ECEN 765 - Reinforcement Learning for Stock Portfolio Management

Harish Kumar

Abstract

In this project, my goal was to train a reinforcement learning agent that learns to manage a stock portfolio over varying market conditions.The agent's goal is to maximize the total value of the portfolio and cash reserve over time. The first part of the Project was to model the problem as a Markov Decision Process. After doing this, I used Q-Learning to form an optimal policy over this process. Various challenges were encountered during training - the agent had no obligation to learn when it can buy/sell, the agent was unable to learn multiple tasks at the same time, the performance of the agent underwent unbounded oscillations, etc. and this report describes how each of those challenges were surmounted using various techniques. In the end, I used a Double Deep Q-Network as the function approximator for the agent and managed to achieve performance that is a lot more superior to what previous attempts using reinforcement learning managed. A caveat is that the model is still quite risky and is not suited for trading with real-world money.

I. Introduction and Motivation

ANY kind of repeatable advantage that one can get in the world of stock trading can easily translate into profits of millions of dollars, considering that one can repeatedly collect the payoff on that particular advantage. Being able to predict which way the market is going to move has been a gift of the most successful investors all over history. Technology can be used in different ways to assist humans in this prediction - either to generate useful indices and markers that can be manually analyzed, or to directly predict the market movement. In this project, I examine whether a software agent can learn to directly trade profitably on its own.

Beyond all the technicalities, the simplest way to state most successful trading policies is "Buy low, Sell high". In other words, a trader is expected to predict peaks and valleys in the price of a stock, buy the stock when it is at a valley, and then sell it at a peak. A simple repetition of this process would result in enormous gain for a trader, irrespective of the starting capital. However, the primary challenge here lies in being able to identify the peaks and valleys correctly, as well as accommodate for the possibility that what the trader predicts to be a valley may not actually be one(thus causing losses) and what the trader predicts to be a peak may not actually be one(thus causing reduced profits).

In reinforcement learning, we hope to learn a "policy", i.e. a methodology of making decisions based upon the present state of things. These decisions should be made such that some cumulative reward is maximized over the long term. This is especially applicable here as the trader is essentially working in a highly changing environment where the right decision depends upon the state of the environment, and in the process hopes to maximize the total value of the portfolio over time.

II. Problem Description

The problem is as follows: An agent starts with a fixed capital of say, \$1000 (C_0), as well as data on past prices of various stocks, their past prices over a wide time period as well as optionally other fundamentals(Quarterly results, dividends, etc). It should make buy/sell/hold decisions every day such that the portfolio value V (cash reserve + present value of stocks) is maximized over the long-term. To simplify the problem and make it tractable, I restrict the set of buy/sell/hold decisions to be on a select group of N stocks.

The training data can consist of time series data of stock prices, stock fundamentals and macroeconomic indicators(Inflation, GDP, etc). Essentially, the agent needs to be able to understand how the stock price datapoints will vary in the future and take actions accordingly to maximize its portfolio.

III. Mathematical Modeling

I choose to model this problem as a Markov Decision Process. Here, we have a set of states S, a set of actions permitted for each state A_s(and a global set of actions A), a reward distribution $R(s_t, a_t, s_{t+1})$ and a set of transition probabilities $P(s_t, a_t, s_{t+1})$.

A. State

Each state s_t consists of the following information:

- The numbers of shares of the N stocks that can be held by the agent: $n_1, n_2, ...n_N$.
- The prices of the N stocks: $p_1, p_2, ...p_N$ at time t.

- The most recent fundamentals for each stock: $\{f_{11}, f_{12}, ...f_{1K}\}, \{f_{21}, f_{22}, ...f_{2K}\}, ..., \{f_{N1}, f_{N2}, ...f_{NK}\}$. There are K numbers that describe the fundamentals for each stock. Fundamentals are numerical values that attempt to estimate the intrinsic value of a stock. One example is the company's profit margin over the last quarter.
- Weekly and monthly average stock prices over the last 4 weeks and 6 months respectively.
  $W = \{p_{1,week1}, p_{1,week2}, ...p_{1,week4}\}, \{p_{2,week1}, p_{2,week2}, ...p_{2,week4}\}, ..., \{p_{N,week1}, p_{N,week2}, ...p_{N,week4}\}$ and
  $M = \{p_{1,month1}, p_{1,month2}, ...p_{1,month6}\}, \{p_{2,month1}, p_{2,month2}, ...p_{2,month6}\}, ..., \{p_{N,month1}, p_{N,month2}, ...p_{N,month6}\}$
- The set of permitted actions for the present state, $a_1, a_2, ...a_{nActions}$. This is encoded as a set of binary variables that correspond to every possible action, which take a True value if that action is permitted and a False value if the action is not permitted.

