Independent Study (CS 604) Final Report

Trader Joe - A Stock Trading Bot

DECEMBER 16

By: Aayush Mandhyan (am2447)
Mentor: Professor Charles Wes. Cowan

Abstract

I aim to build a Reinforcement Learning Agent for trading stocks. First step in this process is to generate an environment for this agent to work in, this environment will be simulated using different simulation techniques and will contain stock ticker data. In the next step, agent will be trained in this environment using various Reinforcement Learning Techniques to make profitable trades. Finally, once the agent is trained, it will be validated using historic stock data of different firms registered with the New York Stock Exchange.

Introduction

The goal here is to build a Stock Trading bot using various techniques in Reinforcement Learning. Reinforcement Learning requires a vast amount of data for training. It is significantly larger than any amount of Stock data we can possibly find on the internet. So, the natural way was to simulate the data itself. After some extensive research I found that I could use Geometric Brownian Motion (a mathematical technique) to simulate the data required for training and implemented that using Python.

Once the simulator was built, next steps were to build a trading environment for the Trading bot to trade on. For this `StockMarket` module was built; to perform hold/buy/sell actions and keep track of current balance and the current stock count. Also couple of other utility functions were required to give a structure for the agent to learn. Such as `Environment` module to create windows for the stock ticker data, the `estimate_reward` module to estimate the maximum reward on a single duration (255 days i.e. a single market year) of the stock data & the `transform_inputs_delta` module which is used for feature engineering the environment data (windowed ticker data + current portfolio data) to return delta transformed inputs for the learning network.

For the actual learning part. As our environment consists of continuous action set, we used Deep Neural Networks for function approximation which takes current environment state as an input and provides us with the best possible action set which will maximize future rewards. For the architecture of the Deep Neural Network I used 2 hidden layers and 1 final layer this being constant for all the Reinforcement Learning algorithms. For Reinforcement Learning algorithms such as Deep Q-Learning, Double Deep Q-Learning, Deep Q-Learning with Replay Memory, Double Deep Q-Learning with Replay Memory, Actor-Critic Q-Learning with Replay Memory were used. At this point the best possible results are returned by Deep Q-Learning Networks with Replay memory using delta transform feature engineering and reward type 3 (explained later in detail). It achieved average reward of ~5k$ on initial balance of ~10k on 100 evaluations with only 30 evaluations had negative reward whereas the estimated reward for these evaluations were 30000. So overall, we ended up having quite significant amount of returns using trained Trading agent.

Methodology

Simulating Stock Ticker Data

Simulations of stocks and options are often modeled using stochastic differential equations (SDEs). Because of the randomness associated with stock price movements, the models cannot be developed using ordinary differential equations (ODEs).

A typical model used for stock price dynamics is the following stochastic differential equation:

$$dS = \mu S\,dt + \sigma S\,dW_t$$

where S is the stock price, μ is the drift coefficient, σ is the diffusion coefficient, and W_t is the Brownian Motion (https://en.wikipedia.org/wiki/Wiener_process).

In modeling a stock price, the drift coefficient represents the mean of returns over some time period, and the diffusion coefficient represents the standard deviation of those same returns.

The Brownian Motion W_t is the random portion of the equation. Each Brownian increment W_i is computed by multiplying a standard random variable z_i from a normal distribution N(0, 1) with mean 0 and standard deviation 1 by the square root of the time increment √Δti.

$$W_i = z_i\sqrt{\Delta t_i}$$

The cumulative sum of the Brownian increments is the discretized Brownian path.

$$W_n(t) = \sum_{i=1}^{n} W_i(t)$$

For the SDE above with an initial condition for the stock price of, the closed-form solution of Geometric Brownian Motion (https://en.wikipedia.org/wiki/Geometric_Brownian_motion) (GBM) is:

$$S(t) = S_0 e^{(\mu - \frac{1}{2}\sigma^2)t + \sigma W_t}$$

Used the above closed form solution to simulate the stock market data.

Deep Q-Learning

Deep Q-learning algorithm is a policy-based learning algorithm with the function approximator as a neural network.

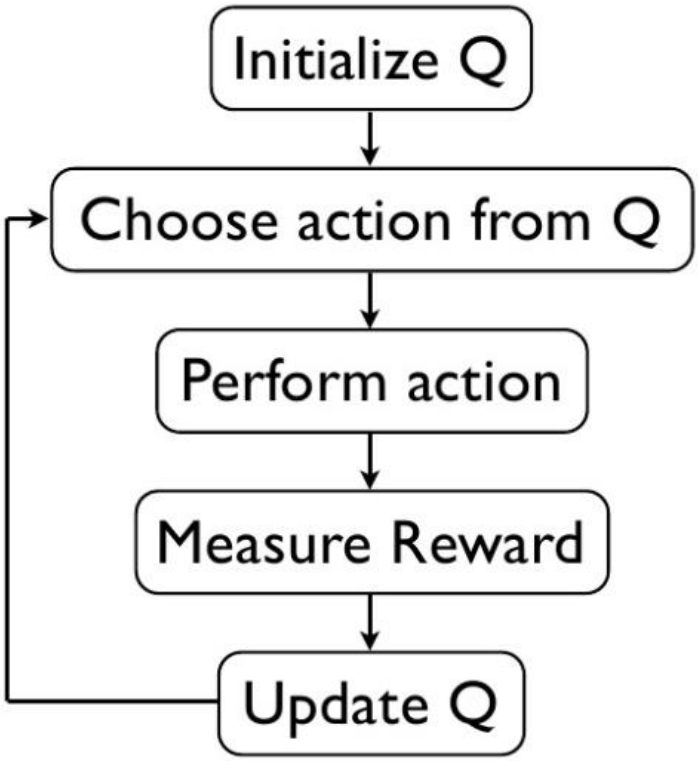
Steps of Q-learning:

