Stock Trader Pro with MLP Networks

ECE 457B: Computational Intelligence

Group 14

April 4, 2016

Table of Contents

Table of Figures

Abstract

Multi-Layer Perceptron (MLP) artificial neural networks are used in this project to try and determine optimal times to purchase or sell stocks. Historical data is mined for training the networks, with an 'expert' marking ideal buy and sell times in the data, based on available information from future states of the market. After training, the networks are run on further historical data in order to evaluate their performance. By analyzing single stocks it is possible to achieve better than random results, with significant improvements over random on downward trending stocks. The ultimate goal of this project is to provide a software tool to buy and sell stocks in weekly timeframes to make a profit for any user on commonly traded stocks.

The algorithm will develop two neural networks one for choosing whether or not to sell a stock; outputting 0 or 1, hold or sell classes, and one for choosing when to buy; outputting 0 or 1, hold or buy classes. The neural net's output perceptron will produce a value between 0 and 1 representing the algorithm's confidence to buy or sell the stock at the given price. First and second derivatives are taken to find the maximums, minimums, and inflection points to better track the price trends. These derivatives along with the normalized price are the three inputs into each neural network to help the algorithm trigger buys and sells.

The scope of this algorithm will initially focus on longer time periods (months) because historical data via yahoo finance are readily available while live data is only available weekdays from 9:30-5:00. In real high frequency trading as a small fish the margins taken by brokers turn out to be too large to result in a successful algorithm. Over months this margin taken by brokers becomes insignificant when investing a typical middle class savings account of $20,000.

Introduction

The stock market is a system set up to enable many people to invest in companies and share both the risks and rewards associated with these investments. This system is split up into several central exchanges, the top 3 in the world currently trading a combined volume of 3,981 [bn USD] per month. [1] Many have made and lost fortunes on this system and the allure of the potential earnings has resulted in many people working on algorithms to ensure profits on trades.

With the advent of networked computers, the stock exchanges to have become digitized and interconnected. The initial goal of this shift was simply to provide convenience for investors. Computerized trading did however immensely speed up how fast one could trade, made data regarding market trends much easier to collect and access and opened up the possibility of writing code that can make decisions about trades at timescales and precision the human mind cannot achieve. Many early trading algorithms sought to capitalize mostly on network and computer speed. By trading much quicker than a human can, this software can capitalize on small market variations and obtain the best possible bid-ask spreads. This type of algorithms are called High Frequency Trading and remain the predominant methodology in place today, albeit with some added intricacies. [2]

Due to the switch to computerized systems for stock trades, vast amounts of data is being generated regarding the historical value of stocks. Sifting through such large amounts of information is daunting, and oftentimes reviewers feel that there is no true pattern in it. Some feel that it really is just a volatile system with no internally predictable output. These qualities of having large amounts of data available and potentially containing patterns that are hard to classify makes this a problem well suited for an MLP approach. Having access to a large set of historical data means that both training and testing can be run on either large or small scales. It is theorized that if there exists discernible patterns within the datasets available, an MLP algorithm could pick up on them and be used to predict market trends.

Theoretical background

MLP networks are a statistical inference tool that can be used to classify sets of data. These networks are based on a high level view of how the neural system works in naturally evolved processing systems. They require training sets of data which include inputs of real, or expected data along with an indication of what their outputs should be for these training sets. Using this training data, it is possible to 'teach' the network how classify certain input patterns. Once trained, a network can be used on fresh data to help classify patterns.

The base unit of an MLP network is the perceptron. This neuron consists of a few components in the MLP context. Data is received as a sum of weighted value inputs and passed through the activation function which normalizes the result. The result is the value of the neuron which is passed towards other connected neurons. A diagram of this is shown in Figure 1 with three inputs and two outputs.

