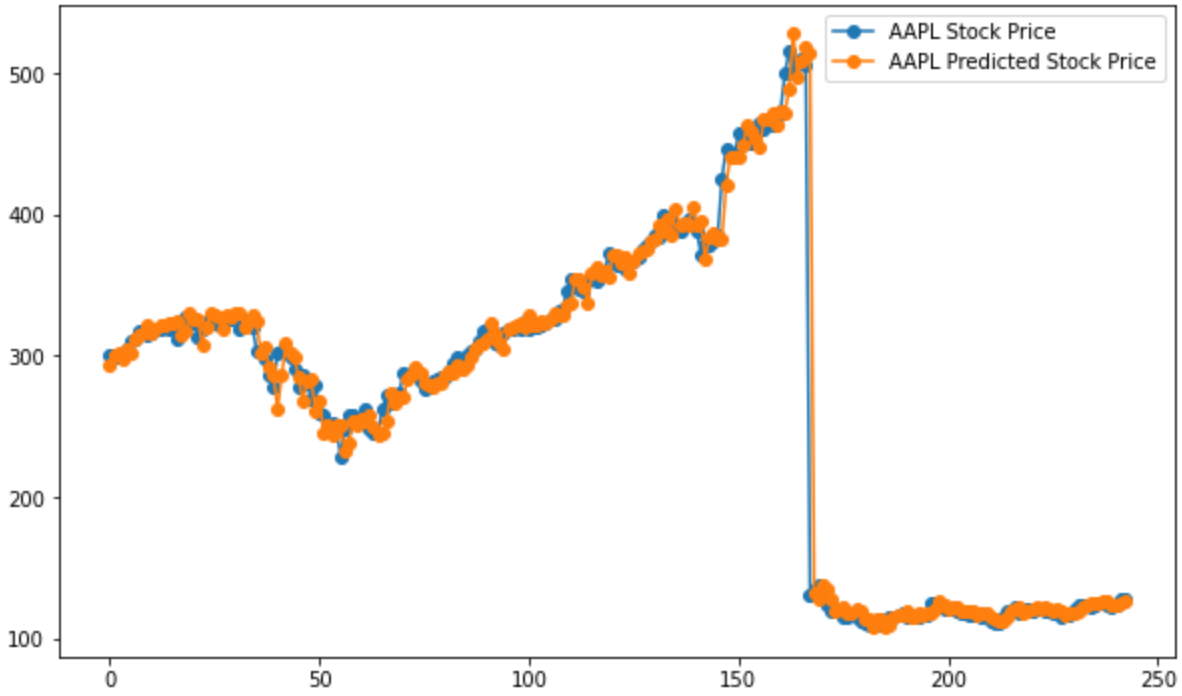
The first set of experiments is done with the LSTM model with target ‘Open’ and feature ‘High’. Since the dataset is numeric and classifications are not a binary answer but rather some deviation from the actual value of the stock price the Mean square error and root mean error were the metrics used to determine the average error you could expect for any one prediction.

This also means that the loss of this type of model also corresponds to the mean squared error value. The model then received unseen data occurring over the next three months to predict the stock price over this period of time. The output was graphed and the model statistics were condensed into a table. Below is the base these experiments started from. The X-axis is time and the Y-axis is dollar value.