1. Initialize the Values 'Q (s, a)' by initializing the Neural Network.
2. Observe the current state 's'.
3. Choose an action 'a' for that state based on one of the action selection policies (eg. epsilon greedy)
4. Take the action and observe the reward 'r' as well as the new state 's'.
5. Update the Value for the state using the observed reward and the maximum reward possible for the next state. The updating is done according to Bellman Equation.

$$Q(s,a) = r + \gamma(max(Q(s',a'))$$

6. Set the state to the new state and repeat the process until a terminal state is reached (end of duration in our case).

A simple description of Q-learning can be summarized as follows:



Important details for above implementation:

1. Architecture of DNN: Hidden Layer 1 (units=2*input_size), Hidden Layer 2 (units=input_size), Output Layer (5 units) this layer corresponds to 5 actions for every given play.
2. Inputs contain: Env (stock1 prices with 30-day window + stock2 prices for 30-day window) + current portfolio value (current_balance + stock1 count + stock2 count).
3. Output contain q-values for every action [hold, buy1, sell1, buy2, sell2].
4. Loss Function: Root Mean Squared Error.
5. Optimizer: Adam Optimizer.
6. Reward computed as: current_portfolio_value (current_balance + stock1 count * stock1 current_price + stock2 count * stock2 current_price) – initial balance.
7. Value of gamma (discount factor) is 0.9.
8. Action selection is done using explore (select random action) vs exploit (action with maximum q-value) strategy. The probability of explore is initially 0.9 which decrease by 0.1 for every 10% iterations.
9. Keywords used iterations meaning number of times we are doing certain experiment, duration is the time of stock prices taken per iteration; plays are duration – 32 which so because we have taken 30-day window for inputs in the DNN.

(take a look at the code for exact implementation details).

Deep Double Q-Learning

Double Q-Learning is like Q-Learning. The key features for Double Q-Learning over Q-Learning is that it uses 2 Q-Networks, so when computing q-value for an action the values outputted by both Q-Networks are taken and their average value is used as the final q-value. And while learning it updates one network at random per play. This kind of network provides some regularization to the resulting q-values. Hence the q-values has less variation though time.

Deep Q-Learning with Replay Memory

Deep Q-Learning with Replay Memory is an extension of Deep Q-Learning. It consists of a Q-Network whose current state, action, reward and next state values are saved in a Replay Memory (a queue with limited capacity) and used for training the Trading agent. So, in this method we take random sample from the Replay Memory for training; where as the values from every play i.e. current state, reward, action set, and next state are enqueued in the Replay Memory for further Training. This is a superior algorithm to simple Q-Learning as it requires lesser number of iterations to perform profitable trades.

Double Deep Q-Learning with Replay Memory

As the name suggests, Double Deep Q-Learning with Replay Memory is extension to Double Deep Q-Learning architecture and provides a Replay Memory for training in the same way as in Deep Q-Learning with Replay Memory architecture.

Actor-Critic Q-Learning with Replay Memory

Actor-Critic Q-Learning with Replay Memory is similar to Double Deep Q-Learning with Replay Memory. The difference here is that; here the two Q-Networks are one for Actor and second for Critic. The q-values returned by Actor network is used to find the optimal action but here the Critic network is used for learning. One this network has learned for certain number of iterations the Critic network (weights & biases) are copied over to the Actor network. Now the new Actor network is used to generate q-values and Critic network is used for learning. Again, the Critic network is copied to Actor network after certain iterations and this cycle keeps on going until we are done with set number of iterations for the training.

Evaluation Metrics

How can we judge if our model if performing good enough? For this we need to estimate the maximum profit can be made on a particular stock play. Premise for this being that not every stock play goes in upward direction but can also goes in downward direction as well. Hence, we need a measure to compare the output of our model with the true potential for making profit. For this `estimate_reward` method is built. Which finds the maximum expected profit by taking the maximum difference between a stock price at time t and stock price at time t + k (k being 1 to end of duration), for this maximum difference purchase the stock with maximum units possible at time t and selling them all at time t + k.

One we have this module in place we evaluate on 100 iterations by using exploit strategy and then compute the average reward and its standard error. Also, keeping a count of iteration with losses in above iterations.

Feature Engineering (Delta Transform)

Experiments we performed based on above Q-Learning algorithms were run in a simulated environment of two stocks. So, the action set required contained 5 values (hold, buy1, sell1, buy2, sell2). In this kind of environment, one can see that say a bigger firm which has not diluted there share has a higher stock price where as another firm which has diluted its shares and are selling their stocks at a low value, there is a possibility of generating a higher profit buy trading stock 2 in this case by buying higher quantity of share if there is a higher chance of profitability.