The final item was added to reduce the difficulty of the problem and it was observed that this additional information greatly improved the learning speed of the model. This is since the model just has to learn that all actions that have a permission input of 0 will result in large negative penalties.

B. Action

The global set of actions A is different from the set of actions permitted to us in each state(A_s). This is since

- We cannot sell more units of a stock than we have in our portfolio.
- We cannot buy more units of a stock than we have cash for.

Thus, the present state s_t that the agent is in decides the set of permitted actions A_s. [6] uses a straightforward approach to simplify this set - they decide to greedily buy as many shares of a stock as possible with their cash reserve, and sell all the shares they have at once if they decide to sell a stock. They do not worry about violating the above two conditions and simply ignore erroneous decisions. While this approach greatly simplifies the action space, it produces a highly risky and volatile agent due to its inability to hedge bets on different actions(e.g. sell some shares of a stock if there is a moderate probability of it going down, but retain some more shares since the agent is not fully sure). Their experimental results validate this, and my experimental results show that having an expanded set of actions does reduce the agent's risk.

I define the agent's actions in the following approach:

- Having a separate action for selling $1, 2, 3...$ shares would cause an explosion of the action space, and even in this case, we will have to bound the maximum number of shares the agent can sell to keep the action space finite. Instead, we choose to have actions that decide what **proportion** of shares of a stock the agent is selling, i.e. we could choose to sell 25%, 50%, 75% or 100% of our Microsoft shares.
- For buying stocks, I follow a similar approach and have actions to use 25%, 50%, 75% or 100% of the agent's cash reserve to buy shares of a particular stock. In case the agent decides one of these actions and we have insufficient funds to buy even a single share of a particular stock, I return a large negative reward to discourage that action.
- I have one action for holding our portfolio for the day.
- I force the agent to buy *or* sell on a single stock per day, i.e. the agent cannot buy/sell shares from different stocks on the same day. This simplification will not cause significant reductions in the freedom of the model as we are not looking to make short-term gains.

Put together, there are a total of $4N(sell) + 4N(buy) + 1(hold)$ actions possible. Here, N is the total number of stocks that we are focusing on.

C. Reward

For the immediate reward, I primarily use the growth in the portfolio value V from day t to day $t + 1$. There are several other rewards and penalties that I have added since it aids in the learning process in some way or the other. As often described in reinforcement learning experiments, at present, crafting good rewards for an RL problem is more art than science([6]).

- I provide a large penalty(-8) in case the agent tries to take forbidden actions(such as trying to buy shares for which it does not have enough cash reserve).
- I provide the growth of the portfolio value from the initial capital as a reward. This encourages the agent to hold its shares when it is at a high net value even though the incremental growth is zero. It also has the additional effect of encouraging the agent to try and get out of past losses it has made.
- I add a small reward for every buy/sell decision that the agent makes. This encourages exploration all throughout the learning process. It has the disadvantage of increasing the risk of the agent, but if I make the reward small enough, it will become insignificant when the agent has learned to make sufficient profit. It does turn out in the end that the model is quite low-risk and this shows that adding in this reward does not negatively affect the risk of the model.

In practice, I observed that the agent is very quickly able to learn to avoid making invalid trades(in the first 60 episodes), and then is able to move onto maximizing profit. Thus, adding the invalid trade penalty does not destructively interfere with the profit maximization objective. The rapidity in learning this is due to the large penalty I added for making a violating action, as well as since I directly fed in inputs that exactly tell the network what actions are not permitted.

Due to the presence of this supporting reward, the agent is able to focus most of the training effort on learning to maximize the actual profit and not just on learning valid actions.

D. Objective

Given the above states, actions and reward definitions, we wish to maximize the total cumulative reward over time.

$$\max E[\sum_{t=0}^{\infty} r_t] \tag{1}$$

In a Markov Decision Process, to represent the trade-off between higher short-term rewards and higher long-term rewards, we use a parameter $\gamma \in [0, 1)$. The MDP objective is then

$$\max E[\sum_{t=0}^{\infty} \gamma^t r_t] = \max E[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})] \tag{2}$$

In practice, low values of γ resulted in a cautious agent that was not willing to risk its present portfolio value, and thus, the average cumulative reward was almost always less than 5% higher than the starting capital. It was also for the most part not much lower than the starting capital.