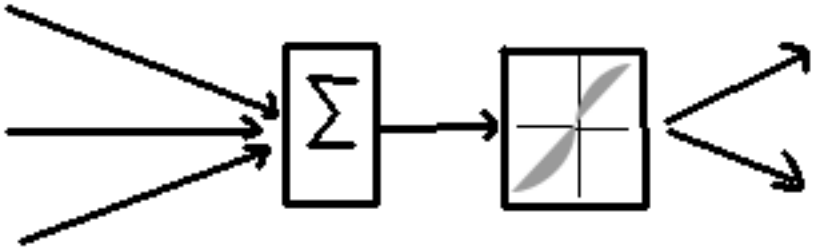

Figure 1: One perceptron unit

The perceptron is typically limited to linear tasks due to the single neuron [3]. In addition, the activation function for a perceptron is not a continuous operation:

$$f(x) = \begin{cases} 1, \sum w_i x_i + w_0 > 0 \\ 0, otherwise \end{cases}$$

By creating multiple perceptrons in parallel, nonlinear patterns can begin to be approximated with machine learning. However, the function needs to be continuous to allow for differentiation; the delta function would not mean much in this context. Instead a continuous function is used, typically a sigmoid (logistic) or a hyperbolic tan, with which a derivative exists continuously.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Conjoining multiple perceptrons in the same layer requires connecting each node in the previous layer to each node in the current layer, then each in the current node to all neurons in the next layer. The system will then consist of an input layer, a number of hidden layers, and an

output layer each with a certain number of neurons. An example of a multilayer perceptron network is shown simply as in Figure 2.

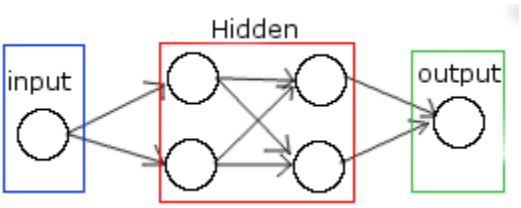

Figure 2: MLP simple network layout

A forward feeding neural network implies that the output is not fed back into the network. That is, the input nodes propagate data forward to the hidden layer(s), and then on to the output layer. When a value propagates, it performs similar to a perceptron by summing the product of weights & values and running it through a function to generate the output value of this node. The issue arises when trying to train this data because the error is found with the output data. This can be fixed by traversing this error backwards, hence the backpropagation algorithm is used.

The goal of the back-propagating algorithm is to reduce the error function of the whole network. Before the error can be found, a set of training input values and training outputs are configured. These will contain a number of input vectors and training output vectors where the length of the vectors equals the number of input & output nodes. The hidden layers between will initialize with small random weights on each connection. Then each training data input set is passed individually through the network to find the output weight. This data set is compared to the training outputs, and then to propagate the error the weights are updated backwards through the network.

The weight updates can be found by minimizing the error function E through gradient descent on each weight w [4]. This can be shown for E:

$$E(k) = \frac{1}{2}\sum_j \left\| y_j - f(x_j) \right\|^2$$

$$\nabla E = \left(\frac{\partial E}{\partial w_0}, \ldots, \frac{\partial E}{\partial w_n} \right)$$

Set η as the learning variable between [0, 1]. The weights are updated by the descending rule:

$$w_i += -\eta \nabla E(w_i)$$

This can be simplified by considering the gradient on the current test points. Taking the partial of E simplifies to and yields:

$$\frac{\partial E}{\partial w_i} = -\sum_j (y_j - f(x_j))x_{j,i}$$

So the weights can be updated iteratively if the current training point is considered. Each weight $w_{j,i}$ connecting output $f(x_j)$ to hidden node x_i is updated now:

$$w_{j,i} += +\eta \sum_j \left(y_j - f(x_j)\right) x_i$$

A common function, also used through this project, is the sigmoid activation function defined as $\sigma(x) = 1/(1+\exp(-x))$. The advantage to this is a continuous derivative that contains the original sigmoid for a simplified representation as:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Plugging this with the partial derivatives into the above partial of E equation, the update rule can simply be represented as:

$$w = w + \eta o_j(1 - o_j)\left(\sum_i w_{j,i}(y_i - o_i)\right)x$$

Where w is the weights matrix, oj is the output array of the network, yi is the training output and x is the vector of input nodes.

