Using Neural Networks to Predict Stock Price Changes for an Exchange Traded Fund

Team Firestorm
David Roden (dr2de), Beni Shpringer (b2ux), Ashwanth Samuel (as7cs)
Stock Price Prediction Project

I. Abstract

The rise of deep learning in recent years has resulted in its application to many fields, especially those concerned with prediction. This paper focuses on the prediction of stock market data - a field and applied practice that was often restricted by its time-series like nature. Data that is affected by time can often be difficult to model because of factors like seasonality and stationarity. The inherently noisy and stochastic nature of markets severely limits the forecasting accuracy of the applied models. Today, we are able to combine our knowledge of deep learning and new technologies, to thoroughly test price movements on the Limit Order Book. In this report, we propose several deep learning models, that can account for the aforementioned problems, to achieve our goal of predicting future price movements. The models that are evaluated include a Feed Forward Neural Network, a Convolutional Neural Network, and a Recurrent Neural Network. Finally, we evaluate and determine which of these models are most useful for predicting a mid-price movement for the next immediate market day based on financial data from 1/06/17 and 1/09/17. Ultimately, the RNN outperforms the other two neural networks.

II. Introduction

In this paper, we conducted a three-class classification for a short time-horizon that is 200 steps (transactions) in the data. We predicted on SOXX, which is an Exchange Traded Fund (ETF) that represents the entire semiconductor industry. We normalized the data using the techniques in the literature we read in order to make our models potentially generalizable across other stocks, and also accounted for the time-series nature of the data. The models we used are multi-class random forest, FFNN, CNN, and RNN. We evaluated our models by calculating prediction accuracy on the next day's trading data.

A significant consideration we paid attention to when dealing with our specific data was how far in the future we wanted to predict. Picking a period of time that was either too near or far in the future could result in stationarity (where local properties are preserved over time) or seasonality (when patterns in the data appear cyclically.) As such, we were cognizant of these dangers and selected a time period that was conducive to our final goals. We therefore chose 200 events as a reasonable amount of time in which to make predictions, given how quickly the stock market fluctuates.

III. Literature Review

There have been two relevant papers that have aided our team's research and endeavors. The paper, "Using Deep Learning for price prediction by exploiting stationary limit order book features" by Tsantekidis et al[1] included many interesting ideas and techniques that are utilized in our project. The

[1] Tsantekidisa et al, "Using Deep Learning for price prediction by exploiting stationary limit order book features" (2018).

research team's specific objective was to create a multi-class classifier for stock market data. They would predict whether a stock's price would go up, go down, or stay the same in the future. They varied the ways in which they defined the "future," as well as what constituted a stock's price remaining constant, in order to compare various models for accuracy. The novel idea from this paper was the introduction of "stationary" features that they created in order to create a generalizable model that worked across stocks and that also normalized for the time-series nature of stock market data. Specifically, they created two features: the first was what they called "price-level difference", which essentially looked at the different price levels in the limit order book, and compared them to the mid-price at a current time as a percentage, rather than as a raw difference. This allowed for consistent analysis across stocks due to different stock ranges (some are around $10, and some are $1,000). The second feature they created was a "mid-price change" which again was a percentage change statistic. This feature viewed how the mid price had changed when a given order had been made, relative to some previous time period. Again, this was generalizable across stocks regardless of stock price, and also accounted for the time-series nature of the data. They used a two-layer model in which a CNN was the first model used on the data as a feature extractor of the limit order book, which produced a new time series that was then fed into an LSTM (Long Short-Term Memory) which performed the predictions. They found that a shorter time horizon was better for prediction, and they used measure of stationary price that created a training set with 20% upward price movements, 20% downward price movements, and 60% stationary price movements.

Additionally, the paper titled, "Modeling high-frequency limit order book dynamics with support vector machines[2]," was centered around capturing the dynamics of high-frequency limit order books in financial equity markets and automating real-time prediction of metrics such as mid-price movement and price spread crossing. The reason that this research was conducted was because High Frequency Trading (HFT) firms started to account for a larger percentage of daily stock trade volume, as well as equity value. The increased presence of HFT firms has deeply affected the traditional traders working in the finance industry. This paper employed a multi-class SVM in order to capture the dynamics for a limit order book and forecast the movements of the mid-price and direction of bid-ask spread crossings over short time periods. From this paper, we learn that SVMs can be effective for this problem, but not incredibly accurate. As a result, we decided to not use an SVM in our analysis for this project.