Training of DNN should be done keeping these scales in mind to give unbiased weightage to each stock. For this the stock price in the price window was scaled using the current balance, also delta price was computed and added to the inputs. By delta price I mean stock_1_t1 / stock_1_t0 ... stock_1_t30 / stock_1_t29 for a window size 30. Also, we incorporated the current portfolio value by computing current_balance + (stock1_count * stock1_price) + (stock2_count * stock2_price).

Reward computed per play also depends on scale factor hence we played around with 3 types of rewards. Type1: current_portfolio_value – initial_balance, Type2: current_portfolio_value/initial_balance & Type3: Log(current_portfolio_value/initial_balance). Out of these we saw the best performance given by Reward Type3.

Expanding Action Set

Until now we were performing actions for a single unit for a single stock per play. But real time stock trading does not work this way. One can chose to spend varying amount in various stocks or buy some and sell some or just don't do anything i.e. hold. This action set should be incorporated for our Trading Agent. For this there were slight modifications done to the DNN where for learning overall q-values we use input place holders and for learning the optimal action set for the optimal q-value we use tempX variable. Where for q-values we train for all trainable variables, but for action set we train only on tempX.

The action set created for this type of trading contain vector values as [phold, pbuy1, pbuy2, psell1, psell2]. Where all the p values should be between 0 and 1 & phold + pbuy1 + pbuy2 = 1. Here we consider phold value as percentage of current balance to hold, pbuy1 as percentage of current balance to buy stock1, pbuy2 as percentage of current balance to buy stock2, psell1 as the percentage count of stock1 to sell and psell2 as the percentage count of stock2 to sell. Thus, incorporating all possible actions sets for Trading stocks.

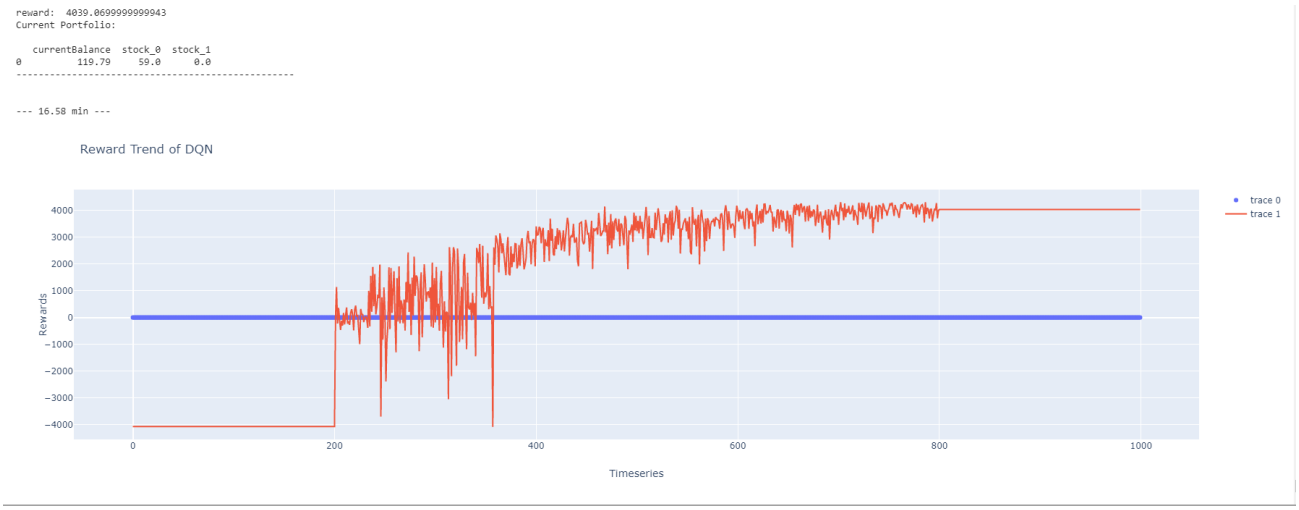
Results

In this section we will look at the experimental results of the above-mentioned methodologies:

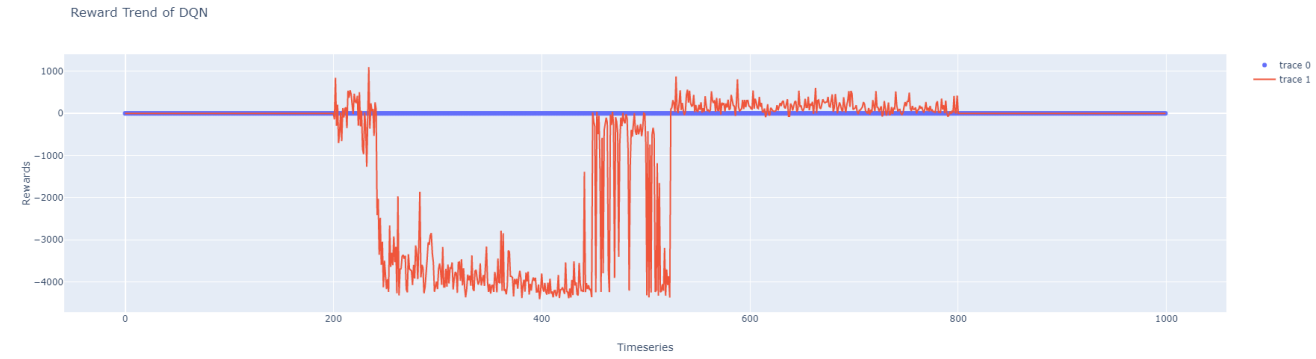
Deep Q-Learning

Initial attempts at DQN. These results are without evaluation function for 1000 iterations with Reward Type1 without feature engineering.

```
reward:  4039.0699999999943
Current Portfolio:

   currentBalance  stock_0  stock_1
0          119.79     59.0      0.0
--------------------------------------------------

--- 16.58 min ---
```

