On Using LSTMs for stock price forecasting

Vegen Soopramanien [1]

Abstract

The prediction of stock prices is an area that many econometricians and analysts devote a lot of energy to solving, over the years coming up with novel statistical models such as ARIMA, GARCH, amongst others. These statistical time-series models have previously demonstrated great accuracy. The question that I am addressing in this paper is whether future data in the financial sector can be accurately forecast based solely on past data from the same series using techniques from deep learning. While there have been attempts to develop deep learning models in the form of Multi-Layer Perceptrons for this purpose, this paper specifically focuses on the use of LSTMs, an extension of Recurrent Neural Networks, and demonstrates how Machine Learning practitioners in the industry can use random search as opposed to grid search to determine the number of neurons that would optimize the accuracy of the output predictions. My experiments have demonstrated that given a specified simple network consisting of 2 LSTM layers, 384 units within both LSTM layers minimized the prediction loss. Although the accuracy of the model does not achieve state-of-the-art, the paper does propose some groundwork on how hyperparameter tuning can help improve a low-performing model even in a limited feature-engineering setting restricted to only a historical time-series.

1. Introduction

Time series prediction involves forecasting the future behavior of some random variable based on its past behavior as well as exogenous factors where available. One of the assumptions is that there are multiple patterns within the given random variable that unwrap over time which we can model. Parametric models have been developed over time using the appeal of those patterns: ARIMA for instance uses Auto-Regression and Moving-Averages; such models are fitted with some parameters of choice to historical data, and then are used to output points in future time periods. This undeniably presents an opportunity to multiple businesses across diverse industries ranging from finance and energy to healthcare and meteorology to unleash the appeal of machine learning as a novel tool in its potential for predictive analytics to address time series problems. Prediction is not an unprecedented problem, and besides Machine Learning is not the sole tool available to address it. Predictive analytics has long sought the power of more traditional statistical approaches. Autoregressive Integrated Moving Average (ARIMA) for instance is a regression method widely used in the financial sector. Deep Learning also isn't new to the field. Multi-Layer Perceptrons (MLPs) have been used over the years (8; 6). Yet, more recently, the appeal of Recurrent Neural Networks and it's many variations, have opened up a whole new field of methodologies to model the nonlinear relationships held within time series data. RNNs are used in sequence modeling, of which time series prediction is a subset. RNNs involve signals passing through recurrent connections which act as a memory for the network; these signals can use information in the memory to then forecast values for future time intervals. More appealing yet to practitioners have been LSTMs (4) which unlike traditional RNNs use memory cells to capture long-term dependencies between time intervals. With RNNs, the vanishing gradient problem poses an issue as the size of the network grows. The LSTM solves this problem, as outlined in (4). This project is an attempt to support the area of research in using deep learning methodologies for time series forecasting, and seeks to identify for a given architecture the optimal number of neurons within an LSTM unit that optimizes out the predictions. My analysis is based on Apple stock prices data (Apple Inc., $AAPL$). Being a major member of $S\&P$ Dow Jones Indices with the highest market capitalization, one would certainly reckon some great financial interest in the movements of its stock prices. The business and practical applications of the results of this paper, or at least of the work and experiments presented here, should certainly be a useful resource in a FinTech-driven economy.

2. Related Work

Machine Learning models have long been used in finance, successfully, for instance in portfolio optimization (5) which involved applications of Neural Networks and Reinforcement Learning. Within risk management, risky assets were classified by Butaru et al. (3) in a supervised fashion using random forests and more complex classifiers like Deep Neural Nets. Here, in this project, I approach the stock price prediction problem. Different methods have been explored already. Bernal et al. (2) used a subclass of Recurrent Neural Networks, Echo State Networks (ESNs) to predict stock prices of the $S\&P$ 500. Their network outperformed a Kalman filter, thereby forecasting more of the higher frequency fluctuations in stock price. Naeini et al. (7) approached the problem using Multi-Layer Perceptrons.