IV. Data

For this project, we used a single index fund, SOXX, as our primary source of data to train and test our models. SOXX is the iShares product which represents the entire semiconductor industry. Some example companies that comprise SOXX, with their relative representation within the index, are Nvidia (8.99%), Intel (7.83%), and Texas Instruments (7.75%).

We used two days worth of data in order to train, validate, and test our models. The use of two consecutive days was necessary because we wanted the training data to be as applicable as possible to the testing data. Ultimately, if our model was to be put to use, we would like to use the most recent data

[2] Zhang and Kercheval, "Modeling high-frequency limit order book dynamics with support vector machines," (2013).

possible to be able to predict the next day of values. The two days we used were Friday 1/6/17 and Monday 1/9/17. We omitted the first 1,000 and final 1,000 rows from both days for training and testing because there is noise at the beginning and end of each trading day that would severely affect training.

The data from 1/6/17 included approximately 186,000 rows. We trained our models using the first 150,000 rows, which made up approximately 80% of data. We validated and tuned our models using a validation set of the final 36,000 rows from 1/6/17. After tuning models using this train/validation split, we then built models using the entire dataset from 1/6/17.

We then applied our models to the entire day's worth of data from 1/9/17, which had approximately 184,000 rows, in order to test and report performance for our models.

SOXX Index Fund	Number of Values
Training (01/06/17)	150,000
Validation (01/06/17)	36,076
Test (01/09/17)	184,282

V. Pre-Processing

We completed pre-processing in order to create the features that we wanted for our model. We used two main features which produced at least 21 inputs for each event (depending on the model). 20 of the inputs are closely related, but required a lot of logic in order to calculate them. We created the full order book by using each line item in the book, and adjusted the books (which are represented as dictionaries for both bids and asks) according to the book event type. The dictionaries were structured like bids = {price: {order_id: quantity}}.

- When there was an add at a certain price and quantity, we looked in the corresponding dictionary (bid or ask), found the price, and added an order with the quantity at that price (or added the price to the outer dictionary if the add was for a price not already represented). We also stored in a different dictionary bid_orders = {order_id: price} to more quickly look up cancels and modifies.
- When there was a cancelled order, we looked up the order_id in the bid_orders dictionary to find the price in the bids dictionary. We then removed that order from the bids dictionary and the bid_order dictionary. We also checked if canceling that order removed all orders at that price, and if so, we removed that price level from the dictionary as there were no longer orders at that price level.
- When there was a modify, we again looked up the price in the bid_orders dictionary. Rather than change the previous order, we simply deleted it and added a new one (sort of like a combined cancel and add). Again, if the previous price now no longer had any orders, we removed that price level from the order book.

- Finally, when there was a trade, we removed the quantity traded at that price that was previously added for a given order. If there were no more shares requested at that price level, it was removed.

After those updates, we had sorted price levels for both bids and asks, with the number of shares requested at each price level. We took the 10 highest bids and the 10 lowest asks and multiplied them by their quantity requested to get a total depth at each price level. This gave us 20 of our features. The last features we included were mid price change compared to a previous step count (in our case, 200 steps), as well as a standardized mid price - that is, the current mid price compared to the mean mid price for the entire training set, and divided by the mid price standard deviation. We calculated the mid price at each event as the average of the highest bid and lowest ask. We calculated the current mid price divided by the mid price from 200 events previous, - 1, which gave us a 21st feature. We then standardize all of these features for better performance with a neural network and for comparison across stocks.

Our labels were mid price movement 200 steps ahead of a given event. We calculated this by dividing the future mid price by the current mid price, and subtracting one. If this ratio was greater than .0001 (a .01% change), it was considered an upwards movement and was labeled as a 1. If it was less than -.0001, it was labeled -1 as a downwards movement, otherwise the price was considered to have stayed the same. Given that SOXX traded around $120 on those days, a change of .01% represented a change in mid price of about $.01. We used these thresholds because they give relatively similar class sizes among our training set (approx 29% downward, 33% upwards, 38% stagnant movements). For a more generalized model that is capable of handling many stocks, these thresholds may need to be adjusted in order to achieve equally weighted classes. We chose to go with approximately equally weighted classes because it would give us a good sense as to whether our models were predicting well compared to random guessing, and would discourage the models from selecting a single class for every row. It was also convenient that the threshold equated to a movement of approximately $.01 which is the smallest denomination possible. However, a mid-price movement of only $.01 still was classified as a stagnant movement as this was less than .01%, while a mid-price change of $.015 was enough to be classified as a downward or upward movement.

