Real-Time Day Trading

Machine Learning Engineer Nanodegree

Josh Boltz
September 26, 2016

I. Definition

Project Overview:

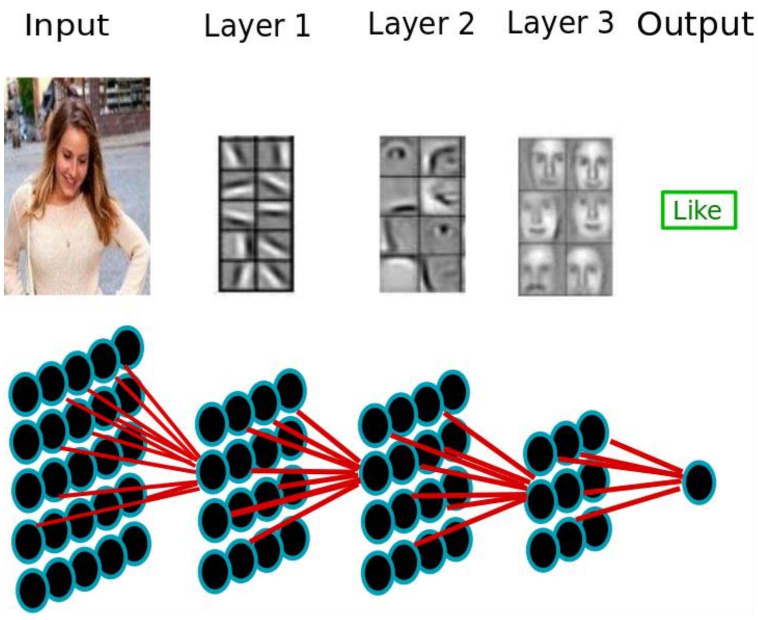
This project is an attempt to take in stock information and output decisions of whether to buy, short, or stay a stock at minute intervals throughout a trading day and in the process, profit from these decisions. This project will take intraday data on a minute by minute basis where we have closing prices for a stock at each of these intervals. We will predict whether a stock price will rise or fall between intervals and make decisions based on these predictions. So if we think that the stock price of Apple will rise within the next minute, we will buy at the current price and then look at what actually happened to the stock price at the next interval. Based on whether our prediction was correct and the stock price rose, determines whether that decision is correct. Overall we will take all of our decision outcomes throughout a day and if we are profitable for that day, that day is considered successful.

This day trading solution will use a subset of a well know machine learning sector, called deep learning. Deep learning can be thought of like how your brain operates. For example, say you are walking down the street and you look up and recognize somebody you know. There's several things that happen between looking up and recognizing your friend.

The first is your brain takes in information (Input), and based on your friend's facial structure, several things will happen. First after taking in the information, based on several factors in your friends face, like nose length, hair color, eye color, skin color, weight, and so forth, your brain will take all of these factors and process them.

For simplicity sakes, imagine for each of these factors, you have what's called an input neuron for that particular feature. Think of a neuron like a mini-computer that takes that feature and based on if it's seen that feature before, it outputs YES and if not NO. These feature outputs are then sent to even more neurons to process groups of features together. For this, think about grouping parts of the face, like saying yes I do recognize that that is brown hair but how about his hair combined with his eyes? Have I seen that feature before? This feature then gives you also a YES or NO value and this continues until you've processed his entire face.

Now if enough neurons have given you a YES value, you'll trip the "hey that's my friend based on these characteristics" output neuron. Not every neuron has to be correctly tripped to recognize your friend, just enough to make a qualified prediction that is in fact him. For example, he could be wearing makeup that lightens his skin color that tripped up one of your features, but since a high enough percentage was tripped, you understand its very likely that’s him.



This is an extremely simplified model of how deep learning works but is satisfactory for your understanding of the algorithm we used here. Our algorithm isn’t exactly the same as the one described but is a subset of this type. We will utilize our algorithm to hopefully make profitable trades on a daily basis.

For our specific purposes our inputs would be stock information, rather than facial features and one big difference compared to the facial recognition example, is we can use info from different time intervals to help our prediction. For our example, we can use what happened in the past 5 minutes to help in our prediction process rather than just using the stock information at present.

The project dataset was acquired from a small provider source that provided free intraday data dating back roughly 4 years that you could acquire for any 100 or so stocks they provided. The data was not perfect and was missing several days’ worth of data but overall 99% of the data was accurate. Considering most intraday data providers charge upwards of $500 dollars for the data we got for free, it was well worth the small inaccuracies. After historical data, we then retrieve real-time stock data from a Google Finance server that we query, that allows us to get the information we need for the stocks we’re are predicting for.

Problem Statement:

The problem that I’m trying to solve is to create a reliable, computationally inexpensive, memory efficient solution for predicting stock prices on a minute by minute basis for a day trading solution that would ensure you making a profit over time. This will attempt to be solved using a machine learning algorithm called Real-Time Recurrent Learning(RTRL). If we are successful, on a daily basis we will be making the correct decision to buy, short, or stay a stock so that we end up profitable each day even when the market is volatile.

Metrics:

To measure our performance in this project, we will look at a day's worth of trades and base our performance on if we were profitable. During the day, we have 390 time steps representing a normal 6.5-hour trading day. Since we work on a minute by minute level, this gives us 390 options in a day. Each prediction will give us what it thinks the return value (current price / previous price – 1) of the stock will be at the next time step. We then multiply the return value by the stock price to get the dollar value increase or decrease. We then multiply that dollar value change by 1000 in our example to show the effects of buying or shorting 1000 stocks at once.

Buying means we buy stock if we think that the return value at the next time step will be positive meaning the price of the stock has risen. Shorting means we sell shares of the stock to an investor at the current price and buy it back at the next time step price. If the price drops, that means we gain money since we sold it at a higher price than when we bought it back at. Finally staying means we simply don't take any action if the return value prediction is zero.

To measure our performance for the day, we will add up all of the cash value we gained and also subtract the value we lost. This isn't a perfect setup as this disregards the fact you need an extremely difficult-to-get license to short stock but it is a simple metric that gives us a good idea of the performance of the algorithm nonetheless.