Proposed Solution

MLP

An MLP class was written in python meant to create, train, and use a multi-layer perceptron neural network that can be defined by the number of inputs, number of hidden layers, and number of hidden nodes per hidden layer. Each instance of an MLP class has 1 output neuron, outputting a value between 0 and 1 due to the characteristics of the sigmoid activation function used. Below is a plot of the activation function $\frac{1}{1+e^{-x}}$ within each neuron in the MLP that was created.

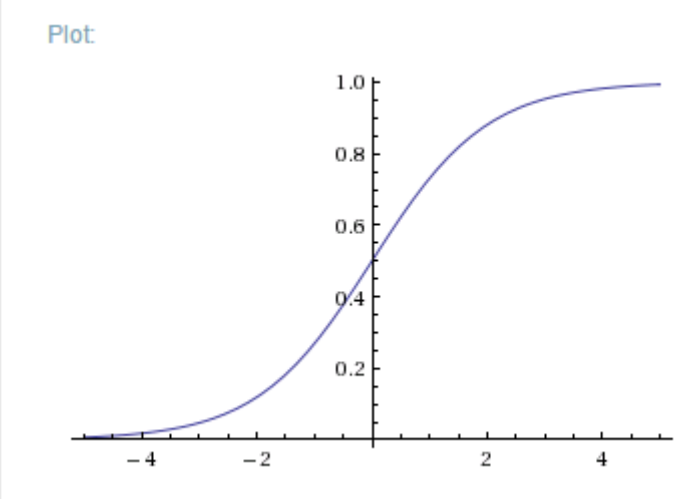

Figure 3: Sigmoid function [5]

We see that the output of this function approaches 1 and 0 asymptotically therefore its output range is (0,1) non inclusive. It is also worth noting the amount of input x that is required in order to output values close to 0 and 1; a input of +/-4 results in an output that is close to 0/1.

Initialization

The Multilayer Perceptron class is initialized by passing in the number of inputs to the network, the number of hidden layers, and the number of hidden neurons. These variables are stored in the MLP object for access in other methods. Based on the given variables the initialization method creates a weight matrix. The weight matrix is initialized to be NxN of zeros, where N is the number of neurons in the entire network. The python module numpy was used for the weight matrix manipulations. Below is an example of a neural network and the weight matrix that is created with 2 inputs, 2 hidden layers, 3 hidden neurons per layer, and 1 output.

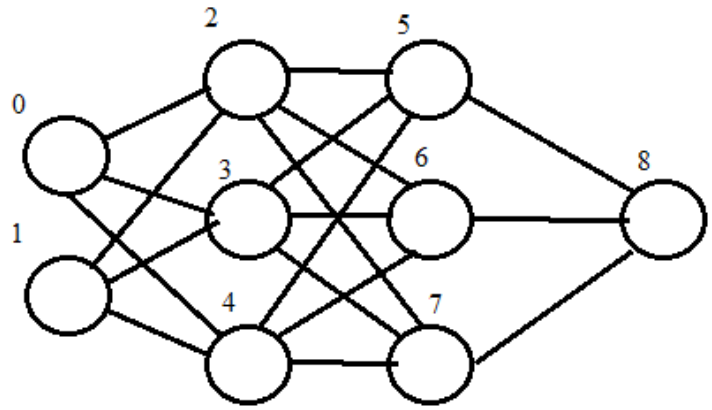

Figure 4: Neural Net Topology

```
[ 0, 0, 0, 0, 0, 0, 0, 0, 0]
[ 0, 0, 0, 0, 0, 0, 0, 0, 0]
[ W, W, 0, 0, 0, 0, 0, 0, 0]
[ W, W, 0, 0, 0, 0, 0, 0, 0]
[ W, W, 0, 0, 0, 0, 0, 0, 0]
[ 0, 0, W, W, W, 0, 0, 0, 0]
[ 0, 0, W, W, W, 0, 0, 0, 0]
[ 0, 0, W, W, W, 0, 0, 0, 0]
[ 0, 0, 0, 0, 0, W, W, W, 0]
```

Figure 5: Generated Weight Matrix

The algorithm to generate the neural net's weights begins by looping through the input layer and the first hidden layer. For each neuron in the first hidden layer we loop through all the input neurons and add a random weight at their indices. This develops the first 6 random weights in the first two columns of the matrix. The algorithm then takes the next hidden layer and loop through it with each neuron looping through each neuron in the previous hidden layer and placing a random weight at their indices. This section generates the weights in columns 3-5 representing the second layer's weights. The last part of the algorithm is to take the output neuron and loop through each neuron in the last hidden layer placing a random weight in the matrix at those indices. This generates the final weights in the last row. Once this algorithm finishes the initialization of the neural network is complete.