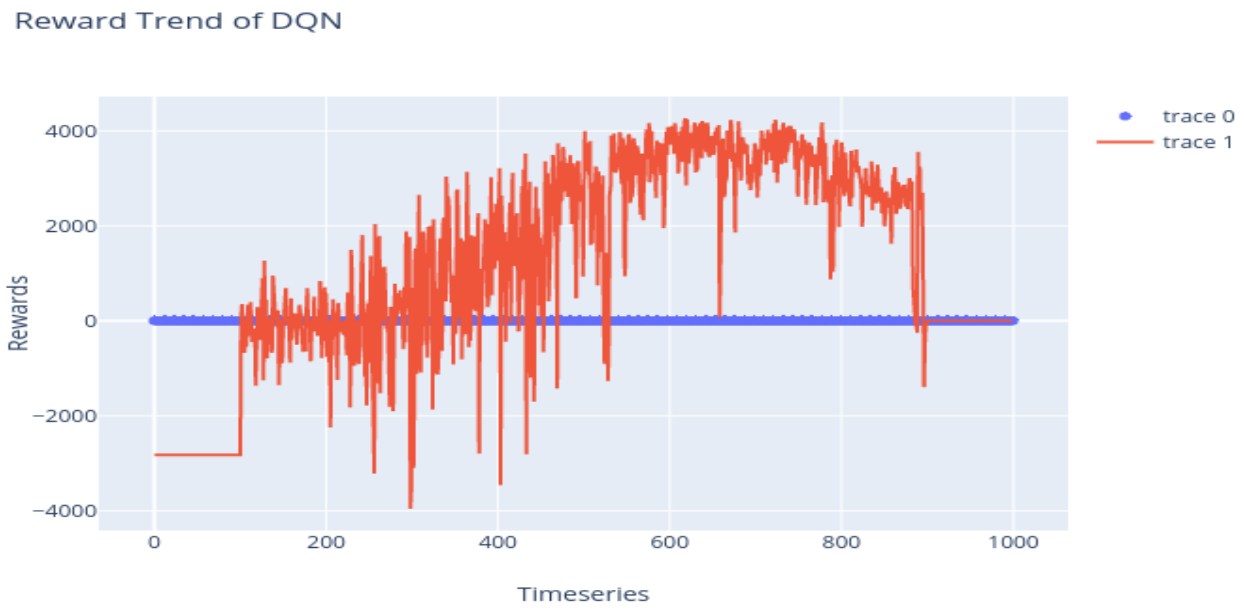


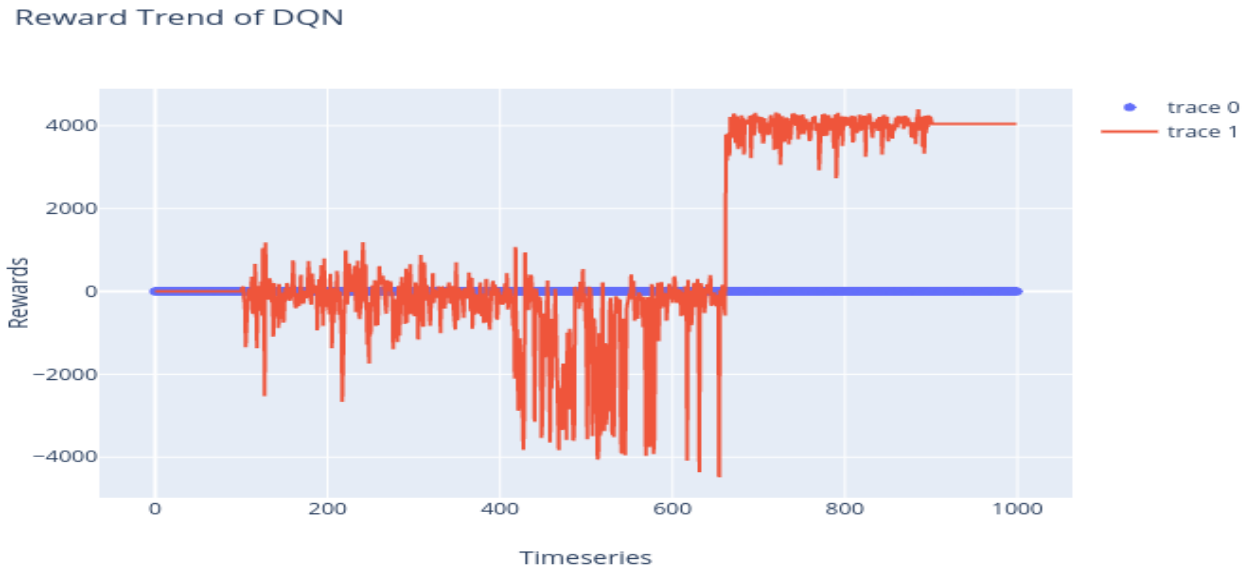

```
   currentBalance  stock_0  stock_1
0         10000.0      0.0      0.0
--------------------------------------------------

--- 15.38 min ---
```



