Modified Q-Learning Algorithm for Stock Trading

By Lazar Andjelic

I. Introduction

With the advent of algorithms in trading, more and more people now are exploring the relationships between these algorithms and the results they may produce. This project I've been working on explores the effectiveness of two different types of machine learning algorithms, reinforcement learning and supervised learning.

Reinforcement learning[1] involves a system of punishment and rewards for performing a certain action. By rewarding good behavior and punishing bad behavior, it seeks to optimize the best possible actions to perform in an algorithm's current state. Deepmind and AlphaGo made headlines recently after their algorithm successfully beat the world's best Go player, a game that is nearly impossible to master. The reinforcement algorithm I'm exploring with this project is called Q-Learning[2]. Q-Learning is a model-free form of reinforcement learning that involves an "agent" that trains itself on an unknown environment. This agent navigates through every possible action during a given state through what is known as state-action, or Q-matrix (hence the name). Stock trading is all about performing an action based on the current state of the trader, and the stock market is always an unknown environment; this algorithm seems like a perfect fit. The Q-matrix updates itself by referencing to information found in the environment's reward matrix, which usually has predetermined values for performing a certain state-action. Usually the environment is framed with a particular goal state in mind, such as victory in a game or reaching the end of a maze. This unfortunately, is where issues start with applying this sort of algorithm to stock trading. There usually is no "goal" in trading; you could set a particular goal such as producing a certain amount of profit, trading a security when it reaches some sort of value, etc.

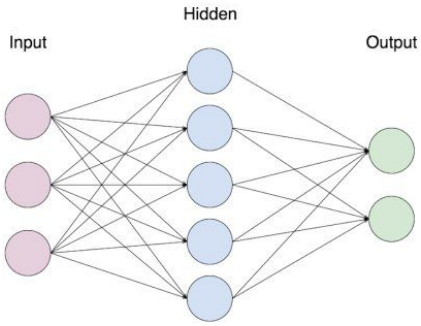
---

[1]Knight, Will. "Deep Learning Boosted AI. Now the next Big Thing in Machine Intelligence Is Coming." *MIT Technology Review*, MIT Technology Review, 6 Apr. 2017,

[2] "Q-Learning .*;" *A Painless Q-Learning Tutorial*, mnemstudio.org/path-finding-q-learning-tutorial.htm.

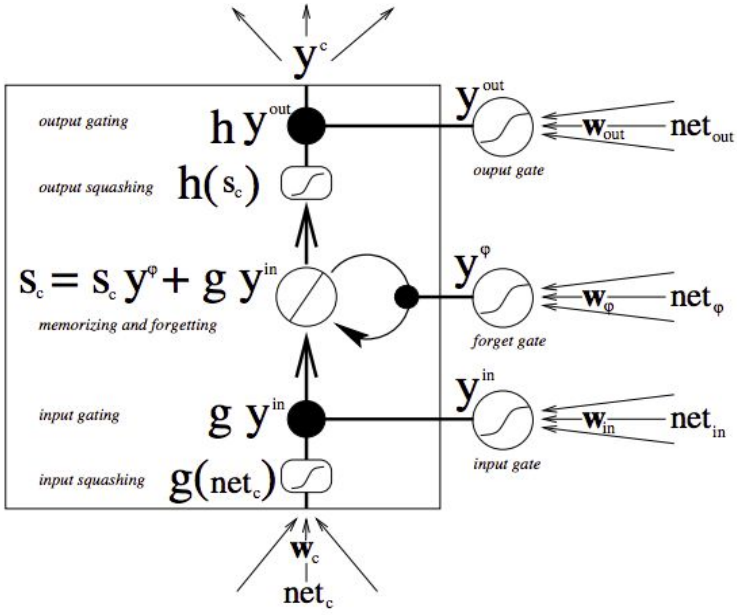
But even if these set goals were added, the other issue comes with the fact that the environments these algorithms navigate are either set or any variations in the environment are predictable. This is not the case with the stock market. Even as our algorithms keep getting better and better at prediction, there is always a level of uncertainty in trying to predict the market. This could make any sort of training on the environment futile, as the agent can't learn how to perform well in an environment if the way to perform well in that environment is fluctuating unpredictably.