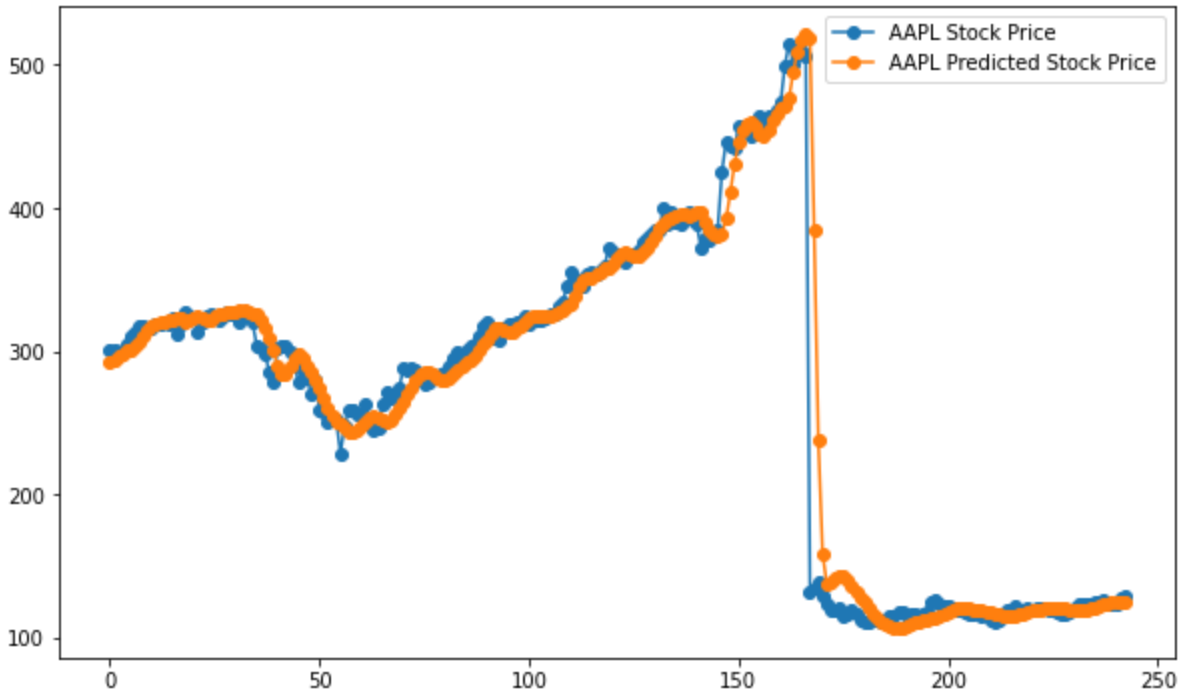


Model	Values
Units per LSTM cell	50
Dropout layers	False
Epochs	100
Batch Size	32
Accuracy (MSE)	0.00045851219329051673
Residual (RMSE)	0.02141289782562175

Next, batch size was varied to see the effects. The batch is related to the input size and epochs. Sizes that could divide the input size to the model were chosen. Input size is 2^6 X 37. At this low of a batch size the MSE was jumpy but, on the curve we can see the model predicted the drop in price better than the previous model.