II. Analysis

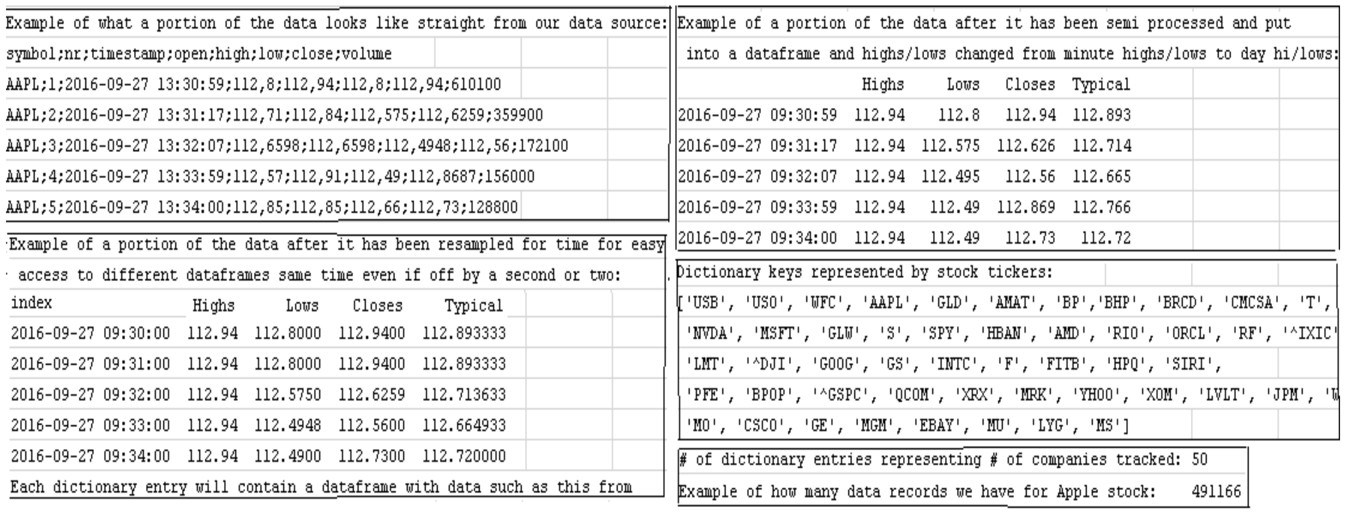
Data Exploration:

For our dataset, a few things were crucial. First, we needed intraday data which is difficult to freely obtain, rather than daily close data which is freely available at several sources. This first problem was overcome by stumbling across a small British data provider that had roughly 5 years of intraday data at minute intervals for roughly a hundred different stocks, 50 of which were used in my project. This dataset contained roughly 390 closing prices per stock day for each company, as there is 390 stock minutes in a stock day ignoring after-hours trading.

The data was not perfect and there was roughly 20 stock days' total that were missing from the data but considering our 5 years of data, it amounted to only about 1% of the data, and was fixed by removing the days from the training.

After our historical data has been processed, we use real-time stock data acquired from a Google Finance server to retrieve stock prices for the stocks we need in a fast efficient way. We then can use this to process the data and make predictions in real-time so this project can be taken from a theoretical (I could make money on this if I could get real time data) to being able to make decisions in a fast enough fashion to reliably make decisions.

Exploratory Visualization:



```
Example of what a portion of the data looks like straight from our data source:
symbol;nr;timestamp;open;high;low;close;volume
AAPL;1;2016-09-27 13:30:59;112,8;112,94;112,8;112,94;610100
AAPL;2;2016-09-27 13:31:17;112,71;112,84;112,575;112,6259;359900
AAPL;3;2016-09-27 13:32:07;112,6598;112,6598;112,4948;112,56;172100
AAPL;4;2016-09-27 13:33:59;112,57;112,91;112,49;112,8687;156000
AAPL;5;2016-09-27 13:34:00;112,85;112,85;112,66;112,73;128800
```

```
Example of a portion of the data after it has been semi processed and put
 into a dataframe and highs/lows changed from minute highs/lows to day hi/lows:
                     Highs     Lows   Closes  Typical
2016-09-27 09:30:59  112.94     112.8   112.94  112.893
2016-09-27 09:31:17  112.94   112.575  112.626  112.714
2016-09-27 09:32:07  112.94   112.495   112.56  112.665
2016-09-27 09:33:59  112.94    112.49  112.869  112.766
2016-09-27 09:34:00  112.94    112.49   112.73   112.72
```

```
Example of a portion of the data after it has been resampled for time for easy
 access to different dataframes same time even if off by a second or two:
index                Highs     Lows    Closes    Typical
2016-09-27 09:30:00  112.94  112.8000  112.9400  112.893333
2016-09-27 09:31:00  112.94  112.8000  112.9400  112.893333
2016-09-27 09:32:00  112.94  112.5750  112.6259  112.713633
2016-09-27 09:33:00  112.94  112.4948  112.5600  112.664933
2016-09-27 09:34:00  112.94  112.4900  112.7300  112.720000
Each dictionary entry will contain a dataframe with data such as this from
```

```
Dictionary keys represented by stock tickers:
['USB', 'USO', 'WFC', 'AAPL', 'GLD', 'AMAT', 'BP','BHP', 'BRCD', 'CMCSA', 'T',
 'NVDA', 'MSFT', 'GLW', 'S', 'SPY', 'HBAN', 'AMD', 'RIO', 'ORCL', 'RF', '^IXIC'
 'LMT', '^DJI', 'GOOG', 'GS', 'INTC', 'F', 'FITB', 'HPQ', 'SIRI',
 'PFE', 'BPOP', '^GSPC', 'QCOM', 'XRX', 'MRK', 'YHOO', 'XOM', 'LVLT', 'JPM', 'W
 'MO', 'CSCO', 'GE', 'MGM', 'EBAY', 'MU', 'LYG', 'MS']
```

```
# of dictionary entries representing # of companies tracked: 50
Example of how many data records we have for Apple stock:    491166
```

(Above is an example of the data we retrieve that is then processed step by step, and the amount of data records we retrieve, as well as how many companies we can track)