This is where supervised learning comes in. Supervised learning[3], as the name suggests, involves taking a set of input data with a set of desired output data, and then trying to develop a model that will map a function from the input to the output. This is why the structure of a neural network is so popular with a supervised learning algorithm, as the sole purpose of the structure is transforming the value of input data to its desired value of output data. When it comes to stock trading, we could feed a neural network many different types of data, such as news regarding the security or sentiment analysis about the company of the security, but for simplicity I'm focusing mainly on the movement of the price of the security itself. We could just take a bunch of data of the past stock price data and feed it all into a feed-forward neural network, similar to the one pictured above, and get a model. However this model would not be as accurate and as powerful as we'd want it to, because the value of a stock price on a given day will have an effect on its value for the next several days. A feed forward neural network would only look at each individual value and not form a relationship to those price movements. The time when the price moves may affect future values, as more recent price movements will have a greater effect on the current price



[3] "Supervised and Unsupervised Machine Learning Algorithms." *Machine Learning Mastery*, 21 Sept. 2016

compared to movements much earlier. This is where an LSTM[4], or long short-term memory, neural network comes into play. It's a type of neural network where the hidden layer is replaced by memory cells that control the flow of information and error through them by regulating input, output, and forget gates, as well as an internal cell state, to determine what parts of past input to remember and what parts to forget. This makes it especially effective remembering sequences of data, which makes it popular with language translation, speech recognition, and forecasting prediction. If we were to have an LSTM model of the price movement of our stock security, it could give more certainty to knowledge of the stock market environment. We could use this knowledge to perhaps take the non-viable Q-learning algorithm and make it more than useful for maximizing profit.



II. Process

My data collection involved downloading 10 years worth of daily security data from the Apple (AAPL) security. I didn't rely on Quantopian or other open source hedge funds for data nor simulation, and thus I had to hard code a rough sort of "simulation", which meant that I limited the portfolio of a trader to only being able to trade and hold onto one share of AAPL at a

[4] Chris V. Nicholson, Adam Gibson, Skymind team. "A Beginner's Guide to Recurrent Networks and LSTMs." *A Beginner's Guide to Recurrent Networks and LSTMs*

time. Since the data was of daily stock prices, the open daily price was mainly used, while volume, the close price, the high, and low were later used as features for the LSTM.

I split my findings into two separate experiments, one which involved only the Q-Learning algorithm, and the other that supplemented it with an LSTM neural network.

Building the framework for a Q-Learning algorithm was definitely the most time consuming part of this project. The design of it went through many different stages as certain things worked better than others. The main challenge was designing the Q and Reward matrices.

I started with the ideas of the states being “losing” and “gaining” money, or with the price “increasing”, “decreasing”, or “staying the same”. However, the issue with these approaches is that Q-Learning algorithms work best when a particular action can predictably lead to a particular next state. With the nature of arranging states in these unpredictable ways, it becomes non-viable. So my solution came with setting up the states as “Long”, “Short”, and “Neutral”, and setting up the actions as “Stay” and “Leave” for the first two states and “Long”/”Short” for the neutral state. To “long” a stock means to buy it when the price is lower in order to later sell it at a higher price. To “short” a stock means to sell it when the price is high in order to later buy it at a lower price. Thus, the state of long would be if a stock has been bought, and now the trader waits to be able to sell it. “Stay” would keep the trader in this trading cycle, while “Leave” is when they can either buy or sell the stock to complete the long/short loop.

