Reinforcement learning for Stock Market Trading

Manan Tomar and Rohan Saphal

Indian Institute of Technology Madras

1 Introduction

The academic reinforcement learning [5] research community has largely stayed away from the financial markets. Possible reasons are, (1) the finance industry has a bad reputation, (2) the problem does not seem interesting from a research perspective, or (3) because data is difficult and expensive to obtain. In our project, we hope to explore training Reinforcement Learning agents to trade in the financial markets and how this can be an extremely interesting research problem.

Reinforcement Learning makes a right fit for applying to financial markets because :

- **End-to-End Optimization** : In the traditional strategy development approach there are several steps before we get to the metric we actually care about. For example, to find a strategy with a maximum drawdown, it is needed to train a supervised model, come up with a rule-based policy using the model, backtest the policy and optimize its hyper-parameters, and finally assess its performance through simulation. Reinforcement Learning allows for end-to-end optimization and maximizes (potentially delayed) rewards.

- **Learned Policies** : Instead of needing to hand-code a rule-based policy, Reinforcement Learning directly learns a policy. Theres no need to specify rules and thresholds such as buy when you are more than 75 percent sure that the market will move up. Thats inherent in the RL policy, which optimizes for the metric. Also because the policy can be parameterized by a complex model, such as a deep neural network, it is possible to learn policies that are more complex and powerful than any rules a human trader could possibly come up with.

- **Learning in a Simulator** Since Reinforcement Learning agents are generally trained in a simulation, and the simulation can be as complex as desired, taking into account latencies, liquidity and fees, we always have the option to not test the policy in the real market until some performance level is maintained while testing in the simulator. The simulator can in turn be tweaked to match the market behavior more precisely.

2 Background and Preliminaries

Previous works[2, 1, 4] do not use a simulator but use a dataset to showcase their results and hence fail to capture the amrket dynamics. For our experiments, we use a recently released open source trading platform called https://gitlab.com/doctorj/sairen/. This simulator is compatible with OpenAI gym like environments and thus makes it a universal choice, allowing us to cover most of the prominent algorithms. Sairen provides access to real world live financial data to test machine learning algorithms on. It is based on intraday trading, done at the frequency of minutes or seconds instead of nanoseconds. The observations are the current market

data, actions are whether to buy and sell, and the rewards are the profit or loss incurred. We futher detail the MDP formulation below :

Agent : The agent in this context is the trading agent. The agent can be compared to a human trader who opens the GUI of an exchange and makes trading decision based on the current state of the exchange on his or her account.

Environment : The exchange is our environment. The important thing to note is that there are many other agents, both human and algorithmic market players, trading on the same exchange. Let us assume that actions are being taken on a minute scale (more on that below). Some action is taken, we wait for a minute, get a new state, take another action, and so on. When a new state is observed, it will be the response of the market environment, which includes the response of the other agents as well. Thus, from the perspective of our agent, these other agents are also part of the environment. They are not under our control. However, by putting other agents together into some big complex environment the ability to explicitly model them is lost. For example, one can imagine an agent to learn to reverse-engineer the algorithms and strategies that other traders are running and then learn to exploit them. Doing so would put us into a Multi-Agent Reinforcement Learning (MARL) problem setting, which is an active research area. For simplicity, it remains best to assume that our agent is interacting with a single complex environment that includes the behavior of all other agents.

2.1 MDP formulation

Reinforcement Learning problem can be formulated as a Markov Decision Process (MDP). We have an agent acting in an environment. Each time step the agent receives the current state as input, takes an action, and receives a reward and observes the next state. The agent chooses the action based on some policy. The goal is to find a policy that maximizes the cumulative reward over some finite or infinite time horizon.

State: The observation that has been used is a vector consisting of the following quantities; time, bid, bid-size, ask, asksize, last, last size, last time, open, high, low, close, volume weighted average price, volume, position, unrealized gain.

- Bid is the value of the stock at which you can buy the stock at that particular time.
- Bidsize is the volume of the stock that is being bought at the bid price
- Ask is the value of the stock at which you can sell the stock at that particular time
- Asksize is the volume of stock that is being sold at the ask price
- Last, last size and last time shows the price, volume and time of the last transaction that took place in the market
- Open is the price at which the stock opened that day
- High is the highest price reached by the stock till that time of the day
- Low is the lowest price reached by the stock till that time of the day
- Close is the price at which the stock closed the previous day
- Volume weighted average price is the stock price multiplied by number of shares traded and then dividing by the total shares traded for till that time of the day
- Volume is the total stock traded till that time of day

- Position is the current status of our agent in terms of the number of stocks currently in our account
- Unrealized gain is the profit or loss that would result if the agent had sold the stock at that moment. This is a hypothetical value.