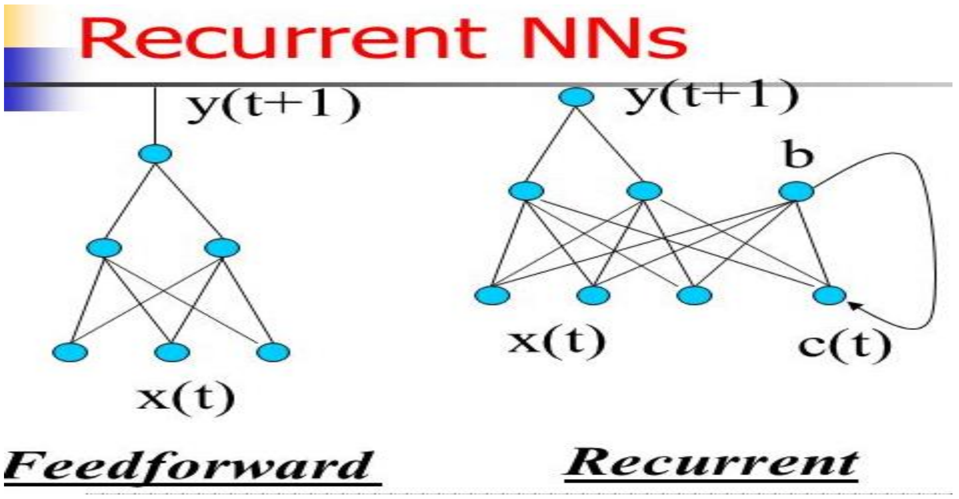
Above we have an example of 5 different information sets. The first (top left), is an example of a portion of the historical data that we retrieve straight from our data source. The data we retrieve is in semi-colon separated text format. After the first line which represents what each item on each line represents. These records contain the symbol, row number, timestamp, opening price, minute high, minute low, closing price, and volume traded during that interval. The second set (top right), is an example of our retrieved data partially processed and put into a dataframe. We only retrieve the data we need for each record and use the timestamp as our index. The third set (bottom left), is our data after having the timestamps resampled.

This is done so that we can call multiple dataframes timestamp and not have to deal with the second or two difference between there times. The fourth set (bottom middle), shows all the stock tickers (companies/indexes/etfs/natural-resources) we will be tracking with this program. And finally the last set shows two items: the number of entries we're are tracking (50), and the number of records that are in our Apple dataframe (491,166) to show generally how much information we have each company. This is shown in particular, to show how much information there is to process for even a single company.

Finally, this is an example of the data that we will use to create all of our indicators and is the base for this entire project. The reason this was shown in particular was because of the importance of this base information and how it's used throughout to create indicators that help us predict stock outcomes.

Algorithm Overview:

Real-Time recurrent learning is a subset of deep learning that is used mostly for sequence processing. You can find examples of this algorithm in areas that require time ordered data like for our project in stock prediction. Therefore, due to the sequential nature of this algorithm, it's used for ordered data and not for unordered data. While most of the leading hedge funds don't release their machine learning algorithms to the public, there widely known to use algorithms similar to this.



Due to the very specific needs of hedge fund firms, a time series algorithm for day trading is critical due to several things.

First is that they make nanosecond trades, which is to say, they take in some sort of stock information and need to process it quickly and efficiently and make a decision not only based on this new data but on previous data information in combination. That's what makes this algorithm so effective at what it does. Its real-time nature allows it to take in information in real-time and also process previous outputs with the inputs at the same time. This means your prediction could be based off not only, say, Apple releasing information about the number of jobs it picked up last month, but also what happened with their financial report 10 minutes ago. This is possible because this is a recurrent algorithm, rather than a feed-forward algorithm. Feed forward allows only one-way processing, from input to output, there's no feedback nature possible to process on inputs and outputs at the same time that's crucial with stock prediction.

The second reason an algorithm like this is so crucial to hedge funds is because how quickly you need to process mountains of data at once. The relatively quick processing that recurrent algorithms allow in real time is what makes it possible to trade in the nanoseconds. Because it can process one data step at a time in real-time, it allows you to continuously update your model and only have to process the data you need at a time and connect the network so that information can flow in as it comes and output flows out with little computation and memory cost.

During the prediction process, this algorithm will compute the error gradient and update weights at each processing point. The error gradient in our project is based using mean squared error, which means, at each time point we take the difference between the predicted output and the actual outputs squared value and update the mean of this over the whole time series. To minimize the error, gradient descent changes each weight in proportion to the derivative of the error based on that weight.

Techniques:

Using our real-time recurrent learning algorithm, we train and test on historical data and then test further using current real time data that we retrieve from a Google finance server that provides us this real time intraday data. For our project we use a modified version of an algorithm that was setup by a creator cited below written in python. I modified it slightly due to two memory leaks, and use of a few slower numpy methods compared to its scipy alternative that would both cut down on the amount of data you could process at once and the timing it took to process this data. I also chose to use mean squared error rather than mean absolute error because mean squared error punishes larger errors more which in stock trading is needed so you don't end up predicting 0 at each interval so your overall error is lower since on average stocks will generally not change drastically during a day. However, punishing larger errors forces us to predict positive and negative values. Finally, I added time constraints to ensure that if it wasn't finding an acceptable weight modification that decreased error in a timely manner to return what it had. The modifications allowed for more input data, and still ran faster.

For the main input parameters, you could easily change the number of parameters used based on your specific needs as well as the amount of data processed at once. The functions created were made with ease of use in mind and allowed for easy manipulation of these parameters and is well documented for the readers within the code.

The parameters we use can be adjusted as you like, and the only parameters used in training that aren't typically changed on a regular basis are the k_max parameter which is the number of times we cycle through the data if we don't hit a gradient plateau before the end, which we set at 10 which is fine for our purposes. It can be higher but due to the fact that when training on historical data, we're training on about 25000 records at a time with about 500 indicator values being used at each time step, many more cycles would add more computation cost.

The next one not typically changed but noted here for you if necessary is the E_stop which is the mean squared error at which we should stop if past that point representing the error is low enough to constitute stopping the calculation early. The other parameters are the dampfac, and dampconst which are the damping factor and constant respectively. These are used to change the amount we alter the weights during each cycle with larger weight alterations during earlier cycles gradually becoming smaller and smaller as the error becomes smaller and smaller and we need to adjust the weights in a smaller and smaller manner.

Those parameters are used in conjunction with the indicators we chose to predict with, where each input value would be an indicator/period-length so we could use anywhere from a single indicator/period-length up to hundreds of indicator/period-lengths at a time. For our example, we use 474 inputs representing 474 indicators from a variety of sources. The main source is the stock we're are predicting on which we take 252 indicators from. The other's we take from what I've called helper stocks (they're actually Exchange-Traded-Funds/Indexes/Natural-Resources but called stocks from this point on for brevity sakes). These helper stocks consisting of 6 stocks allow for information you couldn't glean solely from our Apple stock like how the Nasdaq and S&P500 Indexes are doing, or how Gold and Oil are trading. They help in the prediction because indexes in particular constitute how the entire market is doing, not just a single stock and can account for differences you couldn't glean just from a single stock.

Benchmark:

If we predict that a stock is going to lose 2% of its value within the next time step, the correct decision would be to short the stock, while the opposite goes if it was predicted to gain 2% in value, in which case buying is the best solution. If it’s predicted to stay even, the best option is to stay safe and not do anything.