Figuring out the reward matrix was also a challenge, since there are several ways you could go about rewarding the algorithm for a trade. At first I wanted the reward function to just be the change in price, but to add a level of long-term dependency, as again, previous

movements in price can affect the future current price to varying degrees, I instead made the reward function an average value of the price in a given time window:

$$R(s,a) = \frac{1}{\theta} \int_{t-\theta}^{t} \frac{\partial P}{\partial t}$$

where R is the reward function, s and a are the given state and action that the reward matrix is updating, P is the current price of the security, t is the current time step in the process, and θ is the window of time with which the price movement will be taken into account in the reward for a state action. To be able to keep track of this mean and forget movements after the time window, in addition to the structure of the Q-matrix, the reward matrix has an extra dimension for each state-action that is built like a queue, since a queue is the best possible data structure to keep track of adding new values while getting rid of values that came first in the time window. This adds on to the usual formula for updating the Q-matrix, which is:[5]

$$Q(s,a) = R(s,a) + \gamma[Q(s', max(a))]$$

where Q is the Q matrix, γ is the discount factor (the amount that the algorithm takes into account future actions), s' is the next state, and max(a) is the maximum possible action given s'. I also added a "my returns" part to the algorithm, as while gaining/losing money doesn't directly affect the q and reward matrices, it does help in assessing how well the algorithm is performing.

The algorithm starts with splitting the data for training and testing. Then the algorithm runs through the training data a certain number of epochs, recording the returns and q values from each epoch. The Q matrix is updated after every action, as this is how it's "learning" the

[5] "Q-Learning .*;" *A Painless Q-Learning Tutorial*, mnemstudio.org/path-finding-q-learning-tutorial.htm.

environment. It starts at the neutral state, and then chooses actions based on which one has a higher q value, or picking randomly if they're the same. Once it runs through a long/short loop, it will return back to the neutral state and repeat while the algorithm keeps going. Once it runs through all the epochs, the algorithm will pick the final q values in the epoch that produced the greatest returns and set those as the q values of the "model". The algorithm then runs once on the testing data, simulating as if it were trading in real time given the q values, though these q values don't update anymore. The net returns produced after running through the testing data is theoretically how much money we would've made if we were to apply this algorithm to the real market in the time period of the testing data.

Experiment 2 is similar to that of experiment 1, with the only difference being the inclusion of the LSTM. The LSTM trains itself on the same data as Q training data, creating a model that will predict tomorrow's open price.

The model was trained by first normalizing all the data, both testing and training, between 0 and 1, as LSTMs are sensitive to minute differences in value required in the data we're analyzing, so normalizing it made training faster and significantly reduced error. However, this means that for now we can only backtest this algorithm, as we can't accurately normalize live data as it's coming in this same way. It's possible to train an LSTM without normalization, but it's beyond the current computing power and technology at my disposal.

The neural network is structured first with an LSTM hidden layer of 800 units, which is about half the size of the training data. Then, the dropout rate was set to 25%. Dropout will randomly set values of the input data to 0 during each epoch to prevent overfitting[6], where

[6] "Overfitting in Machine Learning: What It Is and How to Prevent It." *EliteDataScience*, 8 Feb. 2018

training an algorithm too much can actually make it perform worse, as it becomes more sensitive to the minute details of the training data instead of following the general trend. Then, we feed this layer into another hidden LSTM layer, this time with 200 units. Then a Dense layer with only one unit was added. A Keras "Dense" layer is just a typical neural network hidden layer that computes an output based on, [7]

$$\phi\left(\sum_{i=1}^{n}\left[x_i \bullet w_i + b_i\right]\right)$$

where ϕ is the activation function of the layer, x is a matrix of the inputs, w is the weight matrix, whose dot product is then added to b, the bias matrix. The weights and bias matrices are updated over time during backpropagation of the network after an output is generated and error is calculated during every epoch. The activation function for this network is "linear", which just keeps the value the way it is. The loss function is mean squared error, or [8]

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2$$

where Y_i is the output of each input value and $\widehat{Y}_i$ is the output that the neural network produced of each output. The optimizer is known as ADAM[9], which is a combination of two popular optimizer functions, known as Adagrad and rmsprop. The inventors of the algorithm propose below:

[7] *Neural Network Basics*, www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html.
[8] "Mean Squared Error." *Wikipedia*, Wikimedia Foundation, 18 Feb. 2018
[9] Kingma, Diederik, and Jimmy Ba. "Adam: A Method for Stochastic Optimization." *[1412.6980v8] Adam: A Method for Stochastic Optimization*, 23 July 2015

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

During testing, the algorithm is modified by adding the difference between tomorrow's predicted price with today's predicted price. The reason why predicted and actual prices weren't compared was to avoid the possibility of the LSTM model being off by scaling, as even if the price movements follow the same pattern, the actual values between the predicted and testing prices could be different.

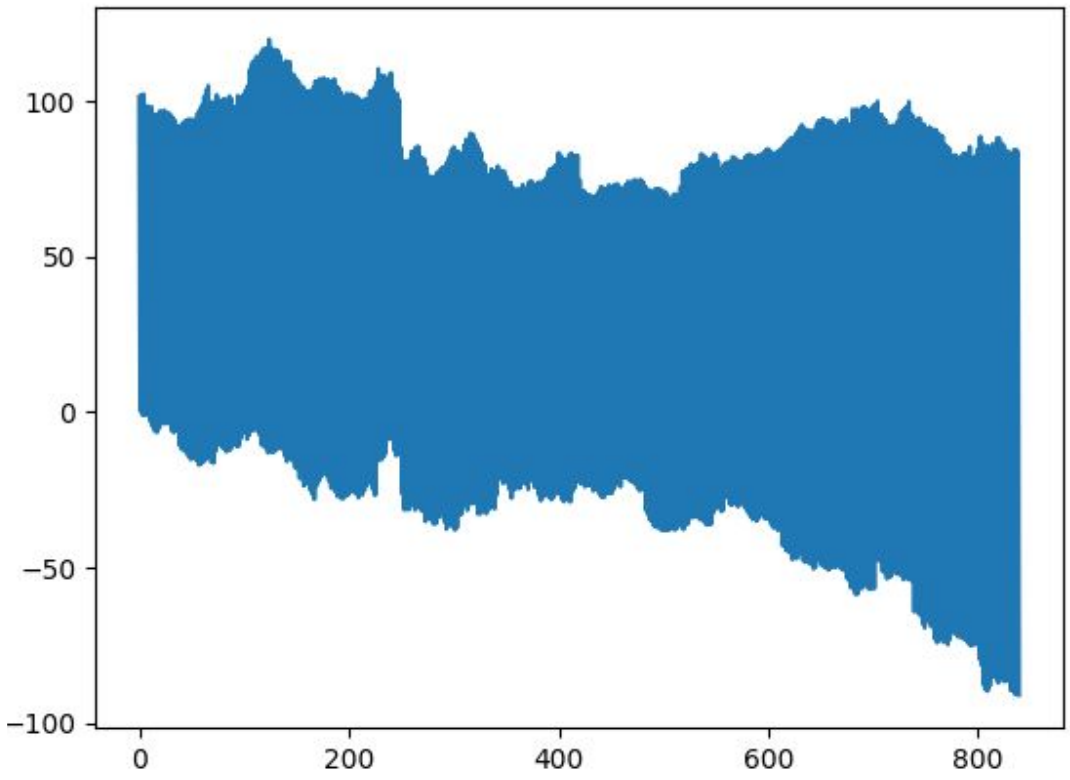
In trying to make this second experiment better, I wanted to see if there was a possibility of reducing uncertainty with the LSTM model to ensure that skewed predictions aren't making the algorithm worse. So I created an alpha value, and divided the added difference by 1 + α:

$$\alpha(t) = \frac{1}{\delta} \int_{t-\delta}^{t} \frac{\partial P^*_{t-1}}{\partial t}$$

where P* is the predicted price, and δ is the time window that "confidence" in the neural network will be taken into account, as this is a crude estimate of how the algorithm is performing, decreasing its effect on the choice of action if the difference in prediction is too high.