Action: We use a discrete action of size 11. We set a maximum quantity of stock that can be bought or sold. Actions 0 to 4 lead to selling the stock with 0 selling the maximum quantity of stock and and 4 selling 20 percent of the maximum quantity. Similarly actions 6 to 10 lead to buying the stock with 10 buying the maximum quantity and 6 buying 20 percent of the maximum quantity. Action 5 leads to the agent holding its current position with no buying or selling. We did not use a continuous action space because all the continuous values might not result in an actual quantity of stock and would be approximated to the nearest integer making it difficult for the agent to distinguish between two actions that result in the same volume of stock.
Reward: The reward that we have used is the profit or loss from the transaction. Although there are other metrics that can be used to increase risk awareness in the agent, for simplicity we have used this reward function. One can image a wide range of complex reward function that trade-off between profit and risk.
The **discount factor** we have used is of value 0.99

3 Methodology

To analyze how Reinforcement Learning can aid in stock market control, we choose to evaluate the performance of the Deep Q-Learning algorithms such as DQN and Deep SARSA as it allows for having discrete actions and neural network estimates for value functions which improves interpretability and tackles large continuous state spaces respectively. Below we provide a formal introduction to the algorithmic details.

Reinforcement Learning (RL), [5], considers the interaction of an agent with a given environment and is modeled by a Markov Decision Process (MDP), defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \rho_\prime, r \rangle$, where $\mathcal{S}$ defines the set of states, $\mathcal{A}$ the set of actions, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ the transition function, $\rho_\prime$ the probability distribution over initial states, and r: $\mathcal{S} \times \mathcal{A} \to \mathcal{R}$ the reward function. A policy is denoted by $\pi(s)$: $\mathcal{S} \to \mathcal{P}(\mathcal{A})$, where $\mathcal{P}(\mathcal{A})$ defines a probability distribution over actions $a \, \epsilon \, \mathcal{A}$ in a state $s \, \epsilon \, \mathcal{S}$. The objective is to learn a policy such that the return $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} \, r(s_i, a_i)$ is maximized, where $r(s_i, a_i)$ is the reward function and γ is the discount factor.

3.1 Deep Q Learning

Q-learning [6] attempts to estimate the optimal action-value function $Q^*(s, a)$. It exploits the Bellman optimality equation, the repeated application of which leads to convergence to $Q^*(s, a)$. The optimal value function can be used to behave optimally by selecting action a in every state such that $a \in argmax_{a'} Q(s, a')$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right] \tag{1}$$

[3] introduce Deep Q-learning, that extends Q-learning to high dimensional spaces by using a neural network to model $Q_\theta(s, a)$. The authors use the TD-error to back-propagate the gradients which are used to learn the parameters θ. A delayed target-network and an experience replay [3] are typically utilized to stabilize the training and reduce the correlation between samples in a mini-batch.

3.2 Deep SARSA

The SARSA algorithm differs from Q-learning in that it uses the action given by the policy instead of the greedy action for updating the Q-values. This lands SARSA in an on policy scheme. The update is given as

follows.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{2}$$

where a_{t+1} is given by the policy at state s_{t+1}. [5] introduce a deep version of SARSA wherein the Q value functions are modeled using neural networks, parameterized by θ, $Q_\theta(s, a)$. The data from each episode is collected and processed in batches to allow the above update.

3.3 Exploration Strategies

In order for an agent to learn how to deal optimally with all possible states in an environment, it must be exposed to as many of those states as possible, and this necessitates the need to explore. From this problem has emerged an entire subfield within reinforcement learning that has attempted to develop techniques for meaningfully balancing the exploration and exploitation tradeoff. Ideally, such an approach should encourage exploring an environment until the point that it has learned enough about it to make informed decisions about optimal actions. We use the following exploration strategies to understand which would perform better in the trading environment.

- **Greedy Policy**: In this setting, the agent takes the action with the maximum Q value at any state in the environment. This is a naive approach to ensuring the agent takes the optimal action at each step. The obvious shortcoming of this method is that the agent will never see the effect of any other action other than the optimal one. This always leads to a sub-optimal solution.

- **ϵ-Greedy Policy**: This is the most widely used exploration technique and is a combination of taking random actions with probability ϵ and greedy actions otherwise. This way the agent might take actions that provide new information by moving to an unexplored part of the state space. The ϵ parameter is adjustable and is generally chosen as a large value at the beginning and slowly decreased to a small value. Despite the prevalence of usage, this method is far from optimal, since it takes into account only whether actions are most rewarding or not.