3. Data Preprocessing

I narrowed my analysis from the broader $S\&P$ 500 to look at the performance of one stock: using AAPL stock prices downloaded from YahooFinance [1] for the last 39 years. The evolution of the price over the last three years is illustrated in Figure 1. Figure 2 provides an excerpt from the tail of the data with respect to the features available in the training space and subsequently the inputs fed to the network. I aim to predict the future direction of the stock using past historical data using a time window of 22 days of open price, high price, low price, adjusted close price and volume data. That means that at any time step t, I train the model on data from $[t - 22 : t - 1]$ and use the model learnt to output a prediction for the stock price at the t^{th} time step. I first rescale the data using a MinMax normalization:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

This reduces the data to the interval [0, 1]. This is appropriate since I eventually will be using a ReLU activation function in one of my Dense layers. The train/test split used on the dataset is 90%-10% respectively. To check if the model is overfitting, I use an accuracy benchmark of 55%, so if it reaches that level, that should imply it predicts no better than random guessing. I would also like to point out that although traditionally, while in statistical time series models such as ARIMA, data preprocessing often involves steps such as first order differencing to remove the effects of seasonality trends in order to enforce stationarity, such steps are not necessary when training with Recurrent Neural Nets as the latter are known for their ability to in fact capture complex nonlinear relationships between features (here, those features are the historical data). Some industry practitioners might argue that a log transformation might be appropriate, but upon analysis, the level of heteroscedasticity in the data was not disruptive enough to justify such a transformation. During preprocessing, each series' scikit-learn scaler that was used in the standardization process was stored so that those parameters could subsequently be used to inverse transform the normalized forecasts output by the model.

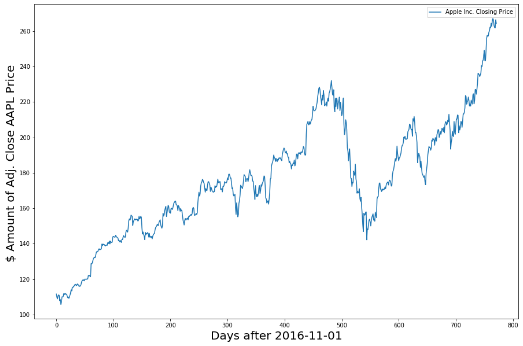
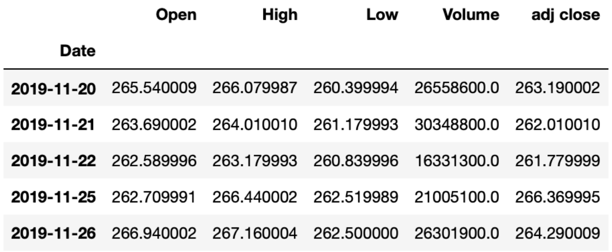


Figure 1. AAPL Stock Price (Adjusted Closing Price) Movements from 2016-11-01 to 2019-11-01



	Open	High	Low	Volume	adj close
Date					
2019-11-20	265.540009	266.079987	260.399994	26558600.0	263.190002
2019-11-21	263.690002	264.010010	261.179993	30348800.0	262.010010
2019-11-22	262.589996	263.179993	260.839996	16331300.0	261.779999
2019-11-25	262.709991	266.440002	262.519989	21005100.0	266.369995
2019-11-26	266.940002	267.160004	262.500000	26301900.0	264.290009

