EECE 5644: Machine Learning and Pattern Recognition

Reinforcement Learning on Stock Data

Group Members (undergraduates): Lawrence Thul, Anthony Bisulco, Brendon Welsh, Jake Messner

Introduction:

After touching on the topic in class, our team has decided to take a deeper look into reinforcement learning. Reinforcement learning has been found to be able to perform amazing tasks, such as robotics manipulation and playing games better than their human competitors. Utilizing reinforcement learning, we built a trading agent applicable to many types of securities and investments. Using stock market data for our project provided us the opportunity to obtain significant amounts of high resolution data cheaply, however this brings along the challenge of analyzing noisy data. We will additionally apply our reinforcement learning algorithm to prices of cryptocurrencies and analyze the way in which these are related to traditional securities. We believe that, by grouping together bundles of securities in related fields, we will be able to build trading data sets that better represent the security we're interested in trading and therefore give us a better fix on price.

By beginning out research with easy-to-acquire data from Bitfinex, we can train a prediction algorithm using Bitcoin prices. This provided example values to work with, which were then applied to different securities to test accuracy. Moving forward, we dive into testing the validity of others' research by using a set of securities as training data and comparing this to the future future of the same index trained on. Continuing past the majority of research conducted by others, we will train our algorithm on one discrete security at a time and use this data to predict the future of the security. Finding mixed success, as discussed later, in this method, we continue on to develop sector index funds of twenty five securities per index for five of the prevalent sectors in security trading. We then use historic data from a single index fund to train our algorithm and then attempt to utilize this data from a sector to evaluate the performance of the algorithm.

This project will be implemented in several components. Our primary data class will fetch security data from Quandl, a publicly available data repository accessible via an API. The financial data class then error checks and cleans the data for training in our algorithm. Using a Q function and reinforcement learning algorithm from PyTorch, we experiment with our own reward function to better balance the output. We additionally perform extensive tuning of the hyperparameters of the algorithm to experiment with the effect of different adjustments.

Prior Work:

It was important for us to begin our research process with learning about other researchers in the field and how we can improve upon and modify these existing approaches to the same problem.

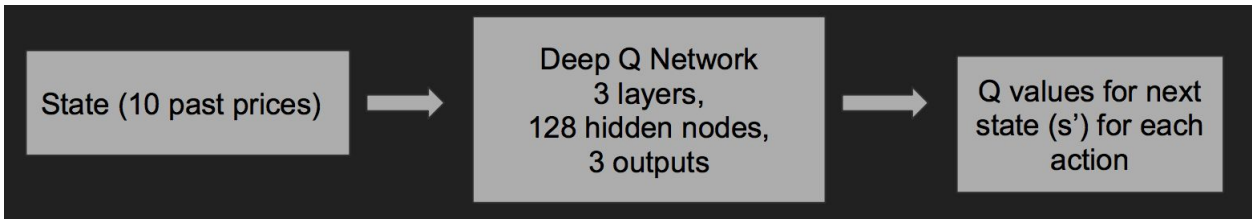
Since the discovery and popularization of reinforcement learning, several notable researchers have attempted stock prediction. Moody and Saffell at the Oregon

Graduate Institute developed a trading system comparing Q-Learning to Recurrent Reinforcement Learning [1], and found that their system could track the Standard & Poor's 500 index (S&P 500) with more predictability than researchers attempting to predict tracks of independent securities. They included additional features to tune their algorithm such as nonlinear interest adjustment over time. Similarly, others such as Lilian Weng [2] have used recurrent neural networks to predict stock prices, but they also use large industry-spanning sets of data. Curious as to why the vast majority of existing research utilized only large datasets, we set out to compare dataset testing with testing of individual securities. We hypothesized that utilizing a large index as a training set, as these other researchers had, would dramatically decrease the amount of noise in the data and thus allow for more accurate predictions.