Based on this, if we predict the right course of action, like shorting when we think the price will drop and buying when we think the price will go up, we gain profit, while making the wrong decision will decrease our overall value. And at the end of the day we should be profitable. That being said, our benchmark will be to obtain the overall profitability or unprofitability for the day based on what how much money we gained or lost. If we are profitable, we have achieved success, with the more profitable we are the better.

III. Methodology

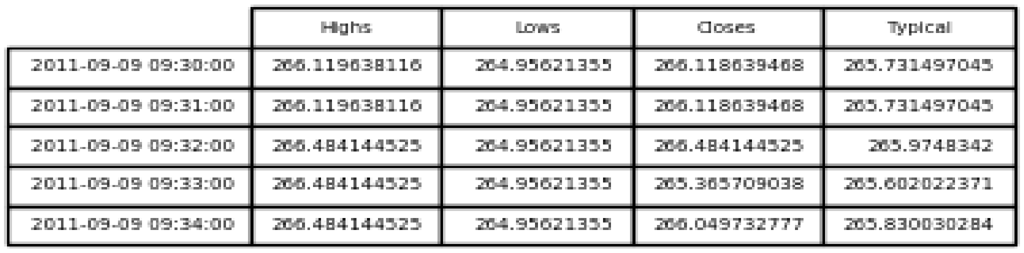
Data Preprocessing:

After obtaining our historical closing prices from our data source, there were many things we needed to do with this data to preprocess it.

First we resampled the data to the nearest minute. This was done so you could easily use another stock to help predict for a different stock. For example, using the S&P500 index to help us predict Apple stock. This resampling was required because there were time differences, typically a second or two, between when they individually were retrieved by the sources data source and it allowed for easier time matching between stocks.

After this resampling, we then converted minute highs and lows to daily highs and lows starting at the beginning of each day and ending at the end of the stock day. For example, we would start the day with the same high value and low value, and as the day progressed, the high and low value difference would become more and more. We created these values because many stock indicators (which will be discussed below) use high and low values to make their indicator values and I found day highs and lows were more informational than minute highs and lows.

The other value we would create was called the typical price value which was the high + low + closing values / 3. This was also used extensively in some indicators. Finally, we grouped these high/low/closing/typical values for each company into a dictionary where each key was the company stock ticker, and the value was a dictionary entry with several years’ worth of highs/lows/closings/typicals in a dataframe format for easy processing and readability purposes. The next step was using these highs/lows/closes/typicals to create our indicators.



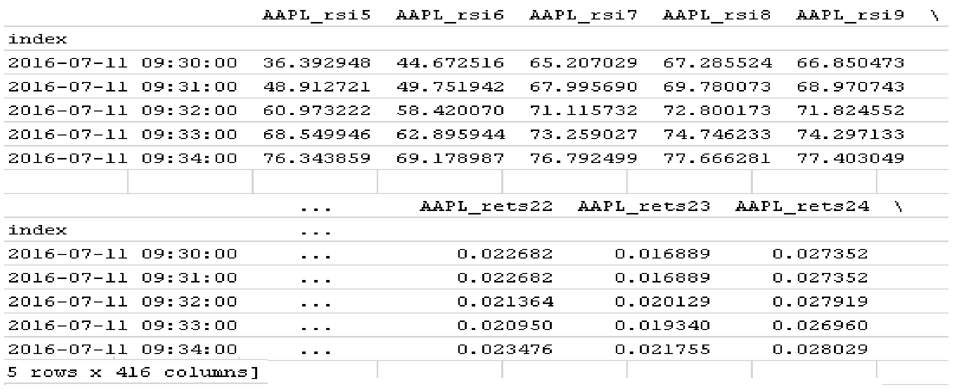
	Highs	Lows	Closes	Typical
2011-09-09 09:30:00	266.119638116	264.95621355	266.118639468	265.731497045
2011-09-09 09:31:00	266.119638116	264.95621355	266.118639468	265.731497045
2011-09-09 09:32:00	266.484144525	264.95621355	266.484144525	265.9748342
2011-09-09 09:33:00	266.484144525	264.95621355	265.365709038	265.602022371
2011-09-09 09:34:00	266.484144525	264.95621355	266.049732777	265.830030284

(Above is an example of 5 minutes' worth of data for a certain company)

Stock indicators are functions used heavily in trading that help predict a stock price. For example, a simpler indicator is, for example, the Simple Moving Average which is the average closing price within a certain time frame from that point in time. The time frame could be 10 minutes, or could be 10 months based on when you were trying to predict for. The smaller the time frame, typically means the sooner your prediction is for but is not the same for all indicators.

Overall we used 13 different indicators, with the last indicator being return value. For each of our indicators, we would have 32 different period lengths, so we could use multiple different period calculations at once because not all indicators had the same best indicator length to predict a stock value. Some were best with longer periods, while others were best with shorter periods even if we were predicting for the same time step. The 32 different period lengths allowed for a range of predictions and training variations so you could use one or many different period lengths at once as well as predict for different return value lengths like predicting 1 minute into the future or predicting 1 day into the future using a variety of period lengths for each indicator.

After we created our historical indicator values for each stock using the provided functions, we then would store all the indicators for each company in a dataframe where each column represents an indicator/period-length value and each row represented all the indicator values at a certain time step. Because there were 13 indicators with 32 period lengths for each, that meant 416 total indicator values at each time step for each company. However, we only use a fraction of these in the training algorithm due to memory and computation costs, but it allows us to train on all kinds of data depending on our needs.



```
                     AAPL_rsi5  AAPL_rsi6  AAPL_rsi7  AAPL_rsi8  AAPL_rsi9  \
index
2016-07-11 09:30:00  36.392948  44.672516  65.207029  67.285524  66.850473
2016-07-11 09:31:00  48.912721  49.751942  67.995690  69.780073  68.970743
2016-07-11 09:32:00  60.973222  58.420070  71.115732  72.800173  71.824552
2016-07-11 09:33:00  68.549946  62.895944  73.259027  74.746233  74.297133
2016-07-11 09:34:00  76.343859  69.178987  76.792499  77.666281  77.403049

                        ...     AAPL_rets22  AAPL_rets23  AAPL_rets24  \
index                   ...
2016-07-11 09:30:00     ...        0.022682     0.016889     0.027352
2016-07-11 09:31:00     ...        0.022682     0.016889     0.027352
2016-07-11 09:32:00     ...        0.021364     0.020129     0.027919
2016-07-11 09:33:00     ...        0.020950     0.019340     0.026960
2016-07-11 09:34:00     ...        0.023476     0.021755     0.028029
5 rows x 416 columns]
```

(An example indicator dataframe consisting of 5 time steps worth of data where each indicator is noted by the stock ticker, followed by an underscore, then the indicator abbreviation, followed by the period number (0-31))

These dataframes were then cut up into 15 different pieces due to the size of the data that it would take up otherwise (about 3 GB per company). We would cut the dataframes at certain dates so each dataframe consisted of the same date ranges.