VI. Baseline Model Description

We used Random Forest Classifier as our baseline model due to its compatibility with multi-class classification. With the 21 variable feature set on which we applied our neural nets, the accuracy was only 33%. We thought that other feature sets might explain the target class more sufficiently, so we tried a 40 variable model, in which the test accuracy was 37% using 500 trees and the entropy criterion. We figured there was likely noise in many of those variables, so we tried a variety of subset models. The best one was a 7 variable model, which achieved 40% testing accuracy using 500 trees and the entropy criterion. The feature importances for the model were as follows:

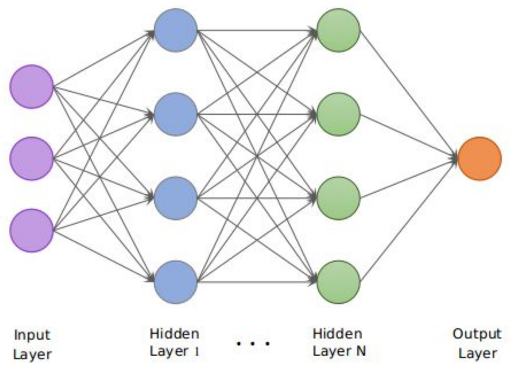
Feature	Importance
normalized_mid_price	0.328982

normalized_mid_price_change	0.209030
normalized_ask_depth	0.174045
normalized_bid_depth	0.142141
normalized_relative_depth	0.123724
book_event_type	0.012997
side	0.009080

VII. Methods and Analysis

Convolutional Neural Network

Our Convolutional Neural Network was tricky to create because CNNs typically work well with image processing. Thus, in order to run a CNN on our data, we had to modify and extract parts of our data for the full potential to be realized. First, we took the dataframe from the preprocessing procedures and added 20 more columns to the data. These columns represented the top 10 bid and ask prices, respectively, and thus allowed us to use these as our strides in the CNN. In order to additionally scale our data, we utilized the rectified linear unit (ReLU) activation which is unbounded on the axis of possible activation values.



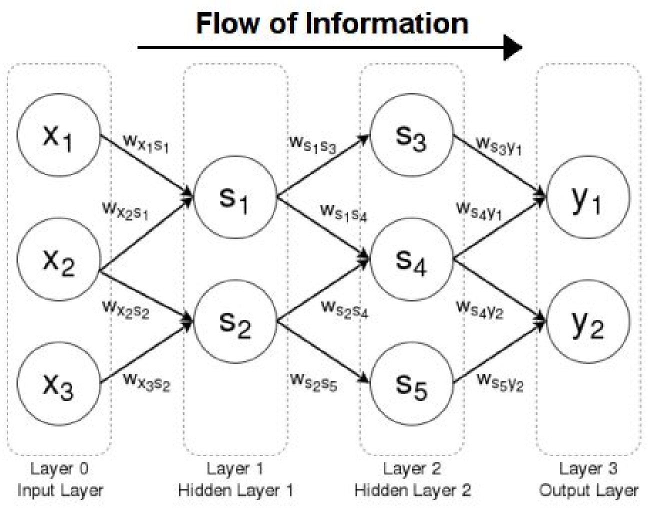
Simplified graph of our CNN with two hidden layers

After the additional bid and asks prices were included in our dataset, we set out to create the CNN. At first, we created three filters that were arbitrarily included. Then, we set the maximum number of iterations to 8000 and changed the batch size to 100. Lastly, we set the number of epochs to 114 after using the aforementioned numbers in our calculation.

We next created a conv2d with a stride of [1,1,1,1] as well as a pooling with a kernel of [1,2,2,1]. Then, we created three hidden layers and utilized the softmax function for our loss. The hidden layers of the network were also transformed by activation functions, specifically ReLU. Finally, we clipped our gradients and used the Adam Optimizer, which is the current default optimizer in deep learning, so that it would not explode. Finally, we trained the data on the model and subsequently fed it the appropriate data to make predictions on our testing data.