Figure 2. Excerpt from the Original Data Prior to Preprocessing

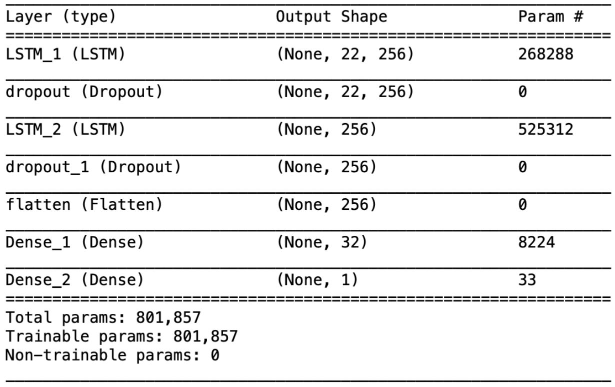
4. Experiment

4.1. Choosing Network Architecture

At first, I tried experimenting with a two hidden layer model, using an LSTM within the first layer and Dense layer (with a ReLU activation) as the output layer. This gave a poor performance with regards to the prediction accuracy: I would attribute such a model weakness to the small dimensionality of the data, or the limited features used for the input data. A better feature engineering using different strategies would have yielded a much better performance using such a basic 2-layer network (some feature engineering approaches that could be explored in future work, include, but are not limited to, sentiment analysis of news and social media feed, using existing historical time series data to generate moving averages and other statistical metrics such as autocorrelation, using portfolio indices' performances, amongst others). This firsthand poor result made it clear that a more enhanced model had to be explored. I used as inspiration for building my network architecture the winning solution from a Kag-