III. Results

Experiment 1, predictably, didn't produce reliable results every time, as the algorithm's performance depends on how it randomly responded to the unpredictable environment. The most common result of the algorithm was generally losing money:



Q values:

[-0.07784999999999975, 0.1892750000000003]

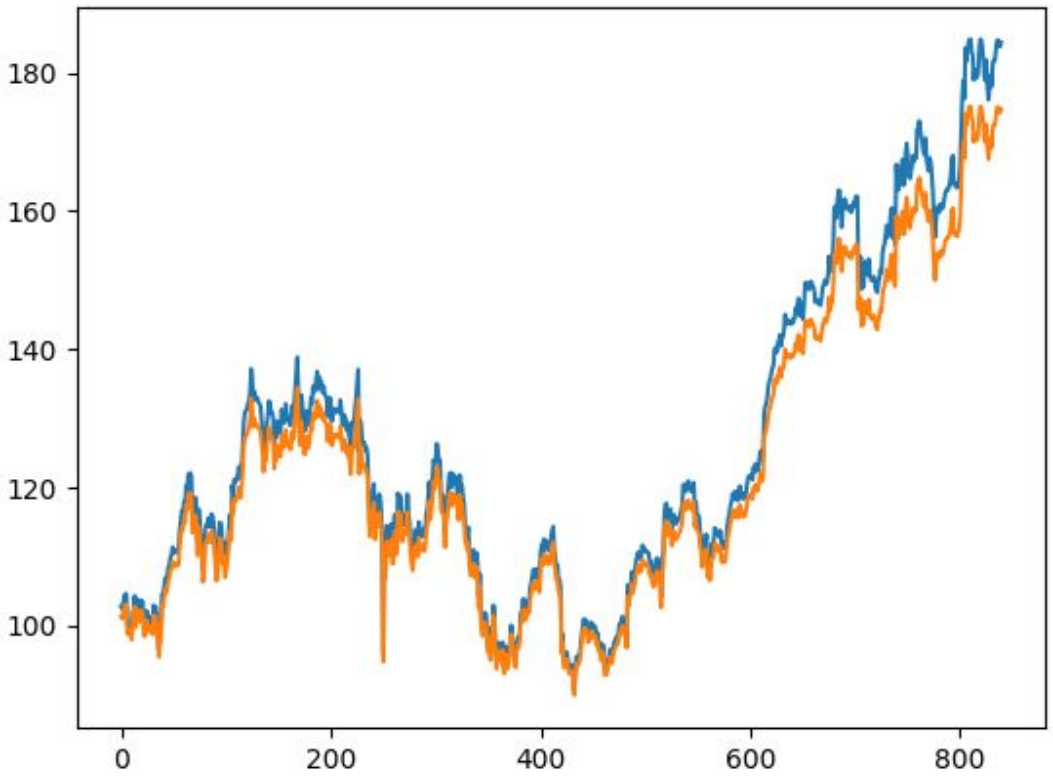
[0.2708454999999999, -0.4486000000000008]

[-0.3438083333333323, -0.18291325000000053]