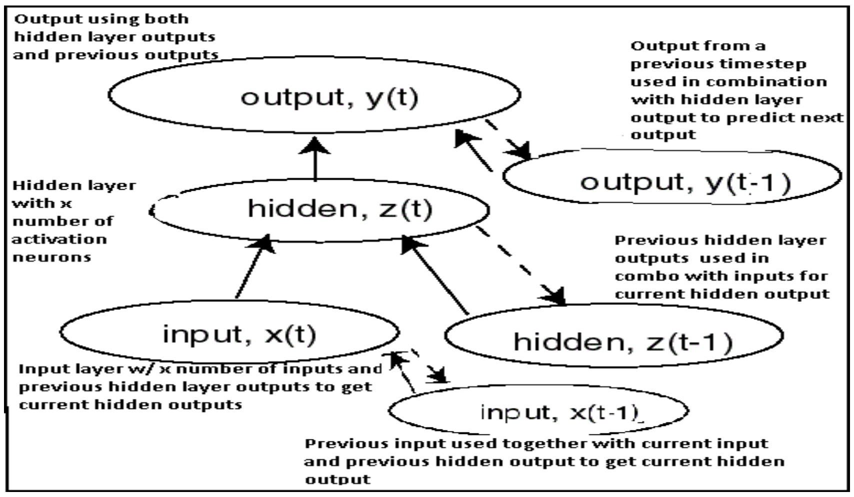
Implementation:

After setting up all of our data and preprocessing it so you could train with historical data, we now come to the issue of implementing the preprocessed data to train, and use the trained network to predict for new data. We would first train our network using our 15 different indicator dataframes for each company (which again, is just a single dataframe consisting of 416 indicators with values dating from roughly 3 years ago to present that is just cut into roughly equal pieces).

We would start by deciding which indicators we want to use to train on. If you don't have advanced knowledge on stock indicators (which I was able to acquire by studying each indicator and creating the functions myself, rather than just using a pre-made package), you should just use the provided indicator list that I've chosen for best results. However, the README goes in detail on how you can create your own indicator list to use. The example we'll go over here has 474 indicators consisting of indicators from Sirius-XM stock (which we're predicting for), and our indicators from our 6 helper stocks. We take the data from our indicator dataframes and start by training on the oldest data which is the first of 15 pieces and use the neural network returned to further refine the network using the next pieces, processing one dataframe at a time, updating the neural network as we go.

We initially set up our neural net to have 474 inputs (1 for each indicator), that has 9 activation neurons in the hidden layer, and what results in 1 output value in the output layer. The output layer is set up to be a predicted return value for that stock for a certain time step.

Each layer has additional delays to them meaning that the input layer is combined with previous input values that are then all fed to our hidden layer. The output to this hidden layer is then fed back to be used with the combined inputs and this is all combined to get the next output layer and this cycle continues for each delay point we give. Finally, the output value that first results from combined hidden layer output and is fed back and combined with the hidden layer output to get the next output value and this also happens for each delay value we give. For example, we use delay inputs [0,1,2] for the input layer, with 0 being just the input, followed by the t-1 input, and t-2 input. The hidden layer has delays of [1,2] so there's a recurrent connection for t-1 hidden output, and t-2 hidden output. Finally, the output layer has delays [1,2] where we add the recurrent connections for output t-1, and output t-2.


Output using both hidden layer outputs and previous outputs
Output from a previous timestep used in combination with hidden layer output to predict next output
output, y(t)
output, y(t-1)
Hidden layer with x number of activation neurons
hidden, z(t)
Previous hidden layer outputs used in combo with inputs for current hidden output
input, x(t)
hidden, z(t-1)
Input layer w/ x number of inputs and previous hidden layer outputs to get current hidden outputs
input, x(t-1)
Previous input used together with current input and previous hidden output to get current hidden output

(above shows an example of our network design)

After setting this network up, we finally were able to feed our indicator dataframe consisting of our chosen indicators. The output was trained for our example using the return value for the next time step which meant what the return value would be a minute from present. The inputs/outputs were then converted to numpy arrays for computational efficiency purposes rather than using pandas dataframes.

Once our network has been trained, a function is available to test how our predictions compare to the actual results. At this point we can graph our results compared to actual output to look at the difference between the two. Then we can examine the differences to figure out whether our decisions were correct.

Refinement:

The refinement process started with the initial network design which used just 13 input values using each indicator's recommended period length for each indicator. The number of hidden neurons at that point was 4, and there were no delays. This was initially set up knowing full well that without the use of delays in our network, we would only be using data from that time step which was not practical in predicting stock prices since stock prices depended heavily on what the stock price was at previously. This was set up like this though to test what little knowledge would mean for prediction. The result from this was expected, the results were wild, and roughly the same as if you predicted randomly what would happen.

The first refinement made was adding delays into our network. At this point we were able to base our predictions on not just current data but previous data. What was found was that the more delays added the very marginally our results bettered. However, it also added significantly more computation cost for each delay added. Eventually, I stuck with the delays that are set up now, and started increasing the number of inputs at this point.

I found as the more inputs I added the marginally better the results got, starting with using 2 periods per indicators, continuing until we were using what we have now which is 21 periods per indicator for the prediction stock. After I found this number of inputs to be sufficient, I started adding hidden layers, ending up with 9. After I added more than 9 I ran into the issue with memory where it would eat up more than my 12GB of RAM.

After all this, although marginally better than the initial setup, it still was lacking. At this point, I started using helper stocks to increase our knowledge of the overall stock market to see if that information could help. So I then took my best 21 period lengths per indicator for my prediction stock, and then added 3 periods per indicator for each of my helper stocks to see if that could help my predictions. At this point, I was at where I am now.

IV. Results

Model Evaluation and Validation:

It is at this point, that I came to the model that I have now that was briefly discussed above. It was a neural network made up of 474 inputs in the input layer, 9 activation neurons in the hidden layer, and 1 output neuron in the output layer. We'd be processing between 23,000 and 26,000 records at a time which resulted in roughly 3 months' worth of data. The delays in network that allowed for recurrent connections was [0,1,2] for the input layer, [1,2] for the hidden layer, and [1,2] for the output layer. I found these to be satisfactory considering our system limits, like 12GB of RAM, and a single dual-core processor. Any more inputs or delays or hidden activation neurons or records, usually resulted in an overflow of memory which forced it to write to disk, significantly killing my computation time.