[1] https://finance.yahoo.com/quote/AAPL/history/

gle competition from some time ago on time series data for analyzing Wikipedia web feed. The winning architecture involved a multivariate Seq2Seq Recurrent Neural Net (further augmented with an attention interface) [2]. Although the original solution for the competition used GRU cells at each time step, in my implementation as outlined in Figure 3, I did not include these layers, but instead used 2 LSTM layers. Adam optimizer was used to optimize the objective loss function. A dropout layer was added to enforce some level of regularization after each LSTM layer. The first Dense layer used a ReLU activation while the second Dense layer used a linear activation function and was the final output layer.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
LSTM_1 (LSTM)                (None, 22, 256)           268288
_________________________________________________________________
dropout (Dropout)            (None, 22, 256)           0
_________________________________________________________________
LSTM_2 (LSTM)                (None, 256)               525312
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
Dense_1 (Dense)              (None, 32)                8224
_________________________________________________________________
Dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 801,857
Trainable params: 801,857
Non-trainable params: 0
_________________________________________________________________
```

Figure 3. Training Model Architecture

4.2. Performance Measure

I chose two performance measures, namely the mean squared error (MSE) and the mean absolute percentage error (MAPE) to evaluate the prediction accuracy of my model. MSE is a good measure since it outlines large forecast errors. MAPE has an edge when it comes to interpretability- it is the average percentage that a point-by-point forecast deviates from the true value.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_{1,i} - x_{2,i})^2$$

$$MAPE = \frac{100}{N}\sum_{i=1}^{N}|\frac{x_{2,i} - x_{1,i}}{x_{1,i}}|$$

where N is size of the data set, $x_{1,i}$ is a predicted value and $x_{2,i}$ is the true value.

The MSE was also used as the objective loss function to minimize.

4.3. Hyperparameter Tuning: Random Search v. Grid Search

Despite the numerous hyperparameters that could possibly be fine-tuned to improve the accuracy of the model (for instance, number of neurons, learning rate, window size, data size, data dimensions (add or drop some dimensions), batch size, number of training epochs, etc...), in this project, I focus on tuning for only two hyperparameters, namely the number of units/neurons and the learning rate. Obviously, in an industry-scale application, one would find true gains, financially, from refinement of as many of the above mentioned hyperparameters. Here, I implement the method of Random Search proposed by Bergstra et al. (1) as opposed to using the more traditional (and yet popular) grid-search method. Bergstra et al. demonstrate that with randomly chosen trials under Random Search, one achieves greater efficiency in hyper-parameter optimization than trials on a grid. $Optimal$ models are found at a fraction of the computation time it would take to find the same model under a manual grid search approach. The tuner is implemented within Keras using tensorflow as backend. The following two hyperparameters were tuned:

- Number of Units (η_x): Used Random Search to find best performing model in the range (32, 512) units incremented in steps of 32 units.
- Learning rate (α): Randomly searched in the range $(10^{-2}, 10^{-3}, 10^{-4})$

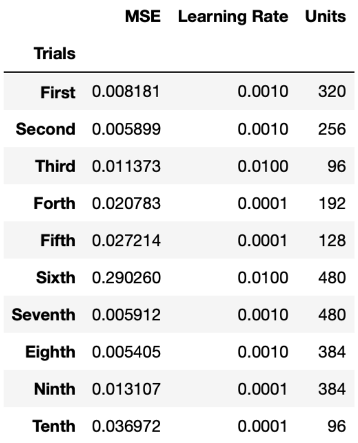
The random search executed 10 trials, each running for 10 epochs. In each epoch, the model was trained on 8820 samples and then validated on 980 samples (equivalent to our 90-10 split established beforehand), and the best performing model from the tuner (with respect to the smallest validation loss) was using 384 units within both LSTM layers and a learning rate of 10^{-3}. The Mean Squared Errors from the 10 trials are reported in Figure 4. The reported wall clock time to run those 10 trials was

$$22\ min\ 34\ s$$

It was particularly interesting to note the case for the sixth trial whereby with 480 units and a learning rate of 10^{-2}, the MSE was significantly higher than for any of the MSEs of the other trials. I treated this case as an outlier, and decided to drop it. Plotting the number of units against the MSEs by trials (after removing the effects of the outlier), we obtain the results illustrated in Figure 5 below.

Figure 5 shows that the more the number of units used within the LSTM layers in the predefined architecture, the better the accuracy of the model. It's also noteworthy that despite the $best\ model$ as selected by the tuner was the one

[2] https://github.com/BenjiKCF/Neural-Net-with-Financial-Time-Series-Data



	MSE	Learning Rate	Units
Trials			
First	0.008181	0.0010	320
Second	0.005899	0.0010	256
Third	0.011373	0.0100	96
Forth	0.020783	0.0001	192
Fifth	0.027214	0.0001	128
Sixth	0.290260	0.0100	480
Seventh	0.005912	0.0010	480
Eighth	0.005405	0.0010	384
Ninth	0.013107	0.0001	384
Tenth	0.036972	0.0001	96

Figure 4. Random Search Reported Trials MSE (Val. Loss)

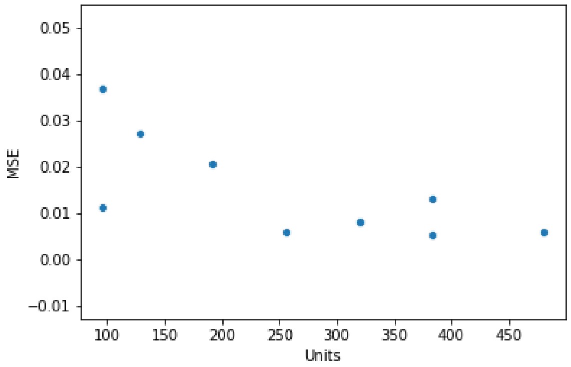