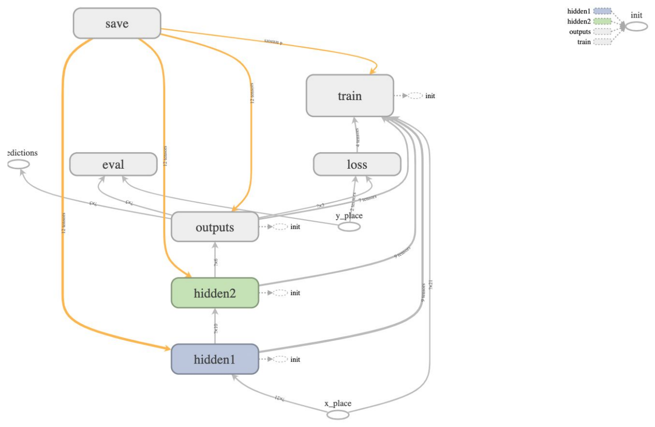
Feed Forward Neural Network

After much training and tuning of parameters, we used a feed forward neural network that begins with 21 inputs, as described above. We use two hidden layers, the first with 13 neurons, and the second with 8 neurons, each with a relu activation function. We then output 3 logits which, when put through a softmax function, represent the probabilities of each class.

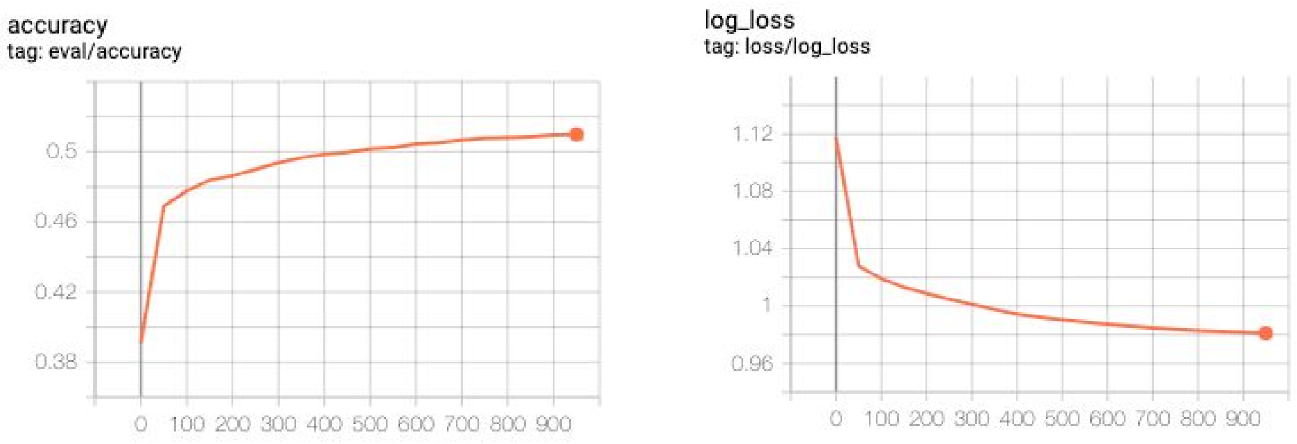


Simplified version of FFNN with two hidden layers

We use adam optimizer to train the model, and use a significant number of epochs - over 1,000 for training. We train a batch size of 200 lines at a time, sequentially.