Double Deep Q-Learning

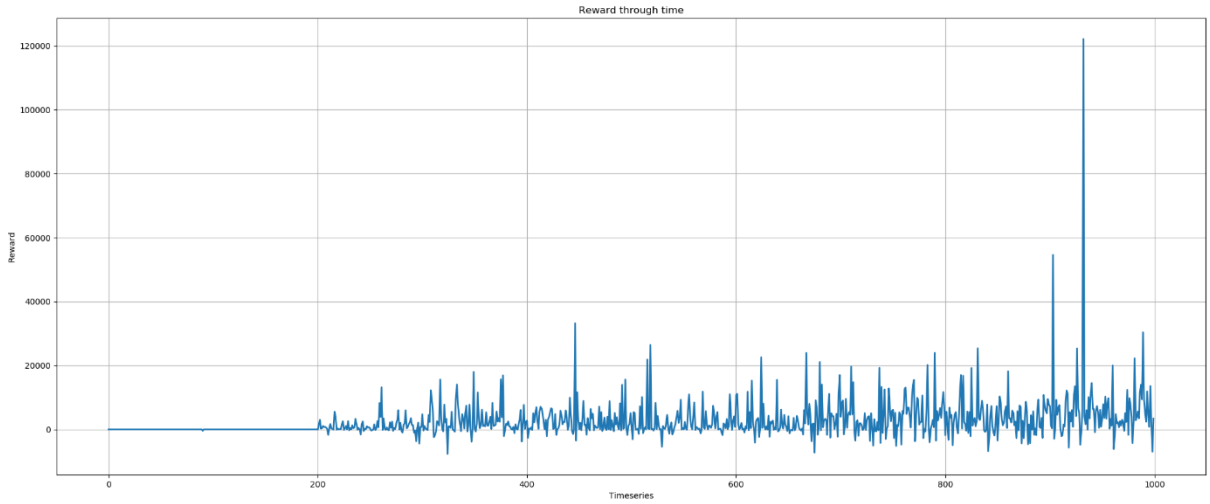
Initial attempts at Double DQN. These results are without evaluation function for 1000 iterations with Reward Type1 without feature engineering.

