CS230

Predicting Stock Movements Using LSTM

Matthew Roche
Department of Computer Science
Stanford University
mtroche@stanford.edu

Abstract

In this project, I take on the problem of automating the process of financial forecasting. Specifically, I build an RNN model for the purpose of predicting closing stock prices at a future date given a series of consecutive previous closing prices for the same stock.

1 Introduction

Financial forecasting is a problem that many people spend their professional lives tackling. The jobs of many people on Wall Street is to try to accurately project the direction and magnitude of asset evaluations, and one of the most common financial realms to make predictions about is the stock market.

Stock prices are often regarded as notoriously difficult to predict, as many see them as too prone to randomness to make rigorous probabilistic predictions about. Because of this, several successful methods, such as billionaire mathematician and investor James Harris Simons' Medallion Fund, are thought to rely on black-box methods such as neural networks. Unfortunately for the rest of us, the methods behind successes remain well-kept proprietary secrets, so I instead must develop my own model to try to tackle to task.

Automation of this task is one that would have lucrative rewards, if done successfully. In this project, I attempt to do just that. I build a model that takes as input a series of consecutive (daily) stock closing prices, and outputs a single scalar representing the predicted price for the next day.

2 Related work

Unfortunately for my research, groups that succeed in creating accurate and precise automated financial prediction agents are well-incentivized to keep their tactics proprietary and private. However, I was able to find several existing implementations that tackle the problem of stock price change categorization and scholarly articles on the matter.

One related implementation that I was able to find, however, was an LSTM model implemented in tensorflow used for predicting cryptocurrency prices given the simultaneous price sequences of several [1]. This implementation was part of a tutorial on a website 'pythonprogramming.net'. One feature of this project that I noted is that even it only output the predicted price for a single cryptocurrency, it took as input the sequences of 5 different types of cryptocurrency. I decided for my project, taking multiple concurrent sequences as input would increase the computation overhead more than it would help the network learn. I believe that this assumption was correct, because my project saw similar results with this one, using only a single sequence as input to the network.

Another implementation that I found online during my preliminary research was a Recurrent Neural Network used by Lilian Weng to predict stock prices of S&P 500 companies [2]. This network used LSTM cells as well, which I decided to utilize for my network.

One feature which both of these implementations utilized that I would leave out of my model was Dropout. I did not use this because I learned in class that it is better to leave efforts for reducing variance for after one has satisfactorily minimized bias.

I did however, use the frameworks of these projects as guidance for setting up my own model and running experiments.

3 Dataset and Features

I have mentioned that the format of my input data is a series of consecutive stock prices, and the format of my output data is a scalar representing a predicted future price of the stock. More specifically, the input vector x is an ordered series of 30 consecutive daily closing prices for a single stock. The output scalar y is the closing price of that same stock on the 31st day, and the output of the model is a prediction of this value.

I obtained the stock price information from Google Finance data repositories. (In fact, I utilized a feature of Google Sheets that allows one to directly import ordered stock price information into an online spreadsheet.) The dates of the stock price data I collected range from 1967 to 2017. The companies for which I collected stock price data are all publicly traded. I aggregated data on a variety of corporations such as Apple (AAPL), Amazon (AMZN), General Motors (GM), General Electric (GE), Ford Motor Company (F), Sprint Corp (S), etc.

After obtaining a data repository, my next task was to format the data into (x, y) pairs to train on. In essence, I had to take sequences of thousands (and even tens of thousands, for a few) of consecutive stock prices, and extract from the sequence many subsequence of 31 stock prices, where the first 30 values would make up the input sequence x and the last (31st) value would be the output scalar y. For each company's data sequence of length n, I extracted the inputs and outputs by traversing from the (0)th index to the $(n-32)$th index, converting the subsequences of length 32 to (x, y) pairs along the way. Using this technique, for a company's consecutive stock record of length n, I was able to construct $n-31$ total (x, y) pairs, which is the maximal number of data points attainable without duplicates.

However, I did not feed the data directly into the model with that form. Instead, in order to make training easier and to correct for the inevitable variation of prices along the different company's stocks and along the different time periods, I normalized the inputs and outputs. I normalized by dividing the calculating the mean closing price for each of the length-32 sequences, and then dividing each element of the corresponding (x, y) pair by that calculated mean. I did this without worry of a divide-by-zero error, because there is no real situation in which a stock price would average 0.00 over 32 days. I also opted against calculating the mean for the entire dataset and using the inverse of that value as the normalizing constant because this approach would not likely result in data-points whose means were all 1.0 (or even in that vicinity, necessarily), while the individual-normalization approach would be guaranteed to do so.

Of course, using this formatting technique, the model is expected to predict the price in the new normalized scale, so making actual predictions with it would mandate rescaling. Across 42 companies, I accrued 174,562 (x, y) data points of this form. Because this is a larger number of data points, I split my data into 90% train, 5% dev, and 5% test. All in all, my test set contained 157,109 (x, y) pairs, my dev set contained 8,728 (x, y) pairs, and the test set contained the remaining 8,725 pairs.