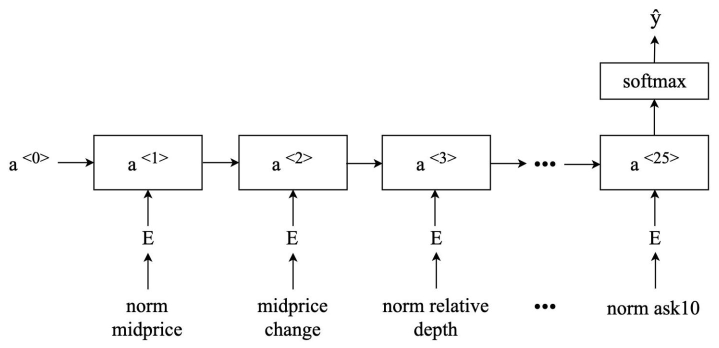
Tensorboard of our FFNN



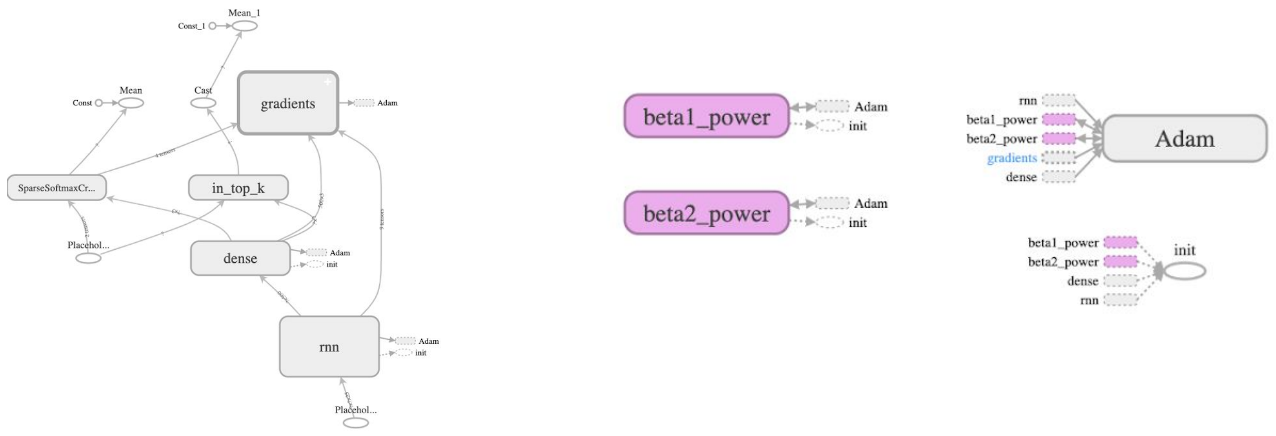
Charts of FFNN Training Accuracy and Loss, Calculated Every 50 Epochs for Entire Train Set

Recurrent Neural Network

We constructed our RNN using the BasicRNNCell function. We used sparse softmax cross entropy as the loss function and applied the Adam Optimizer. The architecture of the single-layer RNN is shown in the below diagram and tensorboard graph.



Here, $a^{<n>}$ is activation from the previous timestamp and and Es are the values from the input layer.



We experimented with a variety of combinations of tuning parameters which are described in the RNN table in the following results section.

VIII. Results Analysis and Improvements

For our CNN, we utilized learning rates of .002 and .02 and were able to achieve around 32% accuracy with both our training and testing data. After varying both the number of epochs and batch sizes for the CNN, we were only able to unearth minimal improvement in the model. We suspect that our CNN model did not work as well as the other two neural nets because of our idea behind shaping the data to the model. Further, there may just be too much noise in the dataset for our CNN to adequately predict future mid-price movements. For our FFNN, we used a learning rate of .0001. We were able to achieve over 41% accuracy on testing data with the FFNN. A confusion matrix which details the various prediction outcomes of our FFNN testing results is below. Given that we began with roughly even classes, a 41% prediction rate indicates some predictive power of our model. The green boxes indicate true positives, which again make a 41% accuracy rate. The red boxes indicate what we termed "bad misses," which was when the model predicted an upwards price movement, and the truth was a downwards movement, or vice-versa. Only 13% of our predictions were "bad misses," which is better than the 22% we'd expect from random guessing. The yellow boxes indicate "ok misses," which is a prediction of an upward or downward movement, when the true value was a no price-movement. Because the model does better than random for true-positives, and better than random for bad-misses, we believe that the model could be useful if put into use as we're more likely to predict a true upward or downward movement than making a type I or type II error. Of course, we'd have to further incorporate the magnitude of the movements in order to fully understand whether our predictive model could ultimately be profitable.

As one can see below in the following charts, there were many opportunities for tuning involving the learning rate, the number of epochs and the batch size. Furthermore, Below are the different results for both CNN, FFNN, and RNN after adjusting the aforementioned parameters. Additionally, we included an in-depth accuracy measure through the format of a confusion matrix for the FFNN.