Train

The train method takes in the inputs and expected outputs as a list of lists, and it takes in the number of epoch training cycles. Locally in the method the learning rate, and maximum error are set. The number of input points to cycle through are found by taking the length of the list of lists. The train method is meant to run the desired number of points through the neural network for the desired number of epoch cycles in order to change the weights of the network so that it can better predict and classify future inputs.

Forward Propagation

Applying forward propagation to the neural network is done in 4 steps. First step is to iterate over all input nodes and assign the input value as their output. The second step is to iterate

over all neurons in the first hidden layer summing up all inputs from the input nodes to calculate the output of each neuron in the first hidden layer, with equation $O_i = sigmoid(\sum O_j W_{i,j})$. The third step contains 3 FOR loops, one to loop through the remaining hidden layers, a second to loop through the current layer's neurons, and a third to loop through the previous layer's neurons. Each neuron in the current layer has its output calculated by the same equation as before $O_i = sigmoid(\sum O_j W_{i,j})$ just applied to the previous hidden layer's neurons, not the input layer. The final step is to calculate the output of the single output neuron by calculating the sum of each neuron in the previous layer's output times its weight value connected to the output neuron.

Back Propagation

The first step of the backpropagation is to perform cumulative error calculations on the output using the equation $E = E + 0.5 * (Ref - Out)^2$. This is what we later compare to our max error term. Error signal calculations and weight updates are performed from the output, propagating layer by layer back to the inputs. The error signal form the output node is computed using $\delta_i = sigmoid'(\sum O_{out} W_{out,j}) * (Ref - Out)$. This error signal is then used to update the weights from the output neuron to the last layer of hidden neurons by following the equation $W_{i,j} = \eta \delta_i O_j$. The error signals for each neuron in the last hidden layer are calculated according to the equation $\delta_j = O_j * (1 - O_j) \delta out * W_{out,j}$. The error signals are calculated differently within the hidden layers because they are receiving feedback from all the error signals of the neurons in the layer in front of it. The error signal equation for this case takes the summation of deltas and weights and changes the equation to $\delta_j = O_j * (1 - O_j) * \sum \delta_{i * W_{i,j}}$. Once all weights have been updated in the neural network the cumulative error is compared to the maximum error, if it is greater the code cycles through another epoch cycle. Once the error is less than the defined max error the code breaks out of the loops and exits the "Train" method as a trained neural network.

Run

Run method of the MLP class takes in a list of lists representing the inputs to be run through the network. The inputs are propagated forward through the network having their outputs calculated according to the formula $O_i = sigmoid(\sum O_j W_{i,j})$. The output of the network

is stored in a list to represent the output for every input point. This list is returned by the method so that the calling function has access to the neural network's classification of the desired points. The output's value is between 0 and 1 and is taken to represent the network's confidence when buying or selling at a particular price point.

Stock Trader Pro

The Stock Trader Pro python file is the main program to buy and sell stocks. The trader was implemented such that it can only trade one stock at a time, buying all in or selling all out to keep things simple. The file requires a list of stock tickers, start and end dates for it to run the historical prices through the neural network and make trading choices. The file has a function to get history using the Yahoo Finance python module. This module returns a stock's daily close, volume and much more for every day in the start date to end date range. The get history function manipulates the data received from Yahoo Finance to return a dictionary object holding the date and date's price.

Initialization

The file is initialized by training two instances of neural networks from the MLP class to buy and sell stocks. One is trained with a list of points that the stock should always buy at, points such as troughs where the price has reached a new low. The other MLP is trained with a list of points where it should always sell, such as peaks where the current price is the highest the stock has ever been. These objects are passed into the main function along with the stock tickers one at a time.