On increasing the value of γ, the cumulative reward had higher variation(due to the increased risk), but the mean cumulative reward was also higher.

IV. Testing and Evaluation

Testing and Evaluation is done as follows. At step(day number) t,

1) Provide the agent with day t's stock prices from the test data and a cash reserve, then get a predicted *optimal* action according to the learned policy.
2) Take the predicted action, then take the transition to the portfolio state on the next day. Revise the stock prices according to the test data to the next day's stock prices.
3) Repeat 1 and 2 over many days, and track the value of the portfolio over time.

This value should grow over time. At the bare minimum, the value after T(a large number) timesteps must not be less than $V_{baseline}$

$$V_{baseline} = C_0 \times \min(\frac{1}{N} \sum_{n=1}^{N} (p_{i,t} - p_{i,0}), 1) \tag{3}$$

This baseline is the minimum value out of "Not investing any money"(C_0) and evenly investing in the N stocks at $t = 0$ and making no further trading decisions. If our model performs worse than both these values, then it is useless.

We have stock price data from a time period of 17 years, from 2000 to 2017. Thus, I trained and tested on different time periods to ensure that the test data does not have any intersection with the training data. I trained the model on the first 3000 days and tested it on the next 1000 days. There are significant differences in macroeconomic conditions between these time periods, and it is this exact difference that we want to simulate to see if our agent is able to learn a good trading strategy.

V. Learning Algorithm

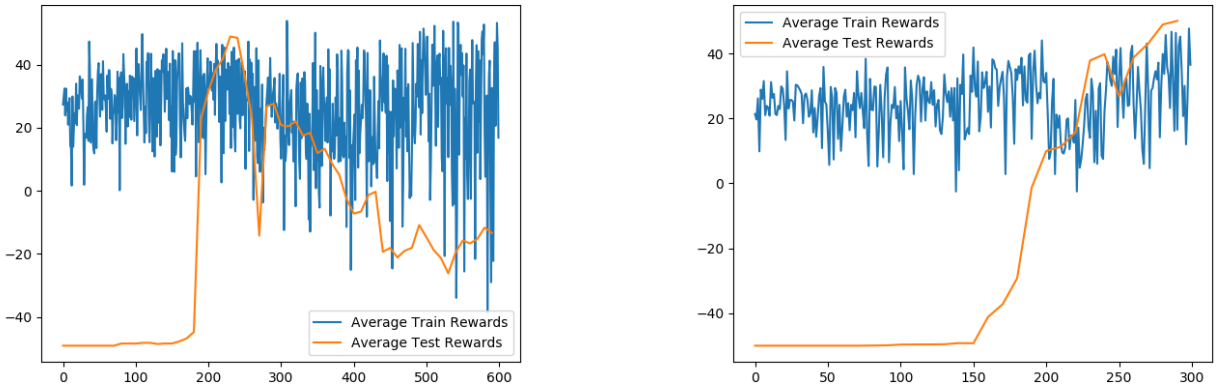
The agent knows neither the state transition probabilities $P(s_t, a_t, s_{t+1})$, nor the reward distribution $R(s_t, a_t, s_{t+1})$, and thus, we cannot use dynamic programming to solve the problem.

Hence, I use Q-Learning([8] and [7]) to learn a policy over the Markov Decision Process described above. Instead of using a table to store the Q-Values, we use a deep neural network to learn the Q-Values for different state action pairs. This has the added advantage of using function approximation to predict the Q-Values for unseen state-action pairs, thus speeding up training.

I chose not to use a recurrent neural network here as the temporal information has already been provided as part of the weekly and monthly average prices in our state vector, and since it is a highly condensed input with very low dimensionality. However, it is possible that an RNN performs well on this data.

An advantage of using a neural network is that we don't have to discretize the prices to get a trainable set of State-Action pairs and corresponding Q-Values. We can instead allow the neural network itself to interpolate between nearby states since the approximation function is continuous. Previous work has discretized the stock prices into a discrete state space and I argue that such an approach makes the problem much more complicated than it should be. This is since we are transforming a set of purely numerical values such as prices into a discrete state space where there is no notion of ordering between different states.

It must be noted here that only the state-space is continuous and not the action space.