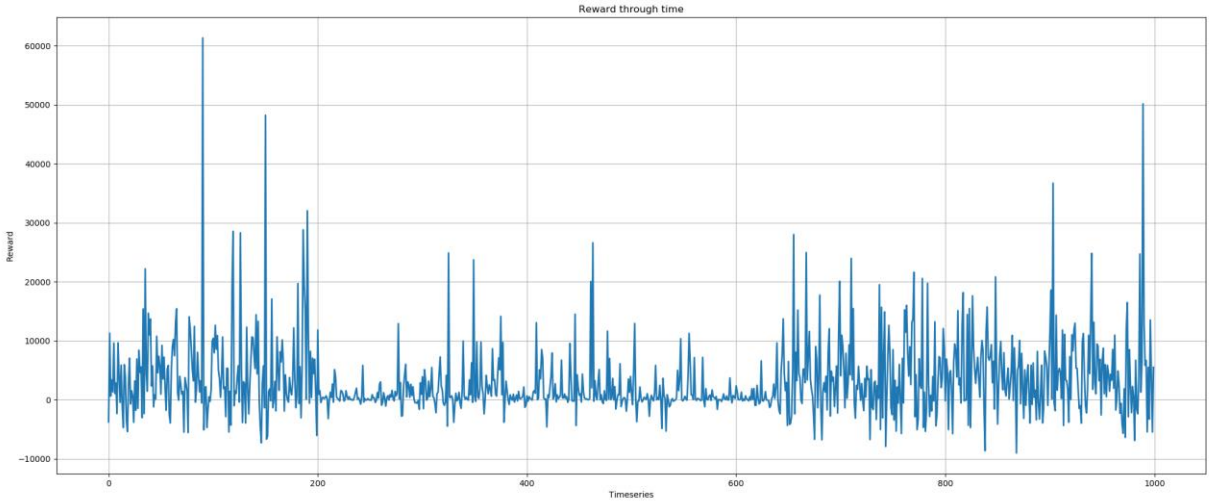




Deep Q-Learning with Replay Memory

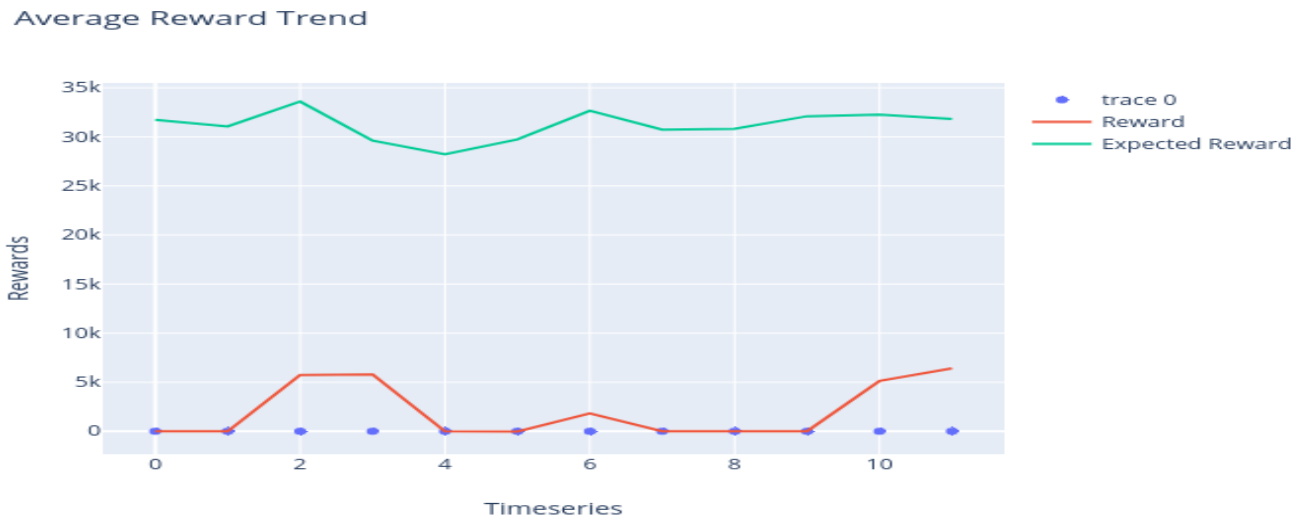
Attempts at DQN with replay memory. These results are without evaluation function for 1000 iterations with Reward Type1 without feature engineering.