Wallet Class

To manage the amount of money earned by the trader a wallet class has been made. The class keeps track of the amount of cash in the bank and the amount of shares owned in a stock as well as the price per share purchased at. When the trader wants to buy a stock the current price is passed into the wallet's buy method. The buy method calculates the most shares that can be bought at the current price accounting for the $9.99 commission, subtracts that value from the

cash account and adds the stock value to the stock account. The sell method takes in the current price, multiplies it by the number of shares owned, subtracts that number from the stock account and adds it to the cash account minus the commission.

The wallet class also has methods to calculate the current profit given current price, calculate current value of its shares at the current price, and normalize the profit to a percentage of the original investment.

Main

The main loop in the file takes in one stock ticker and history range in at a time to work properly. The price history is grabbed using the get history function, then the prices and dates throughout that history are plotted using the matplolib python module. The prices are then passed into the trading algorithm where buys and sells take place. The points where a buy occurs is replotted on the graph with a red square, while the points where a sell occurs is replotted as a green diamond.

Trade Function

The trade function is what actually uses the trained neural network. This function takes in all prices over the history specified and iterates through the list in chronological order. A buffer holds the most recent 70 points which is used to calculate the rate of change and second rate of change for each new point. The code also keeps track of the highest and lowest prices it has seen as it loops through the historical prices. These highs and lows need to be reset every 60 points in order to keep the high and low range reflective of the local price activity. The price is normalized based on this high-low price range using the equation $Normalized\ Price = (P - L)/(H - L)$. In the case where all money is in cash the normalized price, rate of change, and second rate of change are fed into the trained buying neural network to determine what the buy confidence is for this point. If the buy confidence is greater than our threshold we buy the stock putting as much money as we can into it, the point is also replotted to be a red square on the graph. In the case that all money is in stock the normalized price, rate of change, and second rate of change are fed into the sell neural network to determine the sell confidence value. If the sell confidence is

greater than our threshold then all stock is sold and the point is replotted as a green diamond on the plot. Once the function has iterated over all prices chronologically the plot is ready to display and will show all points where the program bought or sold as well as display the total profit made over the entire history.

Results

The program was run over 7 different stocks using prices from December 3rd 2012 to December 31st 2015 in order to evaluate its performance on different market trends. The performance is compared to a random stock trader to decide whether it has met our project goal or not. The random stock trader uses the same framework, but it decides to randomly buy sell or hold every 10 price points rather than use a neural network. The first comparison is between the developed MLP stock trader and a random algorithm on an upward trending stock, Google.

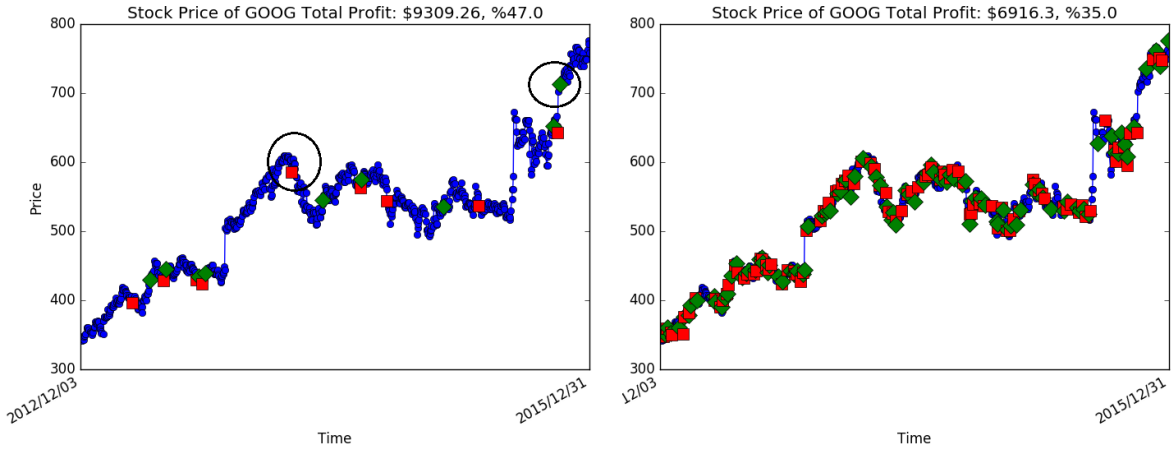

Figure 6: GOOG with MLP (left) vs Random (right)