(a) Deterioration of performance when using a deque-type memory. The network forgets how to perform permitted actions.

(b) Learning to buy and sell only when it is a permitted action. Valid action rewards and invalid action penalties are set to very high values.

Fig. 1: Problems faced and Solutions Implemented

VI. Problems Faced and Solutions

In this section, I describe some of the major problems and pitfalls that I faced during this project along with what solution I used to overcome these problems.

A. Unreliable initial Q-Values

As the neural network is just beginning to be trained, most of the Q-Values it returns are erroneous. Using these as target Q-Values will result in all the Q-Values converging to very similar values and make the different actions practically indistinguishable, while in reality, they have widely varying utilities.

[1] provides a solution to this problem where we start with a small discount factor, and hence reduce the impact of the poorly trained neural network on the learning process. As the network becomes better at predicting near-term utility of a state-action pair, we gradually increase the discount factor to its true value. Effectively, we increase the horizon over which we wish the neural network to foresee utility. This makes intuitive sense as predicting the utility of a state-action pair for the short term is easier than predicting its long-term utility.

B. Agent not knowing permitted actions

The Q-approximator neural network takes a state as input and predicts the Q-Values for every action in our Action space. Some of these actions may not even be permitted, but we are restricted to using a network of fixed output dimension for our agent, and hence cannot alter the network shape based on the state. Ideally, we would expect the Q-Learning process to discover that forbidden actions have a large negative reward, and then also learn what actions are forbidden for each state. In practice, this did not happen easily and the network took a long time(350 episodes) to even learn which actions are forbidden. However, the fact that it did, without any supervised guidance on which actions are forbidden is an attestation to the fact that the whole system is capable of learning ”something”. **Fig. 1b** shows the progress of the agent on this task.

To make the agent quickly learn how to take only permitted actions, I provide a supervisory indicator input that tells the network which actions are permitted. These are calculated by simple deterministic rules from the current state: for instance, if the agent has no shares of stock A, then selling any proportion of stock A is an illegal action, and hence the corresponding input will be 0. Such an input can in no way reduce the operating freedom of the agent, nor is it noise. Thus, it will not disturb the learning process negatively, and thus can only have a positive effect on the agent, or in the worst case, be ignored.

C. Oscillatory behavior

A common problem with training a DQN is an unbounded oscillatory behavior in the policy’s performance. This happens due to the following circumstances:

- The agent temporarily overfits the recently observed Q-Values, and thus its predictions of Q-Values for new state-action pairs becomes erroneous. This is since the recently observed Q-Values are all correlated with each other.
- These erroneous Q-Values cause the wrong actions to be chosen during exploration, thus causing the policy’s performance to become much poorer than that of the recent policies.
- These sub-optimal decisions have an exploratory effect on the model and reduce the overfitting in the neural network.

- Due to reduction in overfitting, the model performance improves again.

The unfortunate fact is that this oscillation is not a stable, decaying oscillation, but an unstable, divergent oscillation that causes the model to often forget whatever it has learned over the last few hundred episodes.

The solution to preventing this oscillation is two-fold:

- Double DQN: Use a second neural network to provide the target Q-Values we use in the Q-Value update equation and update the second neural network periodically.
- Experience Replay: Sample from all past experiences and train on this random sample. This breaks correlation in the data and makes it i.i.d.

In the Double DQN approach, the action is chosen based upon the learner DQN's predictions, while the Q-Values used for getting the optimal future value for a state s_t are obtained from a second target DQN. The target DQN is periodically updated to the learner DQN.

D. Catastrophic Forgetting

Catastrophic forgetting is a phenomenon in neural networks where the performance on training datapoints that haven't been observed for a long time greatly decays. In other words, the neural network greedily trains itself to perform well on recently observed data and forgets how to classify/regress on data that was observed long ago. A common solution to catastrophic forgetting in supervised learning is to shuffle the data and ensure that data of every "kind" is observed at least once in a while.

However, in reinforcement learning, our observations are sequential, i.e. the nature of the exploration process itself causes datapoints to be localized to one area of the input space. A naive solution is to store every state-action pair observed so far and train on a random sample of this dataset. However, we will quickly run out of memory if we do so. Furthermore, the probability of a recent and more relevant experience being fed to the neural network decreases as a rational function($\frac{1}{x}$) of the number of episodes we have trained so far, and thus, agent training is going to greatly slow down if we take this naive approach.