Methods:

In order to implement the reinforcement learning framework, we had to make multiple definitions and policy decisions. The first definition that we implemented was the state set, which we decided to define using the 10 past closing prices of the raw data. For each state, we also defined an action set, which is an action to take in order to move into the next state. Since our data is a time series, the natural next state would be a shift in time data by one day. In order to determine how to move between states we also had to determine a method that would be able to predict the value an action would have between states, which is called the Q-function. In order to calculate the expected Q values we decided to use deep learning because we wanted to attempt to extract that information from the raw data signal. We leveraged the PyTorch deep learning toolbox and examples from the PyTorch website [3][4]. We also defined a policy, which is the method our algorithm uses in order to choose an action given a specific state. The policy that we decided to use is called epsilon-greedy. The deep Q learning neural network basically calculates the Q values for each action given the current state and the epsilon greedy policy will take the action associated with the maximum Q value. Although, taking the maximum Q value everytime will not allow the reinforcement learning model to explore different actions associated with each state, which is the other characteristic of our policy. We created a hyperparameter in our model, ε, which is a percentage of the time that our model will choose a random action in order to allow our algorithm to explore during training. As the training episode iterates through the data, the epsilon value will decay exponentially. At this point, we have defined the state and action set, the Q function estimation method, and the policy. In order to actually optimize our model, we also defined a reward function which tells the algorithm the actual value of the next state once it takes an action. All of these methods were determined in order to try and optimize a trading strategy using reinforcement learning.

Each of the implementation details for each part of this algorithm can be broken down further. In order to predict the expected value of the next state in the network, we created a deep neural network to learn the Q values. The neural network is a 3 layer network with 128 hidden nodes in each layer, and 3 outputs values. The input to the network is the state, which has 10 features representing each of the past 10 normalized closing prices[Fig. 1].



The action is then chosen using the epsilon greedy policy, which was the maximum Q value output from the network. By choosing an action, a reward is also output from the model. The reward can be positive or negative. The reward acts as the target value used to optimize our network with, which is incorporated into the expected value of the next state. The equation below is used in order to get the expected Q value of the next state:

$$Q(s',a) = Q(s,a) * \gamma + \text{Reward}$$

The output from the neural network, Q(s,a), is multiplied by γ, the discount factor, which is a value between 0 and 1. When the discount factor is closer to 0, the model will favor an immediate reward over a long term reward, while a value closer to 1 will favor long term rewards. Then the product of the neural network output and discount factor is added to the reward, which gives an expected reward, Q(s',a) for a state given the action. At this point, in order to train the network, we take the Huber loss between the expected reward and the output from the neural network and perform the backpropagation algorithm using that loss. Our model also uses the RMSProp optimizer that is built into PyTorch. Therefore, our neural network will be able to estimate the best possible Q value for an action given the current state by optimizing the rewards from previous experiences.

We discovered that our results are very dependent on the reward function that we define for each action. In our current implementation, the reward is determined differently depending on the action that was taken. If the action was to buy stock, then the algorithm has a flag which checks if there was stock bought in the past. If there was stock bought, then it gets no reward for trying to buy again, but if there wasn't, then the reward will be the difference between the next price and the current price. Therefore if the stock buys and it went up then the reward is positive and if it went down then the

reward is negative. If the action was to sell, then the algorithm checks whether stock was bought previously, and if it wasn't already bought then the algorithm also receives no reward. If there was stock bought previously, then the reward is 100 times the difference between the price it sold at and the previous price it was bought. Therefore, the reward for selling will be very positive if it sells higher than it was bought, and the reward will be very negative if it sells lower than it bought. The third possible action is to hold stock or do nothing. The reward function for holding checks if there was stock bought and if there has been no stock bought then it receives no reward for holding, but if there was stock bought, then it receives a very small positive or negative reward proportional to the difference between the incremental prices between the holding states.