- **Boltzmann Policy**[7]: Instead of always taking the optimal action, or taking a random action, this approach involves choosing an action with weighted probabilities. This is accomplished by using a softmax over all the Q values at any state. This would generally result in the agent choosing the action that is most likely to be optimal.

  $$\pi(a|s) = \frac{\exp(Q(s, a)/\tau)}{\sum_{i=1}^{n} \exp(Q(s, i)/\tau)} \tag{3}$$

  A temperature coefficient is also used while calculating the probabilities of each action which can be controlled. The main advantage over ϵ-greedy policy is that this method assigns weights to other actions and looks at the corresponding likelihoods. This way the agent can ignore actions which it estimates to be largely sub-optimal and give more attention to potentially promising, but not necessarily ideal actions. One shortcoming is that what the agent is estimating is a measure of how optimal the agent thinks the action is, not how certain it is about that optimality.

- **Max Boltzmann Policy**: This framework is a combination of ϵ-Greedy and Boltzmann policies and aims to combine the advantages of both methods. Instead of taking a random action with probability ϵ, the agent takes the greedy action at this stage and takes the most likelihood action otherwise from the relative probabilities between the actions. This framework makes sure that the agent does not take an action that is too far from the optimal action yet chooses to explore with relatively sub-optimal actions.

There are more hyper-parameters involved with this method, ϵ and the temperature coefficient, making it more cumbersome.

-

4 Experiments and Analysis

The experiments were run primarily with DQN and Deep SARSA with the different exploration strategies to understand which strategy would be most profitable.

The problem is modelled as a finite horizon problem with each episode consisting of 60 steps. The agent is run for 300 episodes for each exploration strategy. At the end of each episode, all the remaining stocks are automatically sold off. This is to keep in track with how the real stock market works. Buying or selling is done at the market price so that it would get executed immediately. However, due to latency, orders are not executed as the market price changes and sometimes orders are partially executed based on what the demand/supply is.

Tabular Q learning was formulated and tested, however it fails to learn a good policy because the number of different states are very large. Use of a smaller state space makes it difficult for the agent to form an optimal policy as the data is insufficient for it to make optimal decisions. This led to approximating the Q value using function approximation. Data is obtained at every second and therefore the number of samples scales linearly with time. We therefore use deep neural network as the function approximator because of the large sample of data that is needed to be processed.

4.1 Model Implementation details

The model consists of four layers and each layer begins with a batch normalization layer. The batch normalization layer scales the data between 0 and 1. The first three layers consists of 16 neurons each and is followed by a ReLU activation function. The last layer consists of 11 neurons and is equivalent to the size of the action space. This is followed by a linear activation function. The learning rate is choosen and fixed at $1e^{-}3$ and the Adam optimizer is used for both DQN and Deep SARSA. The smoothing coefficient used for DQN target network is $1e^{-}2$. The replay memory has a size limit of 57600 which is equivalent to 16 hours of real time training and the window size is chosen as 1. Both the models have a warm start of 600 steps.

4.2 Deep Q learning

4.2.1 Greedy Policy

The policy learned in this strategy results in the agent buying continuously during the course of the episode and selling at the end. The agent does not observe profit during the entire training period and has possibly learned that just continuing to buy stocks at every step would not result in loss during the course of episode and the automatic sell-off could result in profit.
Towards the end of the training schedule, the PNL is zero due to partial or non-execution of orders. If looked at carefully, the agent would have profited towards the end of the 300 steps, had the orders been executed completely. Between steps 240 and 300, the agent bought stocks continually at low prices, hoping that the automatic sell-off could possibly result in profit.

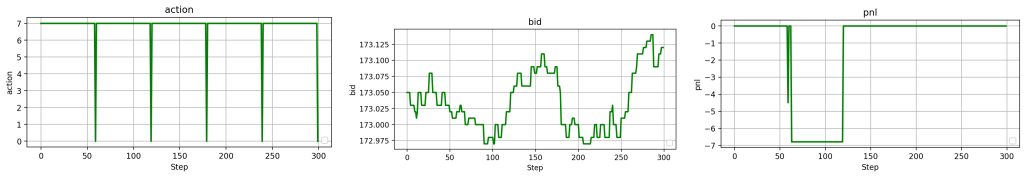

Figure 1: Performance of Greedy policy for last 300 steps of training

4.2.2 Boltzmann Policy

The policy learned in this strategy results in the agent buying and selling vigorously. The agent is trying to gain as much profit as possible from doing so, but in vain. It is interesting to see that the frequency of buying and selling is higher at either ends of the graph when the stock price is lower. It can therefore be concluded that this strategy is extremely volatile.