Having these max limits, I found the best results using this combination of delays, # of records processed at a time, # of activation neurons, and # of inputs. This provided me with then the challenge of finding the optimal input parameters that would result in the most accurate predictions.

I found that using our predicting stock alone couldn't get us all the information, and it had to do with outside data affecting the results. For example, if information came out like the UK leaving the EU, that's information that our stock alone couldn't predict for, which then brought me to using outside info that could help in the prediction process. After adding outside sources like how the S&P500 and the Nasdaq indexes were doing, and how Gold and Oil were doing, these provided outside information that could affect our stock price without being visible using only info from our stock.

The final model was trained on data from around the beginning of 2013 to mid 2016, and our test was from July 12th,2016 to September 15th, 2016. Using our trained model, we made our return predictions at each interval. If it was expected to be positive, we bought, if negative, we shorted, and if 0, we stayed. We also did a random guess at each interval to test how it compared to random. The results were that during this time period, we were profitable $24,982.61 while the random solution was profitable by $286.22. We made a correct decision 72.8% of the time while the random solution was correct 50.1% of the time.

However, one thing to note was that smaller name stocks like Sirius were far better for predictions than big name like Apple. My theory on this is that a company like Sirius has less wildcards associated with it. For example, with Apple, due to its extremely big name, there were things that came with that that made predictions even tougher. For example, think about how Apple would have iPhone leaks before the new release. This is something that could cause wild fluctuations in the stock which would be absolutely impossible to predict for. While a model that retrieves outside information in real-time and doesn't have the minute delays could somewhat easily adjust for this, our model was not set up for this and although I couldn't prove that to be the difference, that was my thoughts.

Justification:

After all of our testing and retesting, on millions of different data samples, our end results were both extremely exciting and disappointing. Although our results were great for smaller name stocks, they were disappointing for more major stocks in general. The reason however for our failure is not so much in our process but in the data itself. We have a LOT of data that is available to process but can't due to system requirements that would over exceed our memory limits and therefore start writing to disk.

I found as soon as we couldn't fit everything into memory, you could consider that setup toast. As soon as disk writing and reading were introduced, not only was it very easy to completely freeze your system by using 100% memory and 100% disk at the same time, but it added hours and hours of processing time even if it didn't (might be different if you use an SSD rather than a HDD, but I don't). For our purposes this wasn't at all practical so strict limits were enforced and as far as our network went, I found the best results (although not perfect) were with the current setup.

The reason I believe the results are significant enough to be considered finished, is for two reasons. The first is that for the strict limits set up, I found the parameters chosen to be optimal. And as much as I would like to be able to test on more powerful systems, I simply don't have the finances to do so. However, considering the difficulty and the mountains of data that are required for accurate stock prediction, I believe this setup was a very good attempt and was well worth the effort. The second reason is because this process could give somebody with a more powerful system the ability to utilize more information at once. This was set up to be as scalable as possible and I believe I have achieved this and have done a good job in documenting how you could change input parameters and such within the code. For these reasons, I can justify this solution to be worthy of being considered satisfactory.

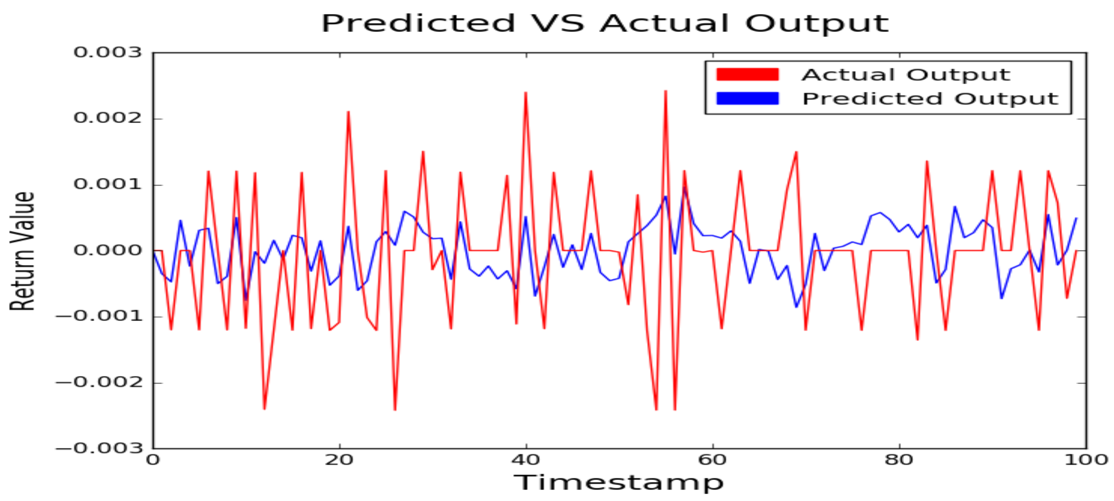
V. Conclusion

Free-Form Visualization:

Provided below is the last 10 trades of our test run using our trained network on the SiriusXM stock. The data below shows in the first column trade number meaning the number of trades since the start, then the Close column which is the previous close value we use to multiply the return value against. Then the actual return column which is what the actual output was at that interval. Then we have $Change, which is the return value multiplied by the previous closing price. Then we have the Guess column which is what our guess for that interval was, then we have $profit which is how much we were up or down, then we have the Random column which is just a random guess at each interval. Finally the random$profit column which is how much the random guesses made you.

```
|Trade#|Close|ActualReturn|$Change|Guess| $Profit|Random|Random$Profit|
18089  4.13004           0       0  short  24952.6  short  286.222
18090  4.13004  0.00120919   4.994    buy  24957.6  short  281.228
18091  4.13503           0       0    buy  24957.6  short  281.228
18092  4.13503  0.00241546   9.988    buy  24967.6    buy  291.216
18093  4.14502 -0.00240964  -9.988  short  24977.6    buy  281.228
18094  4.13503  0.00120773   4.994    buy  24982.6    buy  286.222
18095  4.14002           0       0    buy  24982.6    buy  286.222
18096  4.14002           0       0    buy  24982.6  short  286.222
18097  4.14002           0       0    buy  24982.6  short  286.222
```

To provide a more visual idea of our results, below is an example of the results of 100 stock minutes of prediction for our Sirius stock after training our algorithm. As noted in the visual, we show both prediction and actual return values. The return values represent increase or decreases in stock price for each minute interval. The most important feature is not so much about if we guessed the exact return value but if we guessed above or below 0 correctly.