Our algorithm was incorporated into a class which takes various hyperparameters of the algorithm as input. This functionality allows us to tune them and see the results much quicker. We did not have enough time to try and optimize all of these hyperparameters, but we developed a framework that allows all of these to be changed and optimized in the future. All of the hyperparameters and the default values that we set for them can be seen in the table below:

Hyperparameter	Default Value	Description
BATCH_SIZE	128	The number of training samples passed through our network at once
EPS_START	0.9	The starting value for ε
EPS_DECAY	200	The exponential decay rate of epsilon per iteration
EPS_END	0.05	The ending value for ε
NUM_EPISODES	10	Number of episodes to run training
GAMMA	0.8	The discount factor
hidden_layer	128	Number of hidden layers in the neural network
input_size	10	Number of past days to input as the state
TARGET_UPDATE	1	The frequency to update the target network

Results:

For the results of this investigation, reinforcement learning was used on Microsoft stock data for the past 800 days. The parameters used for training this was a batch size of 128, hidden layers of 128, $\gamma = 0.8$, $\varepsilon = 0.9 - 0.05$, and 10 episodes. The parameter of γ was set to 0.8 as γ is the parameter for future rewards. The focus for this study was long term strategies in trading due to short term trading fees. Therefore the value of γ was close to one to maximize long term trades. As for ε a exponential decay of ε from 0.9 to 0.05 was used. The reason for using an exponential decay pattern is to explore multiple different trading patterns and exploit these learned parameters later. Lastly, episodes was 10 as to allow for the algorithm to have multiple tries to create a strategy.

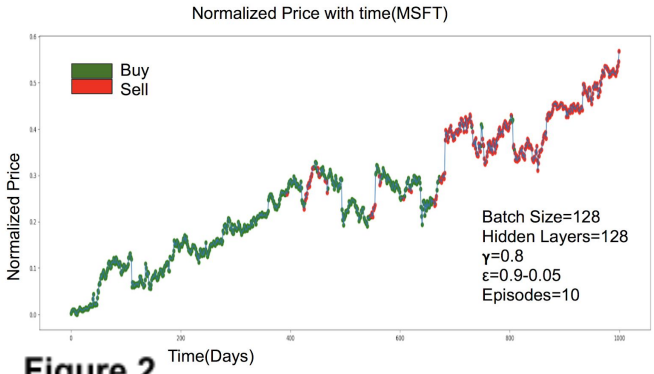


Figure 2

In [Fig. 2] the results highlight the strategy developed to buy and sell stocks. One interesting part of these results is the development of a common strategy of buy low and sell high. Visualizing the huber loss, the loss correctly decrease over iteration [Fig. 3]. Once the loss plateaued it started to become stochastic which is caused by the learned reward cycle per iteration as the network has learned 'an optimized' way to buy and sell stocks. Also shown is the cumulative reward with iteration, which highlights that the algorithm begins by decreasing reward with iteration although in the end is able to increase reward with iteration. This U shape in the reward with iteration can be due to the fact of the γ parameter which will take rewards in the short term that maybe negative, although will maximize the reward in the long term. Both the reward and loss plots indicate that the algorithm learned a strategy.

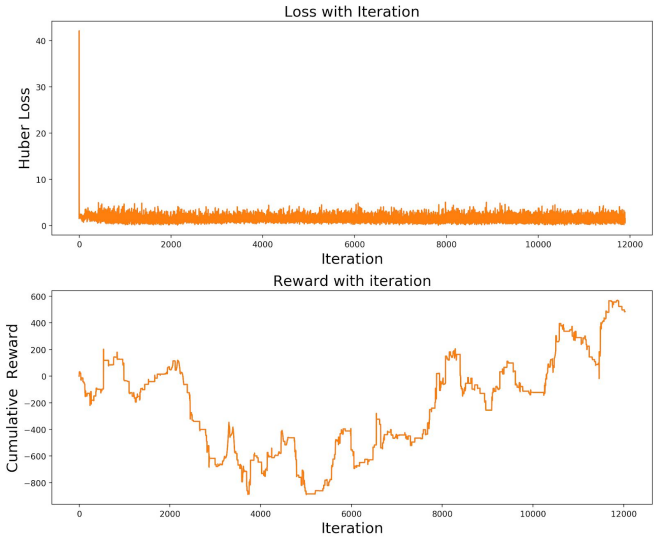


Figure 3

Stock (Trained on MSFT 800 days)	Profit(Normalized Units) >0 +
Microsoft(Technology)	4.45
GE(Industrial)	56.58
CBS(Media)	318.59
MMM(Manufacturing)	-17.98

Figure 4

Applying this strategy from the deep q network to other industries, GE in the industrial sector made 56.98(normalized price units), CBS in the media sector made 318.59 and MMM in the manufacturing sector lost 17.98[Fig. 4].