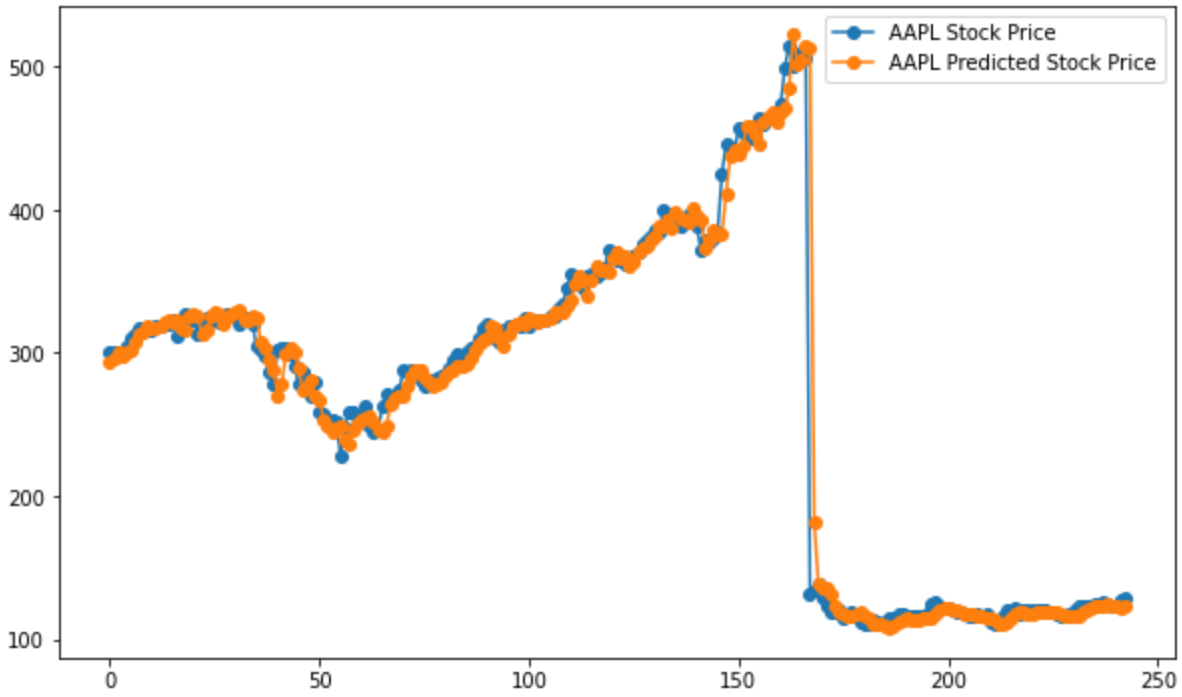


Model	Values
Units per LSTM cell	50
Dropout layers	False
Epochs	100
Batch Size	16
Accuracy (MSE)	0.0004528759454842657
Residual (RMSE)	0.021280882159446908

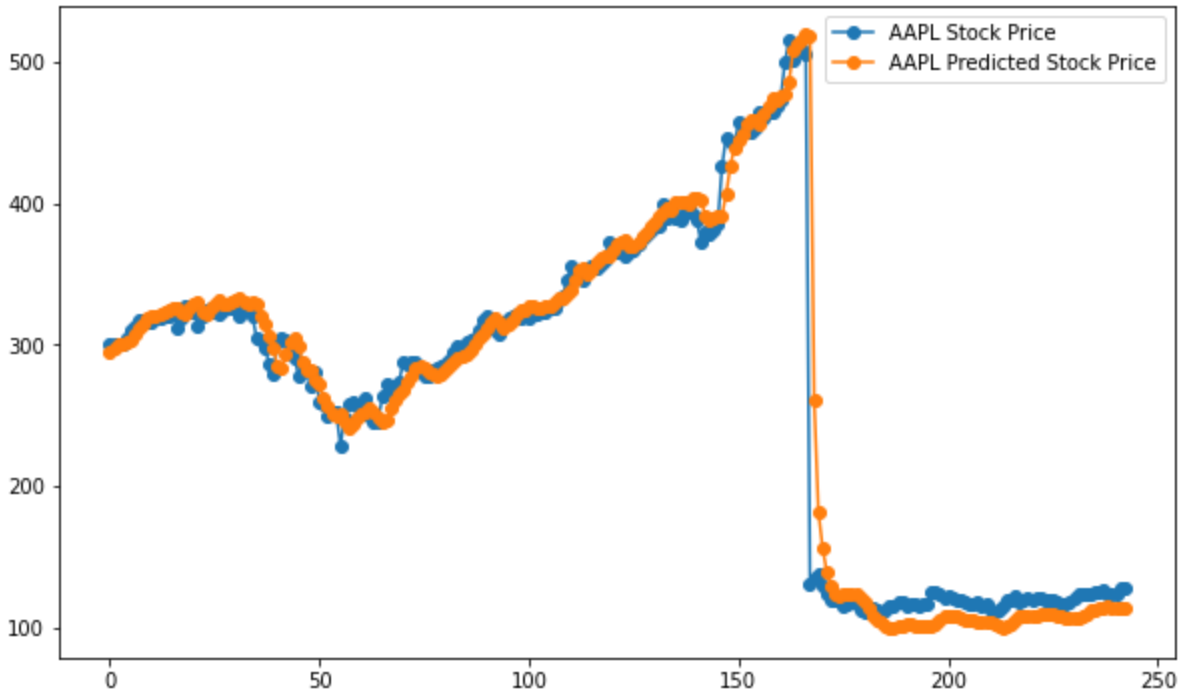


Model	Values
Units per LSTM cell	50
Dropout layers	False
Epochs	100
Batch Size	296
Accuracy (MSE)	0.0006152091664262116
Residual (RMSE)	0.024803410378942077