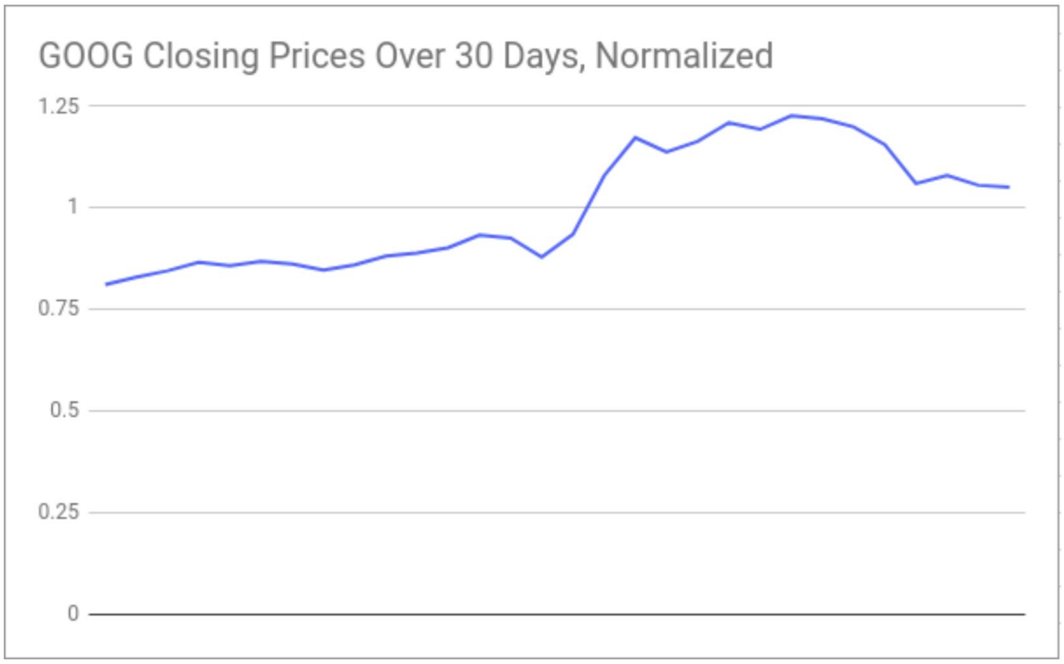

Figure 1: An example 30-day input data sequence for Google stock post-normalization.

4 Methods

The model that I defined for the problem uses a computational device known as LSTM, common in Recurrent Neural Networks. LSTM stands for Long Short Term Memory, and it refers to a type of computational cell that was initially devised to solve the vanishing gradient problem for RNNs. It uses three gates - namely the input gate, output gate, and the forget gate - in order to filter sequential information through and across computational cells [5]. Another cell type known as the GRU is designed for a similar purpose and uses two gates, but it is less commonplace and I am not using it for this model.

Specifically, the model that I applied to this problem was a 5-layer neural network. It is implemented in keras with a tensorflow backend. The input to the network is of course a length-30 sequence. Then, the first layer is an LSTM layer with 30 units. The second layer is again an LSTM layer with 30 units. The third layer is a fully connected layer (also known as a dense layer) of size 64. This is followed by a fully connected layer of size 32. A fully connected output layer with one unit comprises the fifth and final layer. All the nonlinear activation functions that I used for this network were ReLU, or Rectified Linear Unit. During training, I decided on a mini-batch size of 128.

To guide this network, I used the simple squared-loss cost function $(y - y_{pred})^2$, leading to the Mean Squared Error loss function $\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - y_{pred}^{(i)})^2$. I decided on this loss function because I thought that it was a suitable loss function for a regression problem, as it is commonly used.

5 Experiments/Results/Discussion

During my experimentation, one of the things that I optimized for was the learning rate hyper-parameter. I sampled the learning rate on a logarithmic scale, and ultimately decided to use the learning rate $\eta = 0.012$, because it is the value that returned me the lowest training loss after a single epoch. Therefore, I judged that it was the value that allowed for the best/fastest learning.

Most of the existing implementations that I came across used classification accuracy as a defining metric of performance. Since I framed the model to solve a regression problem, I had to some computation in order to evaluate my model against the same metric. To evaluate the accuracy of my model, I defined the binary classification problem of whether or not the stock increased on the next day's closing. My model prediction was 'correct' in this sense if the ground truth value of y and the predicted value of y_{pred} were either both greater than or both less than or equal to the last element of the input vector.

Using this evaluation metric, my model saw modest success. The highest classification accuracy that I observed by the model was 62.3%, and this was only on a subset of the testing data and a result that was not widely replicable. By and large, my model had an overall classification accuracy of 58.2% on the test data. It is worth noting that in a binary classification task where either outcome is equally likely, a baseline of random guessing would be expected to attain a 50% accuracy rate. However, observing an accuracy rating that is 8% higher than 50% is not likely to happen on such a large test set by chance, so this gain is significant, if not impressive. It is difficult to interpret the results, because the value of Bayes' Error in this application is unclear. Bayes' Error is likely far from perfect regarding stock prediction, but based on the enigmatic successes of analytical hedge funds providing a murky bound for Bayes' Error, it is reasonable to conclude that my error is significantly divergent from Bayes' Error. Ultimately, the problem of classification is clearly not solved.