End-to-End Project Information:

From start to finish, this project entailed months of creating and editing and resulted in thousands of lines of code. This project was set up so that a user with no stock experience whatsoever could use the package as long as they were capable enough to install the package correctly that are given in the README. As this README states, you'll first start with the ProgramInfoAndCreation file where you're given instructions on how to create the base high/low/close/typical dataframe dictionary. I've provided the data up until the middle of September so you don't have to download/adjust/resample all that data and adjust for dividends/stock-splits yourself. I only provide about 18 stocks because this is only meant to provide you with an example as you can choose as many stocks as you like from the data source yourself if you want more. In this file, I also give you a function to get info including price, avg daily volume, market cap, etc... This can help in deciding which of these companies you'd like to predict on first.

Then the helper stocks are printed in full so you know what they are and can look them up if necessary. The parameter choices are printed out for you for each indicator if you were curious or wanted to change what indicators you were going to use and wanted to know what each period represented. Then a column-name dictionary was created with names for each column in a specified order which was used internally. Also the news dictionary with url's was created for whenever I add an NLP solution to it. I also had several optional functions to create your own indicator list, grab the previous 10 days' worth of intraday data to compare to our data, as well as give you the dates that we are missing data from.

The next file we use is the DividendAndSplitUpdate file, which as the name suggests, updates the high/low/close/typical value dataframes for any dividends and/or stock splits that have occurred since the data was uploaded. This updates your dictionary to be present and accurate to present, and allows you a way to easily update your dictionary both during creation, and if you need to update one or two in the future.

The next file for use is the indicator creation file. This file allows you to create the indicators for each company in your highlowclose dictionary. In the file, it gives you the option to create your indicator dataframes using an intermediary step placing the individual indicators into their own files in a directory with their stock symbol as the name, so that you wouldn't blow up your memory trying to create the dataframes but another option is given to avoid this in place of not having the intermediary storage space used.

The next file is the RealTimeCalculator file which allows for real-time indicator calculations using real-time stock data retrieved from a Google server. The program was set up so if you didn't have time before 9:30AM Eastern to set the program, you could call it the night before and it would automatically start at the correct time and end at the correct time by testing what time is was every 30 seconds and if it wasn't after 9:29, it would sleep for 30 seconds. At the end, it will update your historical indicators and highlowclose dictionary.

The file RealTimeNeuralNetwork, is where we train the network using our historical indicators. This process can take a long time since we have so much available data to process which is part of the reason for having the indicator dataframes cut into 15 pieces allowing you to process parts at a time and continue the next parts when you get the chance.

The file CalculateAndAddMissingData, is also provided so that if for any reason after being current with indicators, you miss a day or more of using the RealTimeCalculator file to update the indicators, there's an easy function to call that will automatically figure out what data you're missing and adjust to that point using our data source and several indicator/adjustment functions that make it just a click and enter process.

Reflection:

Having given a brief overview of each part of the project, I'll go into what I found to be the most interesting parts of the program. One of the most interesting is the ease of use of this program relative to the degree of market knowledge needed for productive stock trading. It was setup with the idea that somebody who simply wanted to use it without knowing anything about stocks and little about python to use. I believe I did a good job with this and even made it so if somebody did have an advanced knowledge with stock trading, they could use the more advanced functions to adjust indicator parameters. This is something I was very happy with since it provided by beginner and advanced user to benefit from it.

Another interesting part of this program is the scalability ability that I've allowed. Since we get 416 indicators per company, and only use a fraction of these indicators during training, it gives somebody with the system ability to process more data if they wanted to. With the ability to process more data, it gives the user the ability to achieve better results if they can. This program was made specifically with the idea that it should be easy to add/subtract parameters and process in batches so your training didn't have to all be done at once.

Another interesting part I wanted to point out was that every indicator function was created myself. There are packages out there you can use to get similar functions but due to the fact that I wanted to learn more about what these functions were doing and make them in a memory and computationally efficient manner I chose to create them myself. In the process I learned how to create these indicators as well as how to write the code in an efficient manner as well as allowing me to make additions not possible on other platforms like creating many period-length indicators at once.

The last interesting part of the program was my modifications to the neural network that I used from a provider cited in my README and below. Although the modifications were relatively small and simple in manner, the process of finding them could not have been achieved by myself before this project began. For this to be even possible I first learned everything I could about the algorithm, initially rewriting all the function/function-variable names to be more descriptive for what they were doing. This allowed me to gain a deep understanding of the algorithm I couldn't have easily learned elsewise. Along the way creating this project, I learned a lot about all kinds of techniques for memory and computationally efficient programming because of the intense need when processing mountains of data. All of this knowledge allowed me to recognize two major memory leaks that were occurring in one of the functions in the package that would never have been recognizable by me before I started this project.

Unlike a lot of noticeable memory leaks that as the program continues, it would either crash the program or the whole system because it would continue blowing through usable memory which would be relatively easy to notice, these weren't as easy to find. The memory leak only occurred in a single function where, if it got through the function, the memory would essentially reset as it returned its variables to the calling function. At first, I believed it to be a result of processing so much data at once. However, the more I debugged looking for ways to improve performance of the whole program, the more I noticed a rather severe increase in memory use in a single function that would continue rising throughout the program operating.

The next thing I tried doing was testing variable sizes at certain points in the function and seeing which variables were rising in size. Then I had to diagnose whether it was normal for this to be increasing or not. Although the code was correct, it turned out that two of the variables within were using dictionaries and were adding key/value pairs and these dictionaries got very large as they kept being added to. Then after looking at whether we needed these dictionaries to contain all these pairs, I found that we could keep only as many as required to retrieve x-delay minutes into the past entries to save memory. Then I searched for the best way to update these dictionaries, first using a method that used the del dict[‘key’] method after each cycle, but found a much more efficient method was to actually create a new dictionary with our previous 5 keys disregarding the other keys after each loop.

The other change made was using a scipy alternative to the numpy dot method when we were doing matrix multiplication on our Jacobian matrixes. I did this because the scipy alternative didn’t copy the lists before the multiplication. The reason numpy does this is because numpy requires the lists to be in C-contiguous order. Scipy on the other hand allowed us to not have to copy the lists by passing them in as C-contiguous order without the copying which resulted in less memory consumption and quicker processing. The small change allowed that method to run in roughly 1/6 the time. This however wasn’t utilized elsewise in other dot multiplications because it only made a difference on LARGE matrix multiplications.