CNN	**1**	**2**	**3***	**4**	**5**	**6**	**7**	**8**
Learn Rate	.02	.02	.02	.02	.002	.002	.002	.002
Epochs	25	50	25	50	25	50	25	50
Batch Size	64	64	128	128	64	64	128	128
Training Accuracy	.33526	.33119	.34172	.47644	.34217	.32036	.29749	.32407
Testing Accuracy	.35840	.34405	.36049	.30760	.33373	.34249	.31319	.33747

Results and Analysis summary for the CNN

FFNN	**1**	**2**	**3***	**4**	**5**	**6**	**7**	**8**
Learn Rate	.001	.001	.0001	.0001	.001	.0001	.0001	.0001
Epochs	2000	1000	1000	2000	1000	1000	1000	2000
Batch Size	200	200	200	200	500	500	100	500
Accuracy	.3901	.3898	.4102	.4027	.4013	.4068	.4074	.4017

Results and Analysis summary for the FFNN

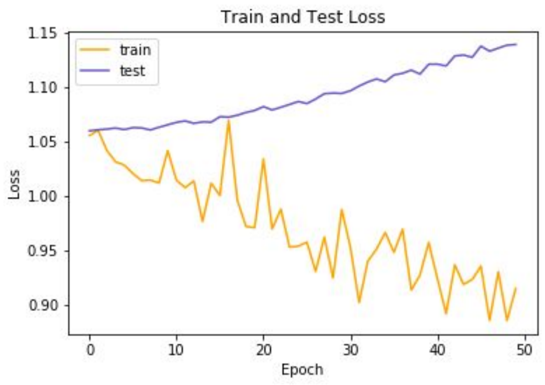
		True Labels		
Predicted Labels		*-1*	*0*	*1*
	-1	19,256	15,420	12,306
	0	25,118	31,896	24,263
	1	12,312	19,350	24,361

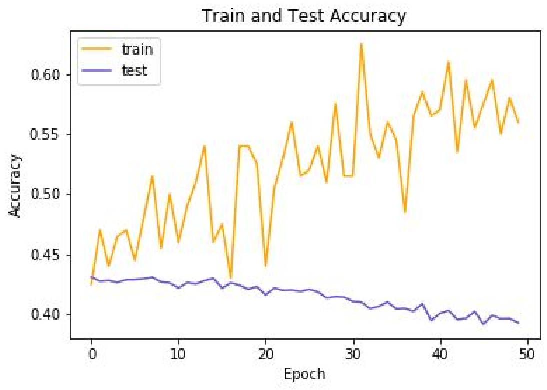
Confusion Matrix of FFNN Predictions with 1,000 Epochs, .0001 Learn Rate, 200 Batch Size

RNN	**1**	**2**	**3***	**4**	**5**	**6**	**7**	**8**
Features	25	25	25	25	25	25	5	25
Learn Rate	.0001	.0001	.0001	.001	.0001	.00005	.0001	.0001
Epochs	25	10	15	50	10	40	100	25
Batch Size	200	200	200	200	100	100	200	500
Neurons	500	500	250	250	250	250	250	250
Training Accuracy	.51076	.48246	.48951	.66202	.50078	.53510	.46842	.50133
Testing Accuracy	.41983	.43102	.43200	.38055	.43130	.41419	.42076	.43019

Results and Analysis summary for the RNN

Best Model (RNN)
The model that achieved the highest accuracy was an RNN with a learning rate of .0001, 15 epochs, 250 neurons, and batch size of 200. Below are plots of loss and accuracy for both train and test data.





Examining these graphs, it is clear that with the aforementioned parameter values, overfitting occurs beginning at about epoch 15. This is where the training accuracy continues to steadily increase and the testing accuracy begins to decrease. Increasing the number of neurons from 250 to 500 caused minimal, if any accuracy improvement, while decreasing to 100 caused a significant decrease in accuracy. Thus, we selected 250 for our final model as it maximized performance and minimized run time. Adjusting the batch size did not have much effect on the model. Finally, learning rates greater than our selected value of .0001, such as .001, caused overfitting to occur within the first few epochs while a learning rate of .00001 did not offer significant performance improvement.

Changing the architecture of the RNN by adding more layers or using different activation functions might slightly increase the performance of the model. That being said, the most likely way to significantly improve the models performance is through additional feature engineering or a restatement of the prediction target.