Most approaches I've seen so far use a "deque" to maintain a fixed memory where the oldest experiences are discarded as new memories come in. However, this caused the neural network to forget much of what it learned during the exploratory phase and caused it to overfit to the training data as ϵ became smaller and smaller.

[5] uses reservoir sampling to randomly forget some experience every time a new experience comes in, and demonstrates that this methodology helps in reducing catastrophic forgetting.

The solution I used was to maintain a memory of fixed size and whenever it is full, we make the agent "forget" a small percentage of its total memory to make space for new memories. This way, the probability of a recent experience being replayed is much higher than that of a very old experience, but there is still a good chance that a few representative experiences from every phase of training remain in the agent's memory. Effectively, it is very close to reservoir sampling, but the computational cost is much lower.

VII. Training Process

Training happens through a series of episodes. Each episode cycles through stock data from a specific contiguous time period, and the following steps happen in an episode:

1) Observe the current state s_t and get the Q-Values for all actions in the corresponding state-action pairs, $Q(s_t, a_{t1})...Q(s, a_{t|A|}$.
2) Pick the action with the highest Q-value with probability $1 - \epsilon$, or uniform randomly select an action with probability ϵ. This is the classic ϵ-greedy approach to exploration.
3) Take the action chosen in **Step 2**. This results in a transition to state s_{t+1}, with an immediate reward of r_t.
4) Update $Q(s_t, a_t)$ using the newly obtained reward and the transitioned state.
5) Repeat steps 2, 3 and 4 until the end of the episode.

After the episode ends, we use the Q-Values for all actions in all the states observed so far to train the function approximator neural network.

The Q-Value is usually updated as

$$Q^{t+1}(s_t, a_t) = (1 - \alpha)Q^t(s_t, a_t) + \alpha(r_t + \gamma \max_i Q^t(s_{t+1}, a_i)) \tag{4}$$

However, in practice, as we do not use a table to store our Q-Values, but instead a function approximator, older Q-Values are generally unreliable when compared to newer ones. Thus, the update equation reduces to

$$Q^{t+1}(s_t, a_t) = r_t + \gamma \max_i Q^t(s_{t+1}, a_i) \tag{5}$$

A very high-level description of the training process is given in **Algorithm 1**.

Algorithm 1 A high-level pseudocode for the learning process

```
procedure AGENT-TRAINER(G)
    MDP ← MDP for the given data and actions        ▷ This takes care of transitions, yielding rewards, etc.
    for i in 1 to nEpisodes do
        s_t ← Initial state                          ▷ Randomly chosen time-point from the stock data
        for j in 1 to nDays do
            Q'(s_t, a_1), ...Q'(s_t, a_|A|) ← Learner-DQN.predict(s_t)      ▷ From the learner neural network.
            a_t ← Sample-Action(Q'(s_t, a_1), ...Q'(s_t, a_|A|))           ▷ Action sampled using ε-greedy approach
            s_t+1, r_t ← MDP.act(s_t, a_t)           ▷ MDP returns both next state and the immediate reward.
            Q(s) ← max(Target-DQN.predict(s))        ▷ Second neural network part of the Double-DQN model.
Predicts for all observed states
            Q(s_t, a_t) ← r_t + γ × max(Q(s_t+1))    ▷ Update equation for Q-value
        end for
        Learner-DQN.train(s, Q)                      ▷ Train for all observed states
        if episode % refreshTimePeriod == 0 then
            Target-DQN ← Learner-DQN                 ▷ Make a copy of the weights.
        end if
        Update-Parameters()                          ▷ Update the values of γ, ε, etc.
    end for
end procedure
```

VIII. Dataset

In this project, I used the S&P 500 stock dataset([4]) that describe the movement of publicly traded stocks over several features. The NYSE dataset([2]) contains fundamentals for each stock, but the time durations of the two datasets are different. For expediency, I had to not use any fundamentals or macroeconomic indicators during training and just use the stock prices as inputs to my model.

IX. Implementation

- I implemented an MDP framework in Python from scratch. I could have used an existing library called `gym`, but this was an interesting coding exercise for me.
- The Neural network was implemented using Keras with Tensorflow as the backend.
- Plotting was done using `Matplotlib`.
- I used my Laptop to train the model. The hardware specification is CPU: Intel i7 4720HQ, RAM: 8GB, Graphics Card: NVidia GTX 965M.

