Reinforcement Learning on Stock Market Prediction

Zhiqi Ma, Minghao Cai, George Xu
Department of Mathematics, University of California, Los Angeles
{zhiqima,mikecai020418,zijian328}@g.ucla.edu

March 15, 2024

Abstract

This paper explores the application of Reinforcement Learning (RL), specifically Deep Q-Learning (DQN) and Proximal Policy Optimization (PPO), in predicting stock market to maximize trading profits. We take a deep-dive into RL learning background, related theories, as well as relevant methodology and algorithms. We then set up different DQN and PPO models, incorporating features like transaction costs and initial account balances to simulate a more realistic trading environment. Tests and evaluations have been applied to three stocks with different volatility, including Apple Inc., Procter & Gamble Co., and Tesla Inc., after which we've compared the results through graphical visualizations, providing both a qualitative and quantitative analysis. We've discussed briefly the contribution, limitation, and possible future work for RL on stock trading in the end.

1 Introduction

1.1 Quantitative Stock Trading

Stock trading involves buying and selling shares of publicly traded companies on stock exchanges. Investors participate in stock trading to capitalize on price fluctuations and generate profits. They employ various strategies to predict stock trends, but this is challenging due to the volatility of stock prices. Factors such as market supply and demand, company performance, and other complex economic indicators can cause prices to change. As a result of today's more interconnected, global economic environment, the dynamics of stock prices have become even more difficult to catch, and it may seem impossible to correctly predict the stock prices using past data. However, researchers have found that the majority of stocks' true value lies within their past stock price movements, making the collection and analysis of past stock price movements essential for predicting future stock prices [1].

In fact, the reality does align with this finding. With the advent of today's AI age, quantitative trading (QT) – a strategy that relies on mathematical models as well as machine learning (ML) algorithms to automate numerous investment tasks – plays an increasingly pivotal role in the financial market. For example, QT primarily involves tasks such as portfolio management, order execution, and algorithmic trading. Specifically, ML is highly effective for stock price prediction, which is a foundational aspect of all these three QT tasks, due to its ability to handle non-linearity in stock price evolution.

Moreover, deep learning (DL), which employs a multi-level neural network structure to extract key information from non-linear and random time series, works even better.

1.2 Reinforcement Learning

Reinforcement Learning (RL) is one emerging sub-branch of machine learning (along with supervised and unsupervised learning). Unlike the other two types of ML, RL assumes a more complex problem structure, providing a mathematical formulation of learning-based control for trial-and-error knowledge extraction. Instead of starting with a labeled or unlabeled dataset, an RL agent is trained to maximize a reward scheme by deciding which actions to take based on observations from the environment. The agent employs a policy to decide how a given state is associated with an action, and receives a reward and the next state from the environment. Such a policy is often chosen to be Epsilon-Greedy, which will be further discussed in later sections. The agent trains by updating its mapping of state-action pairs. In theory, the best action to choose is the one that maximizes future reward for all possible future states, nevertheless in practice, we only consider the best possible action in only the next future state, using the Bellman equation to perform updates.

1.3 Motivation – RL on Stock Price Prediction

In this project, we will implement two types of RL models – the Deep Q-Learning (**DQN**) model and the Proximal Policy Optimization (**PPO**) model. These two models will be set up, discussed, and analyzed separately on the task of stock market predictions to maximize profits. Profitability of the two models will also be presented and compared.

In detail, we must translate "perform stock trading to make profits as much as possible" by rewarding the agent using its current and previous memory of states and actions. The general mechanism behind the agent involves using historical stock data, such as daily closing stock prices from the past several years, to train it. The agent will then choose actions at each step so that it can sell/buy certain amount of stocks or do nothing, which are intended to simulate the actual stock trading environment, and the agent's rewards will be determined based on stock performance. Training process ends when the agent iterates over all training data, after which the model can be applied for evaluation.

For the DQN model, we followed this DQN skeleton code. Based on it, we've made various modifications and improvements. Firstly, we've added features using Yahoo Finance Python Library so that users can have access to any stock they choose for model training. Secondly, the original code is hard to read (with nearly no comments) and seems to contain issues with unsuccessful execution. We've fixed the issue and make the program easier for replication. Thirdly, we've incorporated elements such as transaction cost and initial balance to make the model more realistic, and visualization techniques are used to present the results in a more straightforward and clear manner.

Similarly, the PPO model was guided by Proximal Policy Optimization [2], while following Santhosh's PPO trading environment as the basis setup. Based on Santhosh's framework, we've made various modifications that included: optimizing parameters and the model to enable it to take continuous inputs such that it reflects the confidence of each transaction decisions, addressing coding errors such as limiting the probability function so the agent will never oversell, improving the environment so that the model can take various technical parameters, and finally providing visualizations and testing scenarios.