The random algorithm has a great chance of making money since future points tend to be larger than previous, and just by randomly buying and selling it is possible to make money over time. This is what is seen in the plot to the right, every 10 price points the algorithm randomly decides to buy sell or hold, and more often than not when it can sell it sells at a profit. The algorithm makes 35% over the 3 year span. The MLP trading algorithm trades much less frequently and tries to capitalize on buying low and selling high with the stock trends. This algorithm makes 47% over the 3 year span, outperforming the random algorithm by 13% profit

points. The algorithm is far from optimal as it often prematurely buys and sells when it sees peaks and troughs as highlighted by the circles on the plot. Opportunity to make more profit is missed by the algorithm buying too soon and selling too early.

The next trend that the MLP and random algorithms will be compared against each other with is a downward trend using the IBM stock. The history is from December 3rd 2012 to December 31st 2015, same range as the previous comparison.

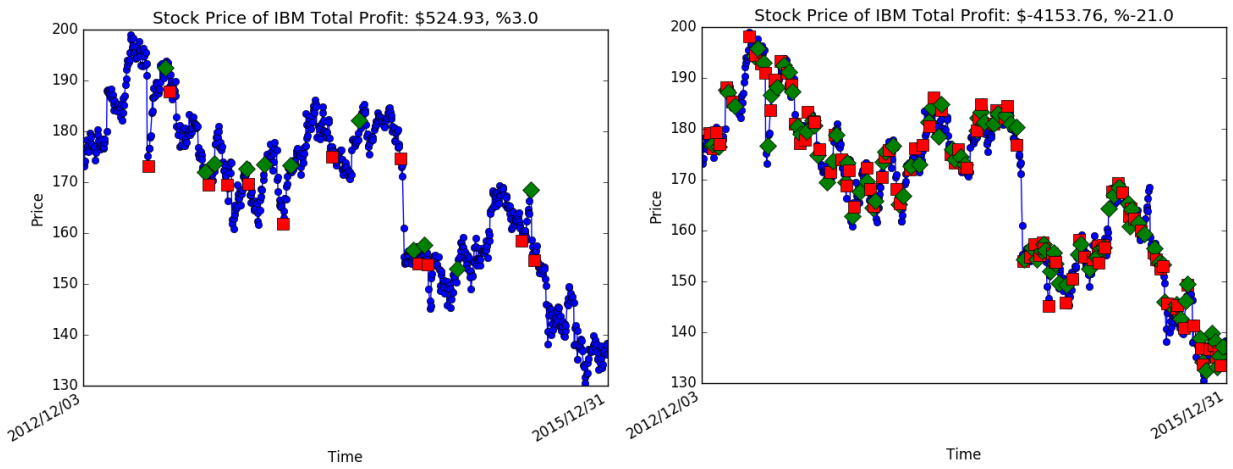

Figure 7: IBM with MLP (left) vs Random (right)

The random algorithm is losing 21% over the 3 years on the IBM stock. The stock is downward trending with a few points of large losses. Over the long run a random algorithm ends up losing money because more often than not the future price points are less than where the program bought. The MLP algorithm fares much better on this downward trending stock, making 3% over the 3 year period. Certain buy and sell points are very well timed by the neural network, but those that are not are what keep this algorithm from maximizing profits. This is a difficult task however, because the buys and sells do occur at peaks/troughs but they are dwarfed by better peaks/troughs soon after.

The final trend to be compared and analyzed is a net stagnate market trend. From December 3rd 2012 to December 31st 2015 TD stock went from $36 to $39 with a rise and fall in between, effectively the same value beginning to end. This is shown in Figure 8.