Returns: $-90.933

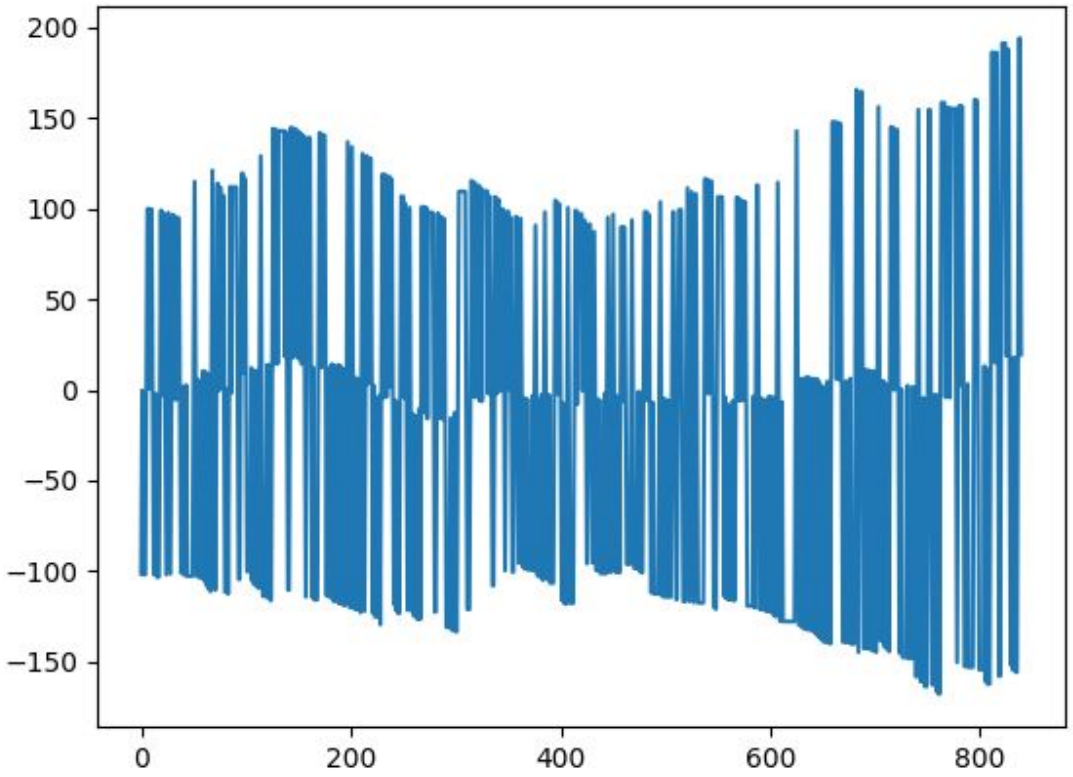
The graph was made with MatPlotLib, and the reason it appears so fluctuated is that each return, whether it was a buy or sell, was taken into account. Judging from the middle of the graph, the algorithm started doing pretty well, but around the time the general curve of the price started to fluctuate was when it started failing. Also, it's noticeable in the graph that the buy and sell loops happened one after another instantaneously. The algorithm had no way of knowing when to stay in a loop to increase profits, and while this could benefit it sometimes, the uncertainty makes it unpredictable.

Experiment 2 had some interesting results. Here is the LSTM modelling the data:



The orange line is the testing data while the blue is the model predictions. While it did overshoot a bit during big spikes in the data, overall it closely follows the data trend.

First I tested it without taking into account the alpha value, as I wanted to see how it would first do just relying on the LSTM model. It wasn't as groundbreaking as expected, but it fared significantly better than Experiment 1:



Q-Values:

[-0.7035499999999999, -0.3464776666666669]