A rough outline of the paper is as follows. Section 2 will provide more background on RL, Deep Q-Learning, and PPO, diving into the history and development of these models. Afterwards we'll illustrate the theory behind DQN, PPO, and the motivation behind using a Deep Neural Network (DNN) to train the RL agent. Section 3 will describe the methodology we used about DQN methods and its various schemes, as well as the PPO method. Section 4 contains major results, visualizations, and figures with analysis. Later, a discussion on the contribution, limitation, and possible further research directions are summarized in Section 5. Finally, we end by giving a conclusion in Section 6.

2 Background and Related Theory

2.1 RL Background

The technique of reinforcement learning is concerned with the problem of finding suitable actions to take in a given situation in order to maximize a reward, which utilizes the Markov Decision Process (MDP) [4]. A basic RL agent interacts with its environment in discrete time steps. At each time t, the agent receives the current state S_t and reward R_t. It then chooses an action A_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state S_{t+1} and the reward R_{t+1} associated with the transition (S_t, A_t, S_{t+1}) is determined. The goal of a reinforcement learning agent is to learn a policy: $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1], \pi(s, a) = \Pr(A_t = a \mid S_t = s)$ that maximizes the expected cumulative reward.

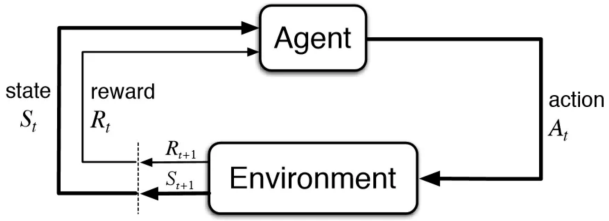


Figure 1: Reinforcement learning process

Moreover, another general feature of reinforcement learning is the trade-off between exploration, in which the system tries out new kinds of actions to see how effective they are, and exploitation, in which the system makes use of actions that are known to yield a high reward. Too strong a focus on either exploration or exploitation will yield poor results.

2.2 Deep Q-Learning

2.2.1 What is Q-learning?

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) MDP. It works by learning an action-value function, often denoted by $Q_\pi(s, a)$, which ultimately gives the expected utility of taking a given action a in a given state s. A policy, often denoted by π, is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state.

Within this process, there is a fundamental concept called Bellman Optimality Equation (BOE). It provides a way to calculate the optimal policy—i.e., the best action to take in each state to maximize long-term rewards. The BOE is given by:

$$Q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \tag{1}$$

where $Q^*(s,a)$ is the optimal Q-value for being in state s and taking action a; R_{t+1} is the reward received after taking action a in state s; γ is the discount factor, which represents the difference in importance between future rewards and immediate rewards; and $\max_{a'} Q^*(S_{t+1}, a')$ is the maximum Q-value for the next state S_{t+1}, across all possible actions a'.

2.2.2 What is Deep Q-Network?

Neural networks are exceptionally good at coming up with good features for highly structured data. We can represent our Q-function with a neural network, that takes the state and action(s) as input and outputs the corresponding Q-value. This approach has the advantage, that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately. DQN utilizes the BOE to learn the optimal Q-values, where the Q-values are approximated by a neural network (the Q-network), and the equation guides the network's training. The key idea is to iteratively update the Q-values towards the expected returns following the optimal policy.

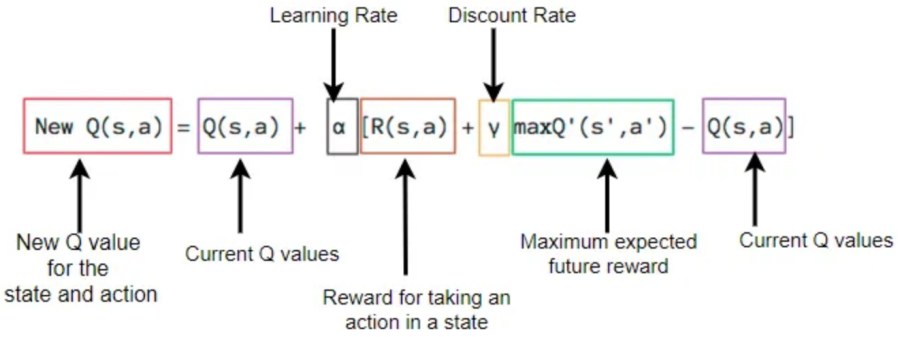


Figure 2: Q-Learning iteration

2.2.3 Epsilon-greedy strategy

The DQN employs the epsilon-greedy (ϵ-greedy) strategy to balance exploration and exploitation mentioned above. This strategy involves choosing actions randomly with a probability of ϵ to encourage exploration of the environment, and selecting the best known action based on the Q-values with a probability of $1 - \epsilon$ to ensure exploitation of the agent's accumulated knowledge. Initially, ϵ is set high to favor exploration, and it is gradually decayed over time, shifting the balance towards exploitation as the agent learns more about the environment. This mechanism allows the DQN to discover a broad range of state-action pairs and refine its strategy towards optimal decision-making, ensuring both comprehensive environmental understanding and efficient reward maximization.