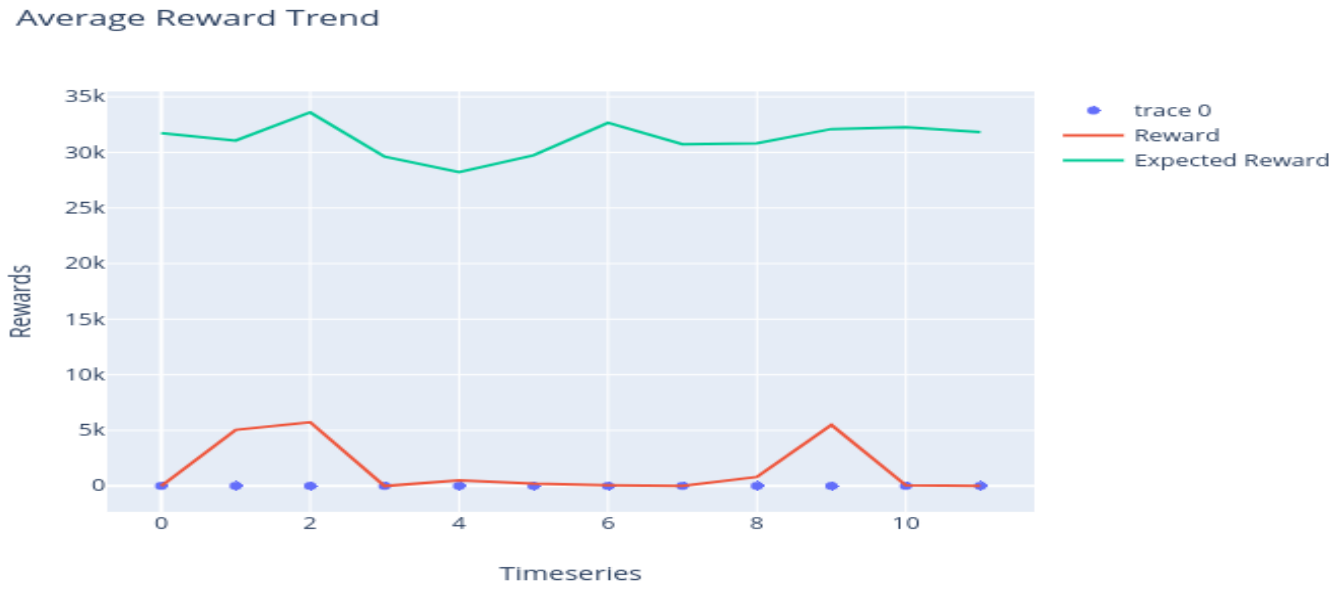


Deep Q-Learning with Feature Engineering

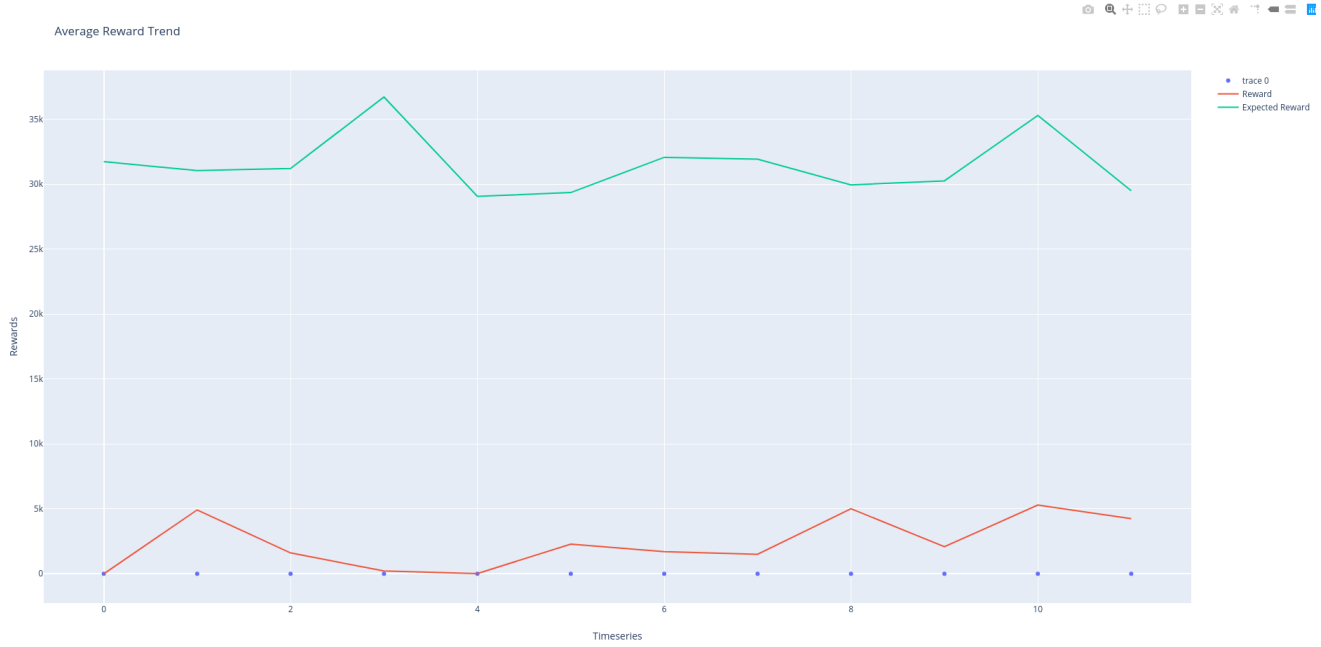
These results are for 1000 iterations with Reward Type1. Evaluated for 100 iterations at every 100 iteration.



These results are for 1000 iterations with Reward Type2. Evaluated for 100 iterations at every 100 iteration.

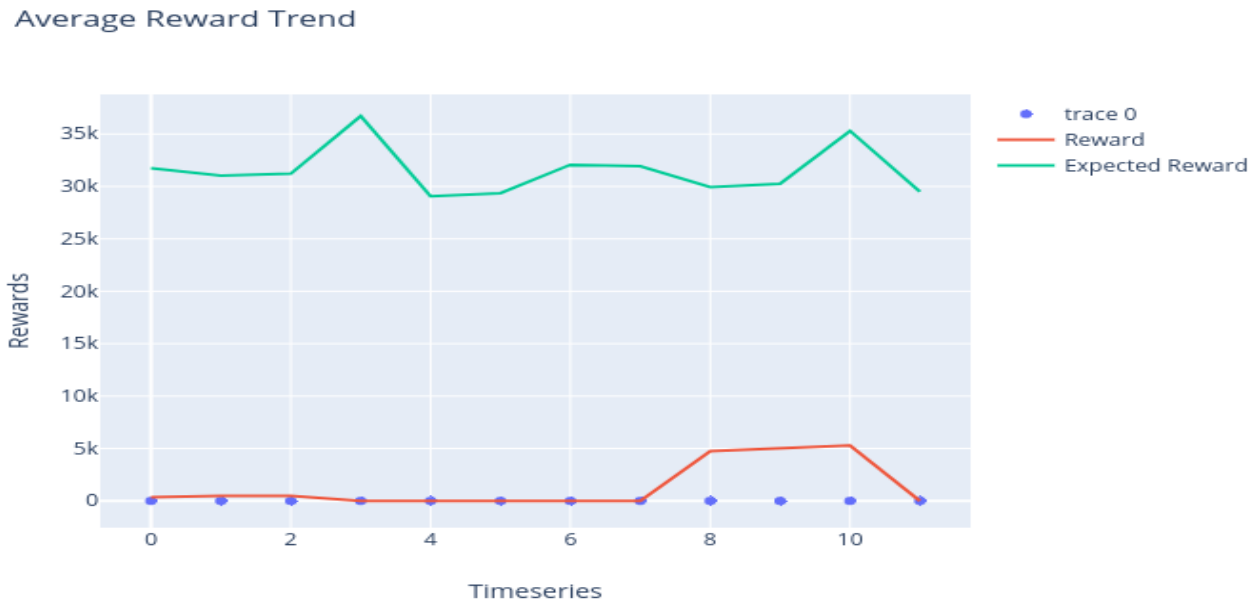


These results are for 1000 iterations with Reward Type3. Evaluated for 100 iterations at every 100 iteration.