Figure 5. Random Search: Number of Units vs MSE (Val. Loss)

with 384 units, yet the difference in MSE between the best model with 384 units and the best model with 480 units is very insignificant, so any fine practitioner should have no trouble using LSTM layers with 480 units even.

In evaluating the effectiveness and computational advantage put forth by Bergstra et al. (1) in their paper, I also ran hyperparameter tuning using the more traditional approach of **grid search**: I designed a grid with the number of units and the learning rates established as above, but this time, I executed a grid search algorithm to loop over the values of the grid in fitting different models with each of the hyperparameter values from the grid, and the validation loss (i.e. the mean squared error) from the last training epoch was reported for each combination of hyperparameter values from the grid. The wall clock time reported for running grid search was

$$1\ hr\ 58\ min\ 54\ s$$

The results from grid search are illustrated in Figure 6. The optimal hyperparameters as produced under grid search were reported with LSTM layers with 480 units and a learning rate of 10^{-3}. This result corroborates with that of random search, and it is also interesting to see how with learning rates of 10^{-3} and 10^{-4}, the validation loss decreases as the number of hidden units within the LSTM layers increases, thereby reinforcing my hypothesis set in my problem statement.

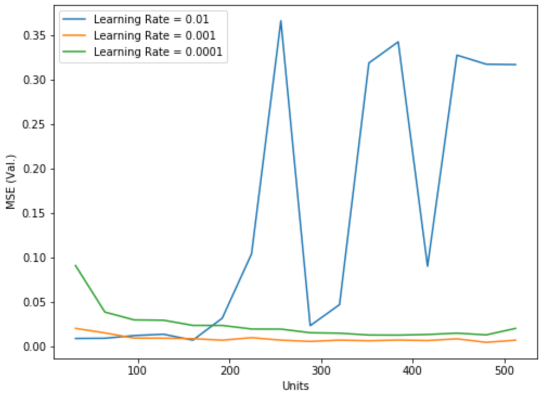


Figure 6. Grid Search: Number of Units vs MSE (Val. Loss)

This result clearly shows that in building recurrent neural nets with LSTM layers using time series data, the use of Random Search over grid search in finding optimal hyperparameter values can certainly bring a computational speedup during training (there is a significant wall clock time difference from optimizing under each method), which is a significant gain considering the growing size of time-series data over time and the increasing tendency of training even more complex models than my somewhat shallow 2 LSTM layers network here.

4.4. Benchmark Model

The benchmark prediction model used is the $median\ model$: it involves projecting the median value in each original time series into the future. In other words, for each series t, the median from $[t - 22 : t - 1]$ is predicted for the t^{th} time. This model is simple and interpretable, for if a given time series has a trend, then one would agree that such trend would continue into the near future under the assumptions of a time series' seasonality. This helps predict a time series while still being conservative about its trajectory. Using median is an alternative to using the mean for point forecasts, and happens to be a better alternative to the mean given the loss function defined as the objective above. As an example, $fan\ charts$, popularized by the

Bank of England, have used median point estimates many times before to represent inflation forecasts.

5. Results

Given the data and the task set forth in the problem statement above, the Deep Learning model does align with the expectations. As outlined in Figure 7, the deep learning model does show a significantly better performance in predicting future stock prices than does the baseline median forecast model (that surprisingly happens to still be widely used as a base for certain statistical models in time-series analyses in the financial industry). An RNN would output strong predictions if the size of the dataset is large enough. In this case, I used data of APPL's stock prices since 1980-01-01; hence the large size of the data and the fact that LSTMs can learn longer-term dependencies better than statistical models do account for the alignment of the expected and observed results. One should however exercise caution here: this does not provide evidence that Deep Learning will always, and given any time series data, outperform traditional statistical models.

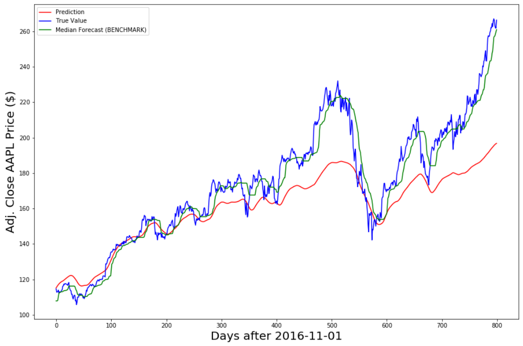


Figure 7. RNNs: Target vs. Forecast vs. Benchmark Adj. Close AAPL stock prices

References

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305, February 2012.

[2] Armando Bernal, Sam Fok, and Rohit Pidaparthi. Market time series prediction with recurrent neural networks. 2012.

[3] Florentin Butaru, QingQing Chen, Brian Clark, Sanmay Das, Andrew W Lo, and Akhtar Siddique. Risk and risk management in the credit card industry. Working Paper 21305, National Bureau of Economic Research, June 2015.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[5] Evan Hurwitz and Tshilidzi Marwala. State of the art review for applying computational intelligence and machine learning techniques to portfolio optimisation. *CoRR*, abs/0910.2276, 2009.

[6] Nowrouz Kohzadi, Milton S. Boyd, Bahman Kermanshahi, and Iebeling Kaastra. A comparison of artificial neural network and time series models for forecasting commodity prices. *Neurocomputing*, 10(2):169 – 181, 1996. Financial Applications, Part I.

[7] Mahdi Pakdaman Naeini, Hamidreza Taremian, and Homa B. Hashemi. Stock market value prediction using neural networks. pages 132 – 136, 11 2010.

[8] Cyril Voyant, Marie-Laure Nivet, Christophe Paoli, Marc Muselli, and Gilles Notton. Meteorological time series forecasting based on MLP modelling using heterogeneous transfer functions. *CoRR*, abs/1404.7255, 2014.