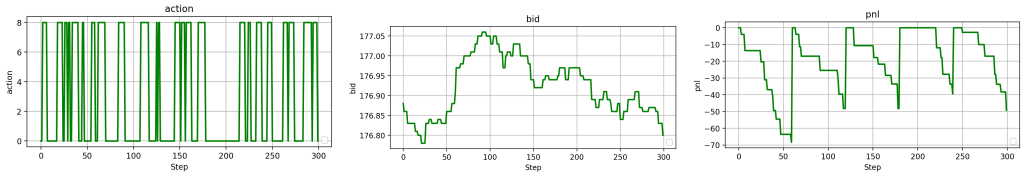

Figure 2: Performance of Boltzmann Policy for last 300 steps of training

4.2.3 Epsilon Greedy Policy

The policy learned in this strategy seems close to the ideal policy. It is observed that the most frequent action taken by the agent is action 3, which results in selling of stock. An important point to note is that a sell action made when there are no stocks to sell would result in no change in portfolio and hence is a pseudo 'hold' action. It is also reasonable to assume that since there is only one action among the eleven that aims to 'hold' or do nothing, the probability of it being seen is very less. The agent over the course of time has however learnt that after selling the stock or buying it, it is appropriate to hold it for some time before buying or selling it respectively. This policy seems similar with the way humans would act. It is also interesting to note that the agent buys the stock at appropriate intervals when the price is lower and sells at higher prices.

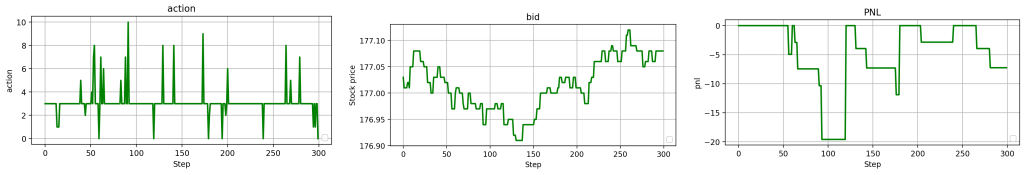

Figure 3: Performance of Epsilon Greedy Policy for last 300 steps of training

4.2.4 Max Boltzmann Policy

The policy learned in this strategy looks similar to the Boltzmann policy except that the policy is biased toward buying more stocks on average. It can be seen that the policy seems to combine the ideas behind the epsilon greedy and Boltzmann policies. The agent is constantly buying stocks hoping that when automatic

sell-off happens, it would result in profit. It also seems to show signs of buying and holding followed by selling and vice versa, a principle followed in the Boltzmann policy. It is also interesting to note that the loss the agent has received is obvious, as the price of the stock was constantly plummeting and most strategies would fail since any stock bought at any point would be sold with a high chance at a loss.

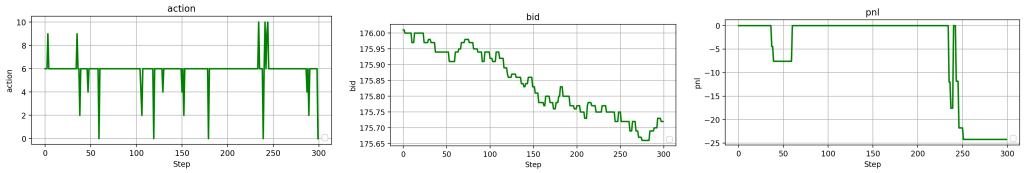

Figure 4: Performance of Max Boltzmann policy for last 300 steps of training

4.3 Deep SARSA

4.3.1 Greedy Policy

The policy learned in this setting is very sub-optimal and seems to have converged to a local optima. The agent has figured out that by just continuously selling, it will not realize loss. The optimal strategy would be to buy stocks as the price is falling.

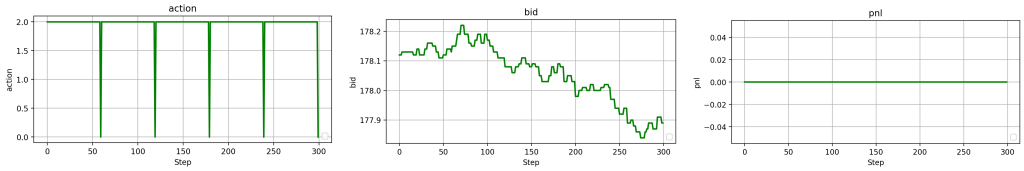

Figure 5: Performance of Greedy Policy for last 300 steps of training

4.3.2 Boltzmann Policy

The policy learned in this setting can be seen to be much more volatile than in the DQN variant. The loss is therefore also steeper and more frequent.