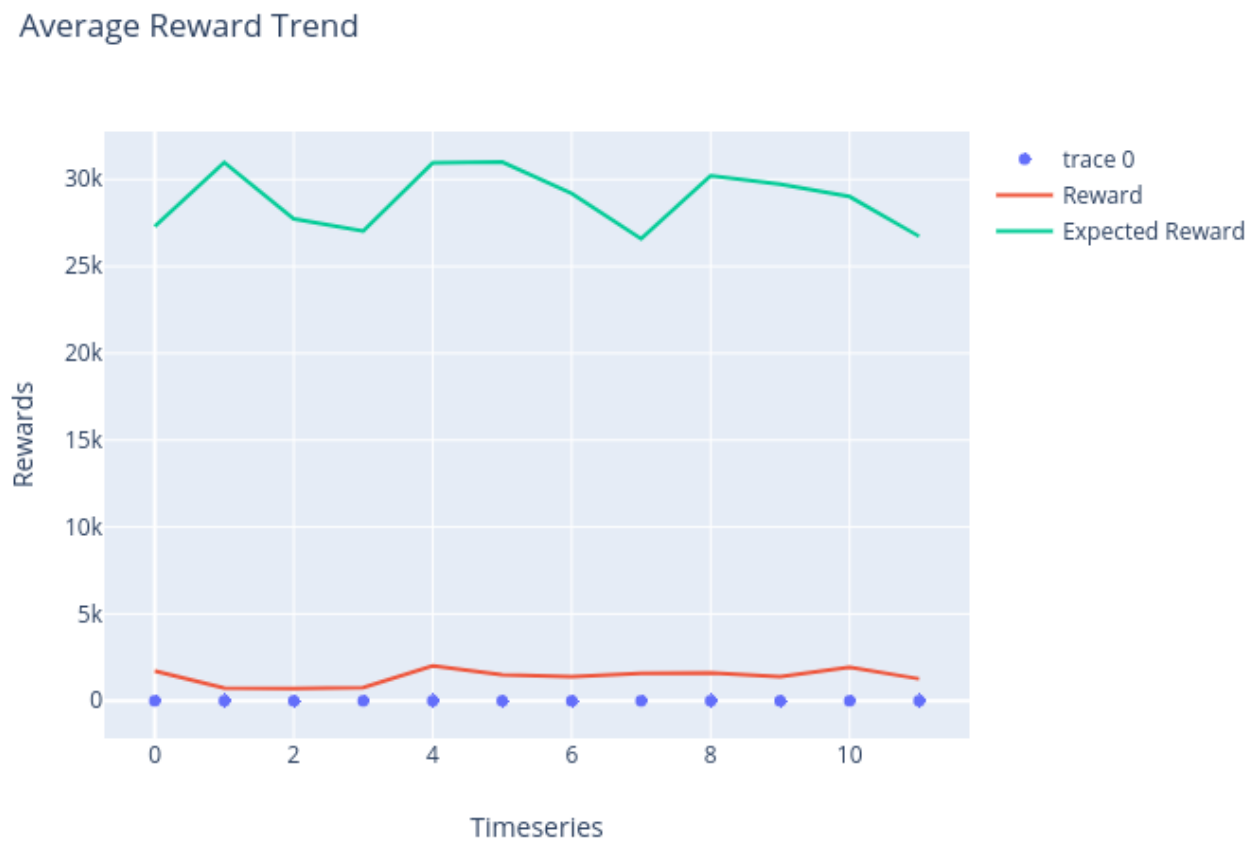
Actor-Critic Q-Learning with Feature Engineering

These results are for 1000 iterations with Reward Type1. Evaluated for 100 iterations at every 100 iteration.



Deep Q-Learning with Feature Engineering and Expanded Action Set

These results are for 1000 iterations with Reward Type3. Here duration=50 i.e. plays=18 (reduced from 255 for compute constraints). Evaluated for 100 iterations at every 100 iteration.



Conclusion

What worked and what helped us explore new ideas? We started out with using a simple DQN (Deep Q-Network) without any form of feature engineering and proper evaluation module. At this stage we were evaluating the model by only looking at the profit incurred on initial balance. This method showed promise for learning Trading and had profitable trades, but it left so much more to be desired. Next steps were to explore other Reinforcement Learning algorithms such as double DQN, actor-critic DQN, DQN with replay memory, double DQN with replay memory and actor-critic DQN with replay memory, etc. to see if they perform better at Trading. Out of these, DQN with replay memory had the best performance.

Once we have figured out that the best performance is given by DQN with replay memory. Still we were getting ~3k profits where the expected profits were ~30k on an initial balance of 10k. We started exploring other ideas, such as scaled inputs, scaled reward functions, etc. to provide same equal weightage to all stocks to increase profitability of our Trading agent. Incorporating these changes improved performance by a significant amount, we started seeing result up to ~5k for the same initial balance. We learned that although the same patterns could be learned by the DNN on its own but providing transformed inputs cuts the learning time by a significant amount and we see significant profitability sooner than later.

Lastly, we looked into expanding the action set where we consider performing multiple action in any combination deemed optimal by DNN in a single play. Thus, bringing into more realistic approach to stock trading.

There is a lot we can delve into, and we have discussed those ideas. Due to time constraint we are leaving the work at this stage with couple of future topics which should be looked into.

Scope for Future Work

- Using RNN's LSTM as part of our DNN architecture.
- Incorporating news sentiment score as an input to the DNN.
- Adding another DNN to output optimal action-set for expanded action set version.

References

List of references:

- https://jtsulliv.github.io/stock-movement/
- https://www.analyticsvidhya.com/blog/2017/01/introduction-to-reinforcement-learning-implementation/
- Code and knowledge shared by you Professor.

Thank you

I would like to thank you Professor for the mentorship and guidance you have given me in the span of this course. I have learned the concepts behind various RL techniques and implementing them in TensorFlow all of this would not have been accomplished without your help. Let's hope to build on it in future and sit home and let our Trading Agent earn our livelihood :D.