Training took about 0.98 seconds per episode of length 200 days.

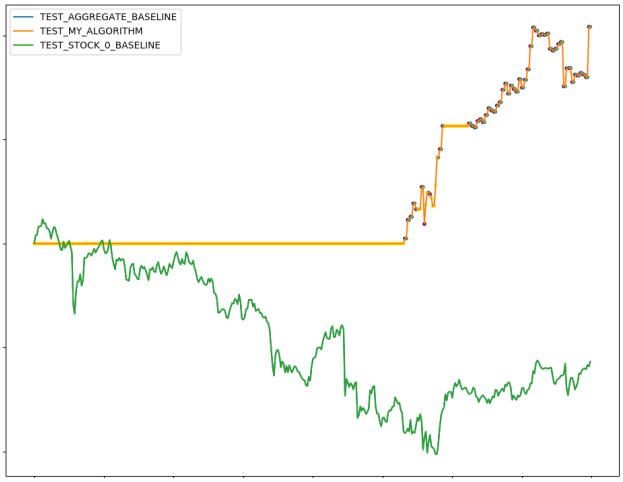
X. Performance and Comparative Analysis

The policy learned by the agent is generally low-risk, medium-yield. I made 500 trials on data not previously seen by the model at all - this was from a different time period altogether. On this data, an invest-and-forget strategy would have made a profit in 58.8% trials and losses in the rest, with an average loss of 7.46%. The number of trials where the trained agent made a profit is $432(86.4\%)$, and the average profit is 10.7% in these instances. The number of trials where the trained agent made a loss is $68(13.6\%)$, and the average loss is just 4.27% in these instances. The number of trials where the model made a profit of more than 15% is $86(17.2\%)$. Per day, my agent achieved an average growth rate of 0.0263%, while the model in [6] achieved a per day average growth rate of 0.0189%. Thus, my model's per day growth rate is 39.2% better than what was achieved in [6].

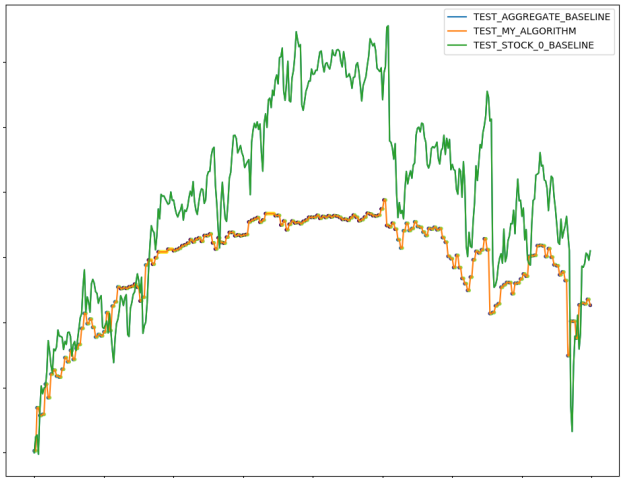
[6] does not provide risk statistics, but it rather states that the model is very volatile. The fact that for my agent, an overwhelming majority of trials resulted in profits despite the bearish market, and the fact that even in the instances where my agent made a loss, the actual loss was quite small, attests to the low risk of the policy learned by the agent. This is primarily due to the expanded set of actions that this model offers, i.e. actions through which the agent can buy/sell different proportions of stock rather than go all-in with no heed for risk.

[3] shows much better returns(daily average growth of more than 0.05, which is almost twice that of my agent) on a synthetic risky asset, but it is difficult to compare performance on synthetic assets against results on real-world data. It would be an interesting exercise to test my agent on synthetic assets that are not influenced by unknown macroeconomic factors that are completely different between the training data and test data.

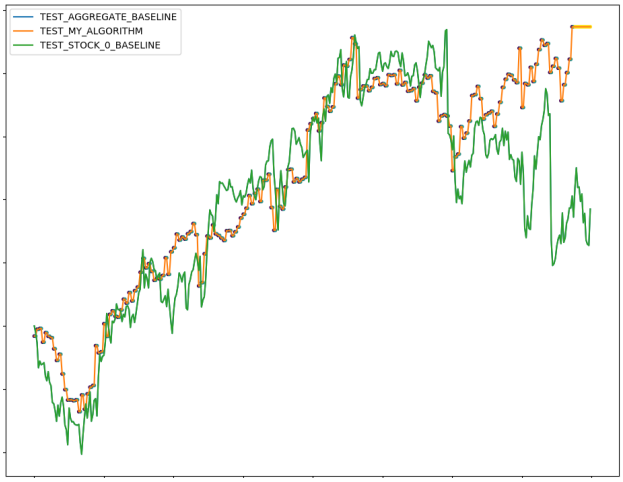
The plot in **Fig. 3** shows a histogram of where the profits lie, and it is evident that the model leans towards positive yields and clearly beats a simple waiting strategy.