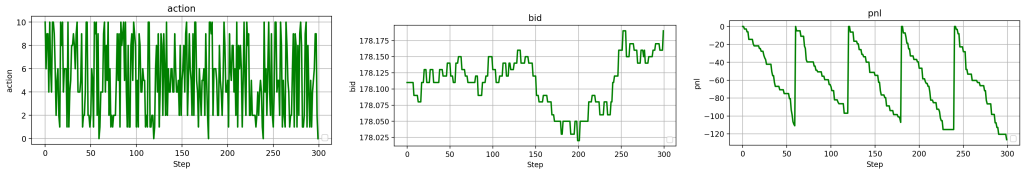

Figure 6: Performance of Boltzmann policy for last 300 steps of training

4.3.3 Epsilon Greedy Policy

The policy learnt in this setting is slightly more volatile than the DQN variant. Similar to previous cases, the volume of stock bought or sold has resulted in the net loss during trading.

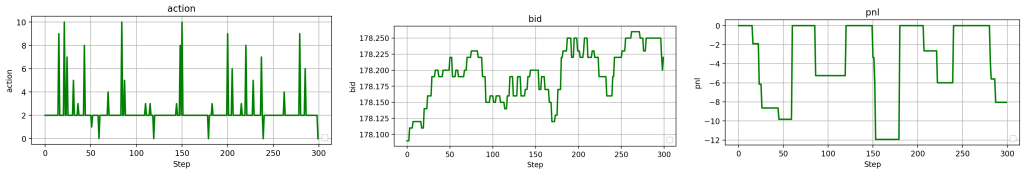

Figure 7: Performance of Epsilon Greedy Policy for last 300 steps of training

4.3.4 Max Boltzmann Policy

The policy learned in this setting can be seen to resemble to some extent the policy in case of DQN. It is disappointing to see that the net profit or loss variable is in the negative, but it is interesting to note that the policy makes sense with the nature of the market. The bid value is continuously increasing and it would ideally be in the interest of the agent to buy low and sell high. A possible reason for the net loss could essentially be the quantities in which the buy and sell is occurring.

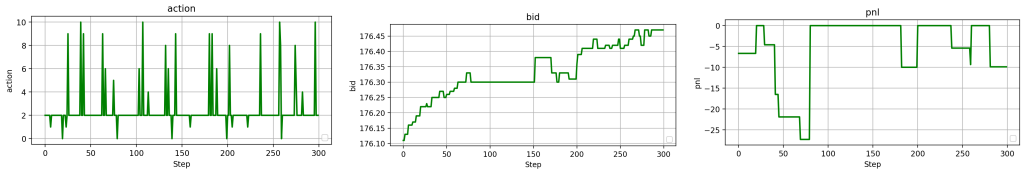

Figure 8: Performance of Max Boltzmann policy for last 300 steps of training

5 Conclusion and Future work

Comparison of different algorithms and exploration strategies has resulted in a better understanding of what works and what does not work in the trading setting using reinforcement learning. From our experiments, we can conclude that Epsilon greedy and Max Boltzmann policy with the DQN algorithm seem to be the better performing strategies. Better performance on DQN is expected as it has been shown to work well in complex partially observable environments by improving stability []. From the above two policies, the better one empirically is Max Boltzmann as its policy is much more interpretable and randomness is much lower. This is of particular interest in stock trading domains where black box models are less preferred.

In the future work, we hope to use recurrent neural networks to approximate the Q values. From experimental results, it can be concluded that the temporal nature of the environment needs to be captured due to the partial observability of the environment, presence of multiple opponent agents and effect of latencies in decision making. We also hope to conduct an ablation study on the hyper-parameters of Epsilon greedy and Max Boltzmann strategies. Studying the effect of the different discrete action space variants is also an interesting avenue to explore.

References

[1] Yan Chen, Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. Trading rules on stock markets using genetic network programming with sarsa learning. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1503–1503. ACM, 2007.

[2] O Jangmin, Jongwoo Lee, Jae Won Lee, and Byoung-Tak Zhang. Adaptive stock trading with dynamic asset allocation using reinforcement learning. *Information Sciences*, 176(15):2121–2147, 2006.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[4] John Moody, Lizhong Wu, Yuansong Liao, and Matthew Saffell. Performance functions and reinforcement learning for trading systems and portfolios. *Journal of Forecasting*, 17(5-6):441–470, 1998.

[5] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

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

[7] Marco A Wiering. *Explorations in efficient reinforcement learning*. PhD thesis, University of Amsterdam, 1999.