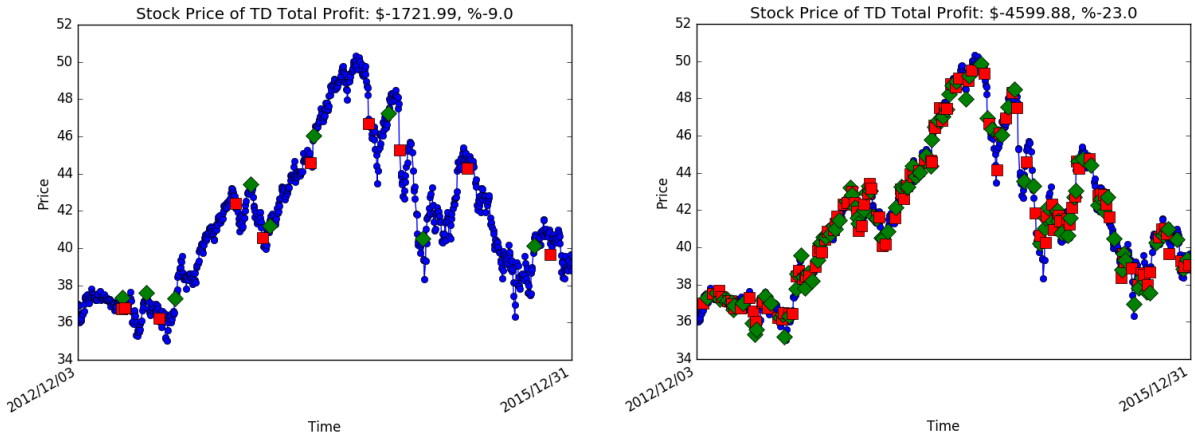

Figure 8: TD for MLP vs Random

The random trading algorithm loses 23% throughout this time frame even though the stock stays around the same value. The algorithm loses a lot of the money during the downward trend in the second half buying and selling at bad times. The MLP algorithm is not perfect at this point, this stretch of TD stock history has the algorithm generating a loss of 9%. This is due to the algorithm choosing minor peaks to buy and sell when soon after there are better points to buy or sell.

There were a total of 7 stocks tested using both the MLP Algorithm and the Random Algorithm. Below is a complete chart of the results obtained from these tests, comparing the profits of each algorithm for each stock. The history for each stock are all the same from December 3rd 2012 to December 31st 2015 to keep data as consistent as they can be for the nature of this experiment. Both trading algorithms take into account a $9.99 fee when buying and selling resulting in $19.98 fee for a buy/sell cycle. This is a typical commission that is charged to a brokerage account when the average person trades stocks.

Stock	Trend	MLP Algorithm Profit	Random Algorithm Profit	Difference (MLP Profit - Random Profit)
Google	Up	47%	35%	12%
Tesla	Up	68%	25%	43%
AAPL	Up	27%	12%	15%

TD	Flat	-9%	-23%	14%
IBM	Down	3%	-21%	24%
GM	Up	28%	8%	20%
Boeing	Up	59%	29%	30%
AVERAGE		31.86%	9%	Difference of means = 22.86%
Standard Deviation		0.283221	0.233289	

Figure 9: Complete results comparison

T-Test Calculations

To determine how much the two algorithm's profit means differ from each other a t test is done. A t test takes in parameters that are a function of the sample means, sample standard deviations and outputs a confidence interval. The confidence interval will be a supporting statistic to how much confidence there is in the MLP trading algorithm being better than the random trading algorithm. The null hypothesis is that the means are the same, we will use an alpha value of 0.15 since this is a relatively small sample size, this alph requires a confidence interval of at least 85% for the two means to be significantly different from each other and reject the null hypothesis.

Number of samples (n) for each algorithm = 7

Difference of means = 0.3186 - 0.09 = 0.2286

$$MSE = \frac{s_1^2+s_2^2}{2}, S_{m1-m2} = \sqrt{\frac{2MSE}{n}} = 0.138687$$

MSE = s1 2+ s222, Sm1-m2 = 2MSEn= 0.138687

Confidence Interval = 87.5%

Therefore we can reject the null hypothesis and say with 87.5% confidence that the MLP algorithm is better than the random trading algorithm.