One possible reason for the profit distribution per industry could be due to the correlation between industries. For example, Microsoft's correlation is a direct buyer to MMM and thus not much relation. Although, Microsoft and CBS are correlated in CBS displays ads for Microsoft. This dependence structure could make the strategy learned easily transferable to correlated examples.

One other piece of work performed during this investigation was speeding up the code developed. Originally, the code developed would run in about 30s per episode running on a 2014 edition Macbook Pro. Some optimization performed in order to speed up this time was removing unnecessary parts of code in high hit loops and optimizing the data acquisition strategy. Through this method about 19 seconds was shaved off the training time of the network. Aside from applying these simple optimization, a GPU optimized approach was also performed. This study developed a version of the code optimized to run on a GPU using CUDA 9.1 and CUDNN 7.1. The resulting optimized code was run on a GTX 1070 and had a training time of around 3 seconds per episode. This allowed our study to try out multiple different hyper parameter steps in order to optimize the model. Note this GPU optimized code is based of the cart pole tutorial from pytorch.

Conclusion:

We developed a framework for a stock trading strategy using reinforcement learning techniques. Other work has been done using stock indicators as an input to a Deep Q Learning Network. By using raw input data as the input to the Deep Q Learning Network, we have developed a novel way to represent and solve this problem. We have provided a framework for future researchers to expand upon.

Our results show that for several sectors, our algorithm comes out to be profitable. However, there are several areas that offer room for improvement in the future. Some erroneous policy-action classifications (choosing an action that would result in a low actual reward) can be attributed to the noise in the input data. Future researchers could expand upon this design by applying different methods to deal with the significant amount of noise associated with working with raw data such as various filtering techniques.

Another area for improvement with the current implementation lies with the reward function used to determine the reward for selecting an action. A more accurate reward function would allow the model to optimize more efficiently because it would more accurately reward or penalize the model for making good or bad decisions, respectively. However, as it stands, our reward function offers us adequate results that are on average profitable.

Contributions:

Anthony Bisulco - In this project, my major contribution was developing some of the initial reinforcement learning code and making an easy to use data interface class. The data interface class that I developed gave access to over 500+ stocks and could also grab other stocks not in the database from Quandl. Additionally, I rewrote some of the code in order to allow our code to run on a GPU.

Jake Messner - I wrote functions in the financial data Python class to pull data from the Quandl API and remove incomplete datasets and data with errors. I also compiled five index funds representing the five most traded sectors and input this data as csv files into a nested dictionary in Python for training our algorithm.

Larry Thul - In this project, my major contributions were designing the reinforcement learning framework implementation in pytorch with Anthony and creating the reward function. There were many issues along the way because our loss was not converging after we wrote the algorithm and I also debugged the algorithm to finally get the neural network loss to converge and actually begin to start learning based on the experiences from each action.

Brendon Welsh - I contributed to the development of the RNN before we decided to use a Deep Q Network. During the research phase, I created the heatmap for the candlestick plots that we were using. I also created the wrapper class for the reinforcement learning python notebook. I was also responsible for most of the figure generation for the presentation.

Github Link: https://github.com/brendonwelsh/Machine-Learning-Final-Project
References:

[1] Moody, J. and Saffell, M., "Learning to trade via direct reinforcement," in IEEE Transactions on Neural Networks, vol. 12, no. 4, pp. 875-889, July 2001.

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

[3] “Reinforcement Learning (DQN) Tutorial.” Reinforcement Learning (DQN) Tutorial - PyTorch Tutorials 0.4.0 Documentation, pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

[4] “Reinforcement Learning (DQN) Tutorial.” Reinforcement Learning (DQN) Tutorial - PyTorch Tutorials 0.4.0 Documentation, pytorch.org/tutorials/intermediate/reinforcement_q_learning.html#replay-memory.