The other change that was small but had fairly big impact was converting to the scipy alternative of the numpy matrix inversion method. This change was due to the fact that scipy always uses BLAS/LAPACK compilation support while numpy only optionally uses this. This provided a much faster alternative that was easy to implement.

The final change in the algorithm I made was adding time parameters so that if a cycle was taking more than 2 hours, it would return. This was because sometimes, it would get stuck trying to find a better cycle than the previous run and it would keep trying and started overfitting.

These changes allowed for a significant increase in processing. When I say significant, I mean it. Prior to this alteration, I could process about 10,000 records with 100 different inputs for each record at a time due to my memory blowing up with more. However once the change happened, I was able to process 26,000 records and 474 different inputs for each record, and do so in about the same, sometimes even less time than previously. The change for me was huge allowing for much more data to be processed at a time and this would never have been possible had I not learned so much along the way during this project.

The most difficult aspect of this project was not having it work that well for big name companies. This was aggravating to me because of the amount of work I put in to get to this point. No matter how much I tested with different parameters and choices, I couldn’t create a solution that beat my benchmark on these companies. In reality, I realize how difficult my benchmark was considering if I did, I might not have to work anymore lol. But still it was annoying since how hard I worked to try to get a working solution.

That being said, I was still happy overall with my project because of the level of difficulty this project took and the increased programming abilities I achieved throughout this project. This might not have been a profitable solution for every company, but it helped my programming skills tremendously in the process.

For this reason, I would say that this solution fits my expectations for the project. However due to the unprofitability for some, I would not suggest use in a general setting for every company unless you were able to test on a system or set of systems allowing you to process more data at once and that achieved profitability for you.

However, one last note is that data is very valuable, and the fact that you can get real-time stock data with my solution means that you could technically sell intraday data which has a surprisingly big market if you had enough of it and it was shown to be very accurate. I don't endorse this only because I'm not positive you'd be allowed to with Google's terms of service. You can use their server for personal use for sure, but commercial use I'm not sure about.

Improvement:

One improvement to this project that could be made that would be most likely to be beneficial in my opinion, would be adding an NLP (natural language processing) algorithm to my solution. If you've seen the code already, you know I already have a function that retrieves url's to stories relating to each stock dating from present to years in the past. With this information that I was able to attain using Google's servers, a NLP algorithm that could work in conjunction with my RTRL algorithm would greatly enhance the prediction process I believe.

The reason for this is that at current, my algorithm uses lots of stock indicators/previous data to predict what's about to happen. This however cannot account for non-data issues like Apple releasing a report saying that its iPhone sales are down, which would undoubtedly disrupt its stock price. This type of information could not be gleaned from our current setup but using our news data, it could. However, we would need to feed it to an NLP which make sense of what an article about this sales drop would mean.

At current, I don't even know where to start when it comes to utilizing an NLP algorithm but the topic is very much interesting enough for me to want to learn about. I don't add it to the project at this point because I won't have time before it is due and feel I've done enough to show my proficiency in a very important area of deep learning and also my knowledge of setting up an efficient solution.

If I used my solution at present as a new benchmark, I believe I may be able to exceed it if I were to add the NLP addition because it's information that I currently cannot glean from the indicators that I have that would only positively benefit the results. The only issue would be having it be efficient enough to be both memory and computationally lean. But if this is done correctly, I believe it could provide a better solution than I have now.

# Credits

1. Following functions thanks to https://gist.github.com/jckantor/d100a028027c5a6b8340:
    - NYSE_tradingdays()
    - NYSE_holidays()
    - NYSE_tradingdays2()
    - NYSE_holidays2()

   This set of functions is used to calculate past and future trading days, making sure
   to adjust for trading holidays and such. The first set of functions, NYSE_tradingdays()
   and NYSE_holidays() is the original functions as was created entirely by jckantor,
   while the second set was altered by me to adjust for a specific requirement I needed,

but was based entirely off the original so thanks to jckantor for providing this set of extremley helpful trading functions!!

2. Following package thanks to https://github.com/hongtaocai/googlefinance:
    - googlefinance

   This package allowed for retrieving real-time stock data from Google's finance servers that allowed this project to calculate real-time indicators. It also allowed for getting year's worth of news data for those companies up to present. Thanks again, this project wouldn't exist if it weren't for them!

3. Following package thanks to https://github.com/lukaszbanasiak/yahoo-finance:
    - yahoofinance

   This package allowed for providing information for each company when deciding which companies to track. Thanks to lukaszbanasiak for making this package public!

4. Following package thanks to https://github.com/quandl/quandl-python:
    - quandl

   Quandl provides all kinds of data that is both free and pay. They are one of the leading trading data providers out there, so check them out if interested. For my portion of the project, I utilize there adjusted closing prices for use if a company has a dividend or stock split you need to adjust for. Thanks to them for providing that useful free data!

5. Following functions thanks to Dennis Atabay at https://pyrenn.readthedocs.io/en/latest/:
    - create_neural_network()
    - create_weight_vector()
    - convert_matrices_to_vector()
    - convert_vector_to_matrices()
    - get_network_output()
    - RTRL()
    - prepare_data()
    - train_LM()
    - calc_error()
    - NNOut()

   These functions allowed for utilizing real-time recurrent learning of neural nets. These were out of a package called Pyrenn. There is a lot more functionality from that package so check it out. These functions have been altered to use more descriptive naming of variables and functions, for both learning purposes and for anyone checking out this package who wanted a little easier to follow group of functions.
   One major change was in RTRL(), the dictionaries deriv_layer_outputs_respect_weight_vect, and sensitivity_matrix. These dictionaries started to cause issues when training on more than 10,000 data records at a time. Due to the way they were set up, these functions would continuously add key/values to there dictionaries after calculating those particular values. The issue arose because the function only needed to keep enough to calculate up to the largest delay. What this meant was that rather than keeping tens of thousands of data keys/values in a dictionary these dictionaries, we could just keep, for example, the previous 5 key/value pairs that were needed for calculating with delays but no more. What resulted was massive performance rewards in terms of memory consumption, and as a result of significantly less memory consumption, computing time as well. That being said, this project wouldn't have been possible if not for them, so thanks again!!

6. Finally, thanks to everyone at http://thebonnotgang.com/tbg/ for providing FREE INTRADAY DATA!!!! This was probably the most amazing find of this project. Prior to this, I was working on daily stock values, rather than intraday, so when I found these guys, I couldn't have been happier. Please Please Please check out there site, they provide apps, and all sorts of help. These guys were a lifesaver so show them some love! They are the only guys I was able to find on here who provided free intraday data, so if you're poor like me and ever need intraday data, check them out!