After analyzing the data from all 7 mutually tested stocks over the same history it is clear that the developed MLP neural network trading algorithm is superior to a randomly trading algorithm. In every instance of head to head comparison the developed network of this project makes more profit that the random algorithm. The MLP algorithm also shows encouraging results on downward trending stocks. The algorithm makes 3% on IBM stock while the random trade algorithm takes heavy losses at -21%. From the t distribution calculations using the comparative data it is known that the MLP trading algorithm is significantly different than the random trading algorithm with an 87.5% confidence. The MLP algorithm is still far from perfect and would not be deemed fit for live trading with real money at this point, but these initial results are very encouraging in terms of the potential of the algorithm

Future Improvements

To take the MLP trading algorithm developed for this project to the next level a significant increase in training and testing of the network must be done on a large sample of different company's stocks to ensure the solution is robust enough for live trading. The current algorithm is trained for only a handful of expertly picked points, this must be expanded to hundreds even thousands of points in order to teach the algorithm how to react for any scenario. The buy/sell confidence thresholds used in the algorithm can be changed dynamically to adapt to different scenarios as well. For example if there is a massive increase in the stock price and the algorithm currently own the stock, the confidence threshold should increase requiring the neural network to be near 100% confident to sell the stock. Similarly if a stock is crashing and the algorithm is looking to buy then it should increase the confidence threshold until the price levels out.

The prospect of unsupervised learning is another point for possible expansion. Instead of relying on so-called experts feeding correct buy and sell points from the stock market into the algorithm, why not have it figure out how to make the most profit on its own? A genetic algorithm may be a great approach to developing the next stock trading algorithm. The genetic algorithm could use the same framework built in this project using a fitness function of maximum profit to grade itself. The downside of this method compared to the MLP neural net is the computation time. The MLP has a fixed computation time based on the number of training points and number of epoch cycles. A genetic algorithm would need to randomly generate many different potential algorithms and evaluate them all in cycles, removing the losers until one stands.

Conclusion

In conclusion, it was found that utilizing a neural network to estimate trends in a stock is an effective method of earning a profit, with some significant drawbacks. The multi-layer perceptron stocktrader algorithm made a profit on 6 out of 7 stocks that were run, losing money on TD stock but to a lesser degree of the random trading algorithm. The goal for this stage of the project is to develop a trading algorithm that outperforms a random trading algorithm with 85% confidence. From the complete results it was found that the MLP trading algorithm is better than the random trading algorithm with 87.5% confidence.

Although the current solution has met the immediate goal it is but the first step in it becoming a fully functioning live trading algorithm. Ultimately the algorithm needs to make a profit on any commonly traded stock, and it currently tends to lose money on stocks that have sharp declines in price. More training and testing data as well as additional fundamental features are needed to increase the neural network's capabilities to the point where the algorithm no longer produces a loss in the long run.

Computational intelligence is beginning to reshape the technological world by giving computers the tools to think more like a biological system in terms of logic and reasoning. This is currently exploited in the realm of high frequency trading, but this is performed by large trading firms with the budget for the software and incredibly fast computers. The algorithm developed as part of this project is meant for the average investor who simply wants a reliable and safe method for investment that yields more return than a savings account. The first step was successful, perhaps someday the algorithm will become robust enough to be used by the masses.

References

[1] “Monthly Reports”. World Federation of Exchanges. Accessed April 1 2016 from http://www.world-exchanges.org/home/index.php/statistics/monthly-reports

[2] Investopedia Staff. “What is High Frequency Trading?”. Investopedia. Accessed April 2, 2016 from http://www.investopedia.com/ask/answers/09/high-frequency-trading.asp

[3] Riedmiller, M. “Machine Learning: Multi Layer Perceptrons”. Albert-Ludwigs-University Freiburg, DE. Accessed April 4th 2016 from http://ml.informatik.uni-freiburg.de/_media/teaching/ss10/05_mlps.printer.pdf

[4] Kun, Jeremy. “Neural Networks and the Backpropagation Algorithm“. Accessed April 4th, 2016 from http://jeremykun.com/2012/12/09/neural-networks-and-backpropagation/

[5] Sigmoid function. Wolfram Alpha. Accessed April 4th 2016 from https://www.wolframalpha.com/input/?i=1%2F%281%2Be^-x%29+from+-5+to+5