The Batch was varied up to 296 where it was obvious the larger batches had diminishing returns. The batch size was set back to 37 after testing 32 and 37 back to back 5 times. Batch size 37 won the tournament style match up with Mean Square Error (MSE): mse of 0.00045300083002075553.



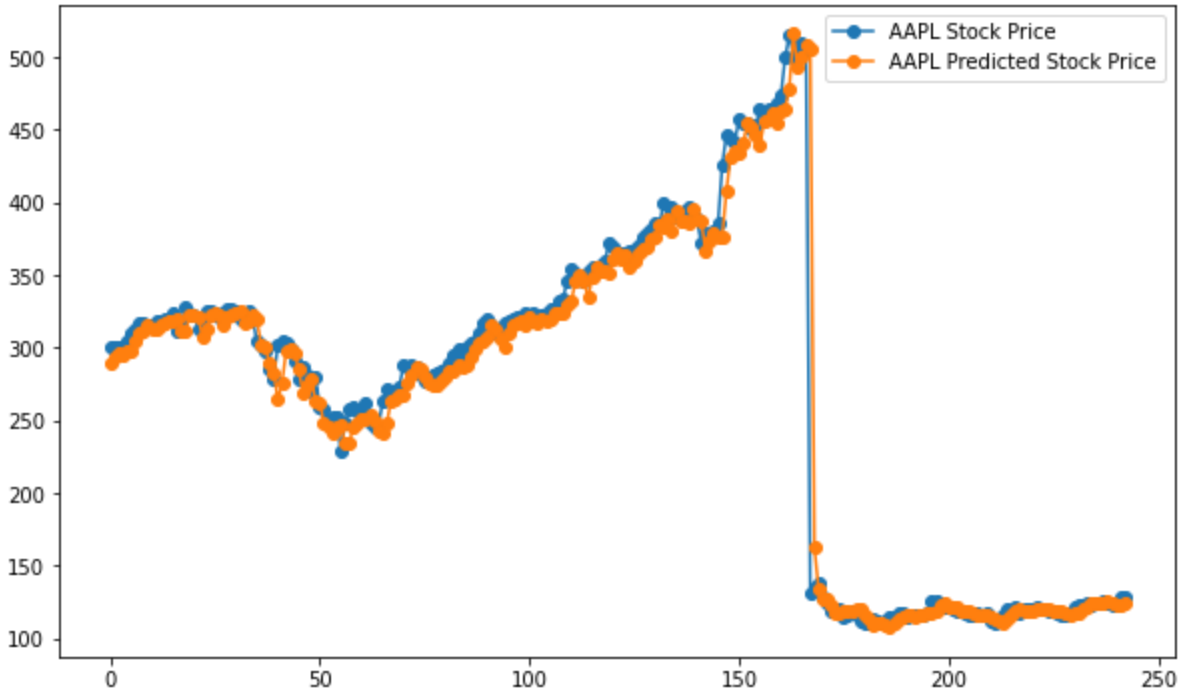
Model	Values
Units per LSTM cell	50
Dropout layers	False
Epochs	100
Batch Size	37
Accuracy (MSE)	0.00043924423516727984
Residual (RMSE)	0.020958154383611163



Model	Values
Units per LSTM cell	50
Dropout layers	True
Epochs	100
Batch Size	37
Accuracy (MSE)	0.0007636433001607656
Residual (RMSE)	0.02763409669521994

Dropout seemed to introduce error into our model. This could be from the relatively low number of instances in the dataset. Based on that last run dropout will continue to stay off for the final run where the units in each LSTM cell are actually doubled. The thought is the