However, while the categorical accuracy of the predictions made by the model is lackluster, I noticed that when the model predicted the category correctly, it demonstrated some predictive power in terms of the score. Of the set of points in the test set that the model predicted correctly on, 27% of the time the model's predictions were within 10% of the ground truth.

While this on its own is not a grounds for celebration, it does demonstrate potential for a model of similar form to return accurate predictions on the type of sequence data I trained my model on.

6 Conclusion/Future Work

In conclusion, the model did not perform as well as I had initially hoped. While one may be able to generate narrow profits under the guidance of this model, it is not powerful enough to even shine a light on the successes of hedge fund black-box secrets. However, the model was able to perform well enough that it is very unlikely to be the result of pure chance, and it has started me off in the right direction in regard to deep learning with sequential data.

If I had more time and computational resources, I would consider training one network for each company's stocks, or at least, one network for each industry. I would guess that not all company's stocks follow the same trends exactly, and similarly that the technology industry would follow different trends than the agriculture industry would have different tendencies from the automotive industry etc. Therefore, by using one network for stock price predictions across various companies and industries, I may have been over-generalizing and thus limiting the acuteness of the network's predictive power.

I would also look into rectifying the bottleneck of price movement direction unpredictability. I would like to be able to measure the predictability of stock price trajectories before making predictions on them, because I read that this is a strong suit of successful analytical hedge funds. I was not able to pursue this during my project because I could not figure out how to model this as a supervised learning problem.

One possible way to do so is to train a single neural network (or several, but I don't know if this would yield any unique results) to make predictions on the trajectory directions of the stock values, and then for each input x that the a network outputs the wrong direction, use the data point (x, 0) to train the predictability-measuring network, and for each input x that a network outputs the wrong direction, use the data point (x,0) to train the predictability-measuring network. The output of the predictability network would be a scalar taking on values in the range (0,1), where 0 indicates very unpredictable and 1 indicates very predictable. The two main drawbacks of this proposal are that it would take a significant increase in computation to train multiple neural networks and it may be over-simplifying the problem with somewhat contrived data. However, if I were to get such a network working, I could filter out unpredictable inputs and create a much more powerful financial forecaster (because as Warren Buffett advises, there are no called strikes on Wall Street).

Another method that I considered as a possibility was to frame the problem as a reinforcement learning model, perhaps using Deep Q-Learning. Deep Q-Learning is a method that uses a neural network to evaluate the favorability of state-action pairs, by exploring possibilities while it accrues the experience it needs to learn. Its learning is defined on the basis of (s,a,r,s') tuples, which are an initial state, an action, a reward for taking that action in that state, and a new successor state. (It also involves transition probabilities between states.) There seems to be an analogy between states in a reinforcement learning problem and the state of a portfolio coupled with an input sequence of relevant stock prices; actions would correspond to buying, selling, or holding; rewards would have to do with the amount of money made (perhaps as well as with the risk involved); and the uncertainty of the stock market is analogous to the possible uncertainty of transitions in such a reinforcement learning problem. However, I am still in the process of trying to solidify the relationship and concretely define the problem, so I do not want to act like I have more of it figured out than I do. In any case, with such a problem state solidified, and with access to a very large amount of computational power, this idea might have the potential to produce a more effective agent than the model I was able to come to.

7 Contributions

As this was taken on as an individual project, the contributions section is self-explanatory. (Note: Gradescope does not let me submit a code.zip file along with the report pdf file.)

References

[1] "Cryptocurrency-Predicting RNN Model." Python Programming Tutorials, pythonprogramming.net/crypto-rnn-model-deep-learning-python-tensorflow-keras/.

[2] Weng, Lilian. "Predict Stock Prices Using RNN: Part 1." Github, 8 July 2017, lilianweng.github.io/lil-log/2017/07/08/predict-stock-prices-using-RNN-part-1.html.

[3] Rajesh, Neeraj, and Lisa Gandy. "CashTagNN: Using Sentiment of Tweets with CashTags to Predict Stock Market Prices." 2016 11th International Conference on Intelligent Systems: Theories and Applications (SITA), 2016, doi:10.1109/sita.2016.7772262.

[4] Goldenburg, Blair. "AI Hedge Fund: Artificial Intelligence Taking over Hedge Fund Markets." Stock Forecast Based On a Predictive Algorithm | I Know First |, I Know First, 23 Sept. 2017, iknowfirst.com/rsar-ai-hedge-fund-artificial-intelligence-taking-over-hedge-fund-markets.

[5] Gers, Felix A. et al. "Learning to Forget: Continual Prediction with LSTM." Neural Computation 12 (2000): 2451-2471.

[6] "Tensorflow." GitHub, github.com/tensorflow.