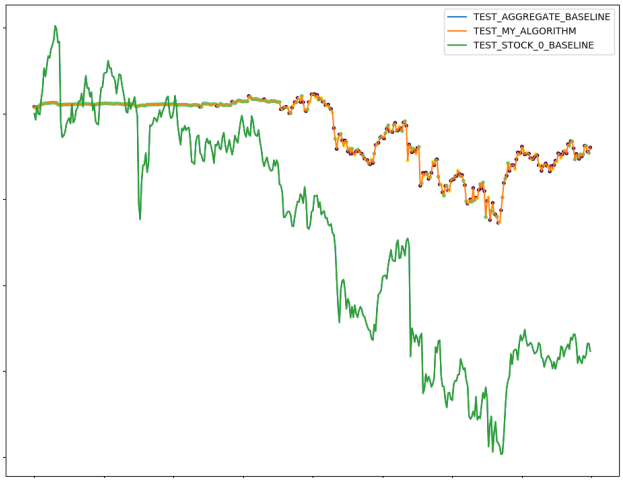
(a) Note how the agent waits until the price is very low, then makes a buy call



(b) An example of hedging where the agent sells off most of its stock to reduce risk. Here, it reduces the profit, but if the price had fallen, the loss would have been reduced.



(c) Good, but not extraordinary performance in a bull market. Note the quick exit before the dip at the end.



(d) The agent makes a poor investment choice. Although it is unable to make profits, the net loss is minimal

Fig. 2: Some performance plots for behavioral analysis. Green dots represent sell decisions, maroon dots are buy decisions and yellow shading is for periods of holding all assets.

I also observed that when the stock market was bearish(when the market is on a non-increasing or a very weakly increasing trend) the agent was able to make significant profits, thus demonstrating that the agent has learned to game the market somewhat well. See **Fig. 2a** for an example trial. In this, the agent realizes that the current price is too low for the particular stock, and chooses to buy, with an expectation that even a slight bounceback would result in enormous profits.

The expanded action space also allows the agent to mitigate some risk by hedging bets on both cash flow and stocks. In case the agent is unsure of whether a stock will rise, it can split its assets between cash and the stock, and thus convert its estimation of the probability of price increase to a risk-weighted profit. **Fig. 2b** shows an example of this hedging behavior.

However, the agent does have some weaknesses. While the agent is able to make extraordinary profits in bear markets, its performance is surprisingly not exceptional in case of bull markets(when the market itself is on a generally upward trend) - the agent invests an initial amount and is content to perform minor trades to make incremental profits rather than exploit peaks and valleys to the fullest extent possible. See **2c** for an example. At the end of this trial, the agent notes the sharp slide from the top and quickly sells all its stock as a ”stop-loss” measure.

In summary, some of the behaviors learned by the agent are

1) Buy and sell only when such actions are valid, i.e. don’t buy stocks when you cannot afford the capital.
2) If a stock is falling rapidly, expect a valley and wait till the valley to buy the stock.
3) If you already have a stock and its value starts decreasing rapidly, sell it to avoid loss(stop-loss)
4) Mitigate risk by hedging your assets between stocks and cash flow.
5) Recoup losses through incremental trades.