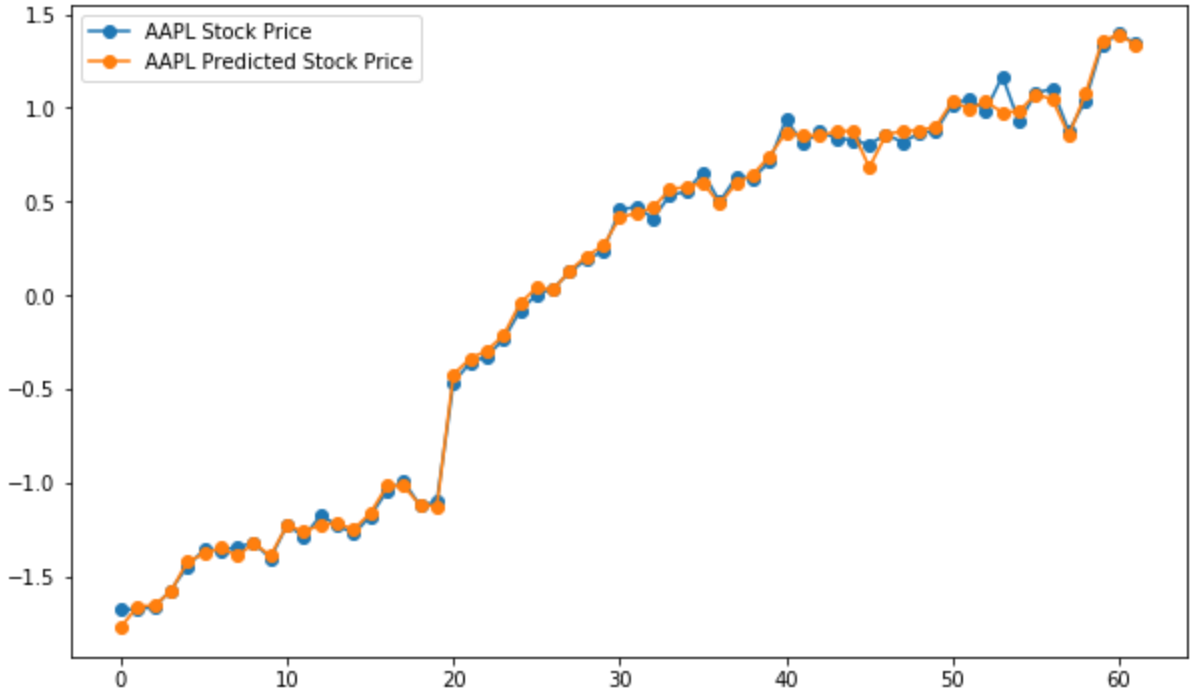
potentially more links could mean stronger patterns emerge from the data and it builds a stronger model.



Model	Values
Units per LSTM cell	100
Dropout layers	False
Epochs	100
Batch Size	37
Accuracy (MSE)	0.00040398299461230636
Residual (RMSE)	0.020099328212960413

In addition to the Neural network a statistical estimator was also deployed on this data to see if it could compete with a more advanced classifier. Linear regression actually out performed the neural network in accuracy. Below are the results as preprocessing and feature selection. It can

be seen the root mean square error is the standard error deviation and signifies the data around the line of best fit to be tightly grouped together. The Most staggering differences are in the curve of the predicted data compared to the actual data.



Mean Absolute Error:	0.019396214538422118
Mean Squared Error:	0.0006165421366255725
Root Mean Square Error:	0.024830266543586933

Section 2 limitations and future work:

Overall the model performed well for being a training experience being built on univariate input. The first step to improve the model would be to use multivariate input along with feature engineering. The next step once introducing more data and the noise that will come with it will

be to introduce dropout to reduce the chances of overfitting. The data has a relationship between its features that's more complex than just “Low” or “High” values of the market day to day. Fitting the model on a well picked subset of the entire dataset would surely improve the predictions made by the model.