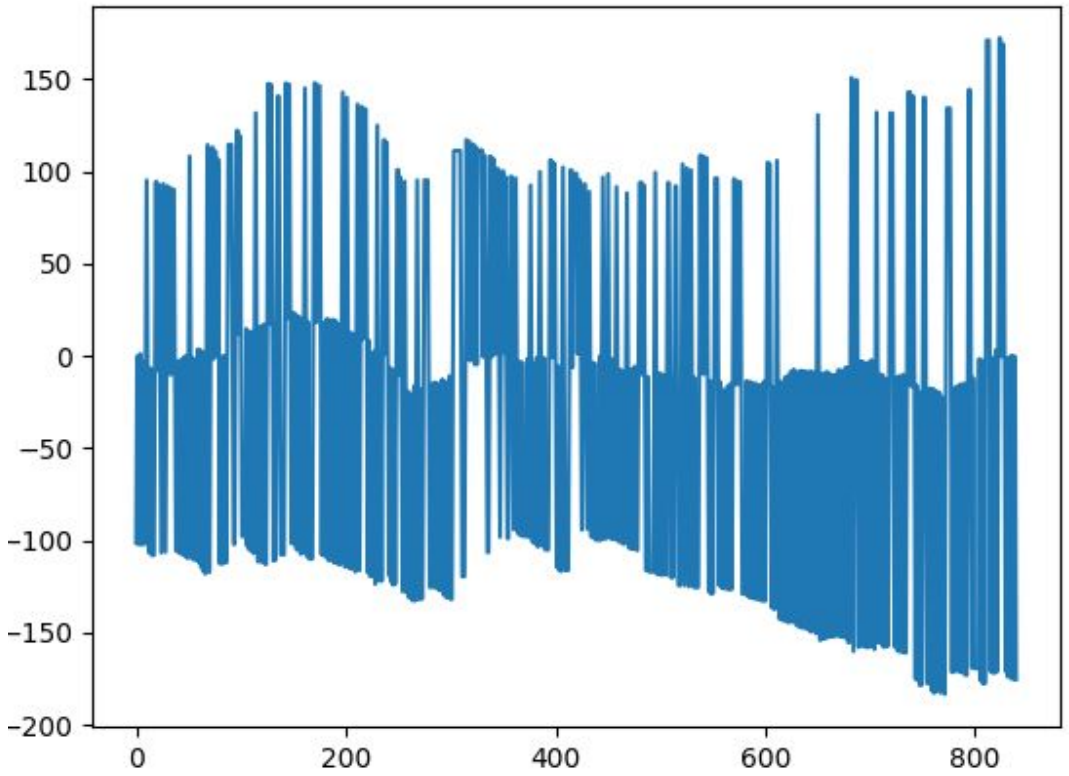
[-0.07784999999999975, -0.07308333333333294]

[0.002969166666666301, -0.2843]

Returns: $19.378

The much greater spacing from the lines indicate that the algorithm did hold onto buy/sell loops much more than Experiment 1 did. Granted it isn't perfect; the middle of the graph fluctuates just above the 0 line, but it's quite the improvement from the Q-Learning algorithm alone. It was able to hold onto profits as the price began to dip, and it made some huge buys and sells near the end, meaning that if extended to more data, it might be able to perform even better.

Then I tested Experiment 2 with the alpha values, and I adjusted the alpha window values to optimize returns:



Q Values:

[-0.7035499999999999, -0.3464776666666669]

[-0.07784999999999975, -0.07308333333333294]

[0.002969166666666301, -0.2843]

Returns: $-175.308

Even optimized at an alpha window of 2, the algorithm did significantly worse than even the first experiment. While the LSTM wouldn't have been overfitted in this case, the balance between the predicted price movements and the q values in choosing an action could've been thrown off by the scaling adjustment. There could be other ways to measure the LSTM model's confidence to improve overall performance, but not every measure of it can be beneficial.

IV. Conclusion

When it comes to stock trading and predicting the stock market, even when our models become more powerful and more accurate, that doesn't necessarily mean we'll make the most amount of money. This project was more of an examination of applying another type of machine learning algorithm could benefit an algorithm that performed on its own. There can be dozens of ways to expand on these ideas; right now, the only real signals the algorithm took into account were q learning and price movement, but there could be others we can add, such as sentiment analysis and when the highest volume of trades happen. While this project only focused on one stock, adding volumes of shares to be traded can add another layer of complexity to the structure of the algorithm as well as giving potential to making exponentially more amounts of money, as no serious hedge fund trades with just one share.

And we can apply these ideas to many other fields besides just the stock market. Cryptocurrency markets are immensely growing in popularity, so it would be interesting to follow them with some of these approaches. Outside of finance, assisting reinforcement-like

algorithms with supervised learning could greatly improve performance anywhere where we're applying AI to action, such as robotics, game theory, and medicine. In the future, I would like to continue exploring these ideas and applications, as this project was quite fascinating to me.