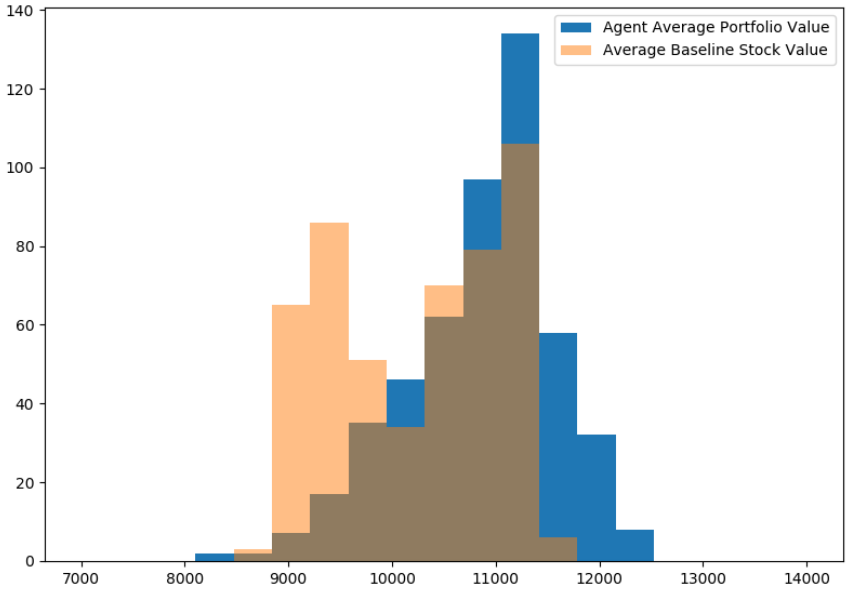


Fig. 3: Histograms of average portfolio value for the trained Agent(Blue) vs an Invest-and-Forget strategy

6) For bullish stocks, invest and forget(which is not the most optimal strategy, but does provide good returns).

XI. Potential Improvements

- The input features should be made as invariant of the actual stock as possible. This would allow us to try and form a generalized trading policy that is independent of the stock that we are trading on. Whether such a policy is possible is also an interesting question, but we will be able to make use of much larger datasets if this is indeed the case.
- Domain-knowledge must be encoded into the input variables. Factors like support levels, stock-specific data extracted from past behavior, etc. can be useful information for the neural network. Whether a network's performance improves by feeding in this data itself would be an interesting exercise.
- Test on synthetic assets to compare performance with [3].
- Evaluate Policy-Gradient based approaches on the same problem.

XII. Concepts/Tools Studied and Used

- Reinforcement Learning
- Q-Learning
- Policy Iteration(Not used, only studied)
- Catastrophic forgetting
- Double DQN
- Reservoir Sampling
- Python Profiler for identifying bottlenecks
- Oscillation in DQNs
- Markov Decision Process
- Learning with Continuous State Spaces
- Experience Replay
- Boltzmann Sampling
- ϵ-greedy Exploration
- Reinforcement Learning for game playing

XIII. Projects Consulted

- Reinforcement Learning for playing Doom ([9])
- Teaching Machine to Trade ([6])

XIV. Comments/Conclusion

The performance of the agent that I've developed in this project is better(more yield, less risk) than other similar attempts I've seen online. I first demonstrated that the agent is able to learn from a reward signal to buy and sell only when it is a valid action. Then, I used carefully crafted reward signals to enforce/eliminate particular behaviors and attempted to maximize the profit over time. The plots obtained do show that some good trading strategies are indeed learned by the agent. Note that the agent is simultaneously learning to predict the future movement, as well as the optimal decisions for that movement based upon the confidence of predictions.

While this has been an interesting academic exercise, it is obvious that the agent is in no shape for real-world trading. Cleverer input crafting methodologies may allow us to expand the size of the training data, thus allowing us to use the full power of deep learning without getting stuck in the pitfall of overfitting.

References

[1] Vincent François-Lavet, Raphael Fonteneau, and Damien Ernst. How to discount deep reinforcement learning: Towards new dynamic strategies. *arXiv preprint arXiv:1512.02011*, 2015.

[2] Dominik Gawlik. New York Stock Exchange Dataset. https://www.kaggle.com/dgawlik/nyse/home, 2016.

[3] Pierpaolo G. Necchi. Reinforcement Learning for Automated Trading. http://www1.mate.polimi.it/˜forma/Didattica/ProgettiPacs/BrambillaNecchi15-16/PACS_Report_Pierpaolo_Necchi.pdf, 2018.

[4] Cam Nugent. S & P Stock Data. https://www.kaggle.com/camnugent/sandp500/version/1/home, 2017.

[5] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy P Lillicrap, and Greg Wayne. Experience replay for continual learning. *arXiv preprint arXiv:1811.11682*, 2018.

[6] Shuai. Teaching Machine to Trade. https://shuaiw.github.io/2018/02/11/teach-machine-to-trade.html, 2018.

[7] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[8] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[9] Felix Yu. Implementation of Reinforcement Learning Algorithms in Keras tested on VizDoom. https://github.com/flyyufelix/VizDoom-Keras-RL, 2018.