Additional improvements to the dataset could also include feature engineering where we intentionally train a model on “bull markets” meaning feed the model stock upswings where the market is at > 20% increase from its previous low value. Such a feature might be able to more accurately time positive market behaviors. As for models with larger datasets dropout layers can be introduced. Drop out layers were defined because they allow nodes in the network to be turned off to reduce an over reliance on extraneous patterns. One more suggestion is to introduce confidence interval estimations on the models accuracy.

Lastly, Compute power is perhaps the largest barrier to training high performance neural networks on large datasets. Big data is a prerequisite for a high performance neural network and even with tools such as google collaboratory gpu’s to speed up training it takes an order of magnitude more time to run and debug than a traditional computer program. Many machine learning algorithms can achieve acceptable accuracies with smaller datasets.

Conclusion:

Utilizing long short term memory RNN to predict future stock pricing is in fact a very good way to predict and add value to your own portfolio or a business.That being said more advanced techniques need to be employed to realize the true potential over a statistical estimator. The neural network did very well given a relatively small amount of data and no advanced knowledge extraction techniques such as feature engineering with multivariate. This is meant to be an introduction to RNN and the model parameters are likely not optimal as well. The deviation between the errors showed both models performed well and in some cases are accurate enough to take advantage Call options. A call option is where a price is agreed upon

up to a certain date for a stock and if the call was placed correctly a small portion of value < 1% of the stock price can be wagered for a much more substantial reward. Based on this the RNN and the Linear Regression are both successful at predicting stock prices and warrant further refinement.

Appendix A:

Model Architecture

```
history =                   {}
scores =                    []
error_rate =                []
mean_error =                0

# Number of hidden units inside of hidden cells in hidden layer
# Can affect the dimensionality of the units computed and run time
n = 100
regressor = Sequential()

# Return sequences for the first two layers to return time step info
regressor.add(LSTM(units=n, return_sequences=True, input_shape =
(X_train.shape[1],1)))
regressor.add(LSTM(units=n, return_sequences=True))
regressor.add(LSTM(units=n))

# Single dense layer to transform our 50 dimension output from the last
step dim n -> 1
regressor.add(Dense(units=1))

# One Epoch is when an ENTIRE dataset is passed forward and backward
through the neural network only ONCE
history = regressor.compile(optimizer='adam',loss='mean_squared_error',
metrics=['mse'])
regressor.fit(X_train,y_train,epochs=100, batch_size=37, verbose=True);

pred = regressor.predict(X_test)
error_rate.append((y_test - pred))
total_error = np.mean(error_rate)
scores = regressor.evaluate(X_train, y_train, verbose=0, steps=60)

## MSE account for variance and bias
## RMSE accounts for the variance from error --> error or the residual
print('-----------------------------------------------------------------
-----------------------')
```

```
print(f'> Mean Square Error (MSE): {regressor.metrics_names[1]}
print(f'> Standard Error (RSME): {scores[0]**0.5}')
print('-------------------------------------------------------------------
------------------------')
```

References

1. Learning Long Term Dependencies with Gradient Descent is Difficult, IEEE Transactions on neural networks vol 5, March 1994, http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf Accessed on 12th November 2020.
2. Working With RNNS, https://keras.io/guides/working_with_rnns/ Accessed on 12th November 2020.
3. Tensorflow api docs model evaluate https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate Accessed on 12th November 2020.
4. The vanishing gradient problem, https://en.wikipedia.org/wiki/Vanishing_gradient_problemAccessed on 12th November 2020.
5. Mean Squared Error https://en.wikipedia.org/wiki/Mean_squared_error Accessed on 12th November 2020.
6. Understand LSTM Networks, Stanford Archive, https://web.stanford.edu/class/cs379c/archive/2018/class_messages_listing/content/Artificial_Neural_Network_Technology_Tutorials/OlahLSTM-NEURAL-NETWORK-TUTORIAL-15.pdf Accessed on 12th November 2020.