2.2.4 Experience Replay

Experience Replay is a fundamental technique in DQN that improves learning stability and efficiency by storing an agent's past experiences (tuples of state, action, reward, and next state) in a replay buffer and later randomly sampling mini-batches of these experiences for model training. Such random sampling helps mitigate the issues of correlated data and non-stationary distributions, common in online learning directly from consecutive experiences. By using mini-batches, the network benefits from learning across a more diverse set of experiences in each update, enhancing generalization and reducing the risk of overfitting to recent sequences of observations. This approach not only allows the DQN to reuse past experiences, maximizing the informational value of each encounter, but also smoothens the learning process by averaging the updates over multiple past experiences, leading to more stable and effective policy learning.

2.3 Proximal Policy Optimization

2.3.1 What is Proximal Policy Optimization?

Proximal Policy Optimization (PPO) is a Policy Gradient Algorithm, a direct enhancement from Trust Region Policy Optimization (TRPO), also introduced by OpenAI [3]. Before discussing the features of PPO, it is necessary to analyze the Policy Gradient Algorithm. As shown below, the policy gradient algorithm optimizes the parameters of a policy to maximize the expected reward by computing gradients of a performance objective and performing gradient ascent.

0:	Input: $\theta \leftarrow \mathbb{R}^{\lvert\theta\rvert}$	▶ Initialize θ
1:	**for** $n = 1$ **to** N **do**:	
2:	$\tau \sim \pi_\theta$	▶ Generate state-action trajectory with π_θ
3:	**for** $t = 1$ **to** T **do**	
4:	$R(\tau \mid t) = R_t + R_{t+1} + \ldots + R_T$	▶ Compute cum. reward
5:	$\theta \leftarrow \theta + \alpha R(\tau \mid t) \nabla_\theta \log(\pi_\theta(a \mid s)$	▶ Update θ

Figure 3: Policy Gradient Algorithm

In the update method, the model uses a fixed learning rate α to update the policy parameters. Fixed learning rates need to determine a specific learning rate, with includes a risk of overshooting. Consequent models such as the Natural Policy Gradients Algorithm [5] and TRPO Algorithm all took different measures to either set up a dynamic learning rate or to limit the degree of the update with a cap. PPO, which is a direct update from TPRO, approached this challenge with both the above methods, while significantly improving its update speed and performance. It is also possible to add an improvement check, which is able to check if the reward is improved comparing to the previous epoch.

The PPO Algorithm, referring to the Clipped Surrogate Objective variant [2], utilizes a clip, which is a hard cap for the degree of policy update, inside a small interval ϵ. Then the final objective function chooses the lower between the original objective function and the clipped objective function. It is worth noting this pessimistic approach will in turn result in worse action, but will furthermore determine a conservative policy update. This

action is shown by the below formula [2]:

$$L^{CLIP}_{\pi_{\theta_0}}(\theta_k) = \mathbb{E}_{T\sim\pi_{\theta_0}}\left[\sum_{t=0}^{T}\min\left(\rho_t(\pi_{\theta_0},\pi_{\theta_k})A_t^{\pi_{\theta_k}}, \text{clip}\left(\rho_t(\pi_{\theta_0},\pi_{\theta_k}), 1-\epsilon, 1+\epsilon\right)A_t^{\pi_{\theta_k}}\right)\right] \tag{2}$$

here A is the advantage function:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \tag{3}$$

, among which $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$; and $\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ is the importance sampling ratio.

The model we are going to set up follows the outline discussed in the PPO paper [2], where the algorithm update as following:

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 4: PPO, Actor-Critic Style

PPO, comparing with Q-learning, should be more reliable as it provides a trust constraint mentioned earlier, which makes it less prone to instability and overshooting.

3 Methodology

3.1 DQN Methodology

3.1.1 General Model Structure

To begin with this section, let's present the general idea and structure of our DQN models. Utilizing the yahoofinance package, we first write a function to create a data frame including stock price data of AAPL for the past 5 years(1259 rows of data), including open, close, low, and high prices. The closing prices of the stock will form the data on which we train and test our model.

Next, we set up the DQN model, which consists of four main parts: Initialization, Model, Act, and expReplay.

The 'Agent' is initialized with essential parameters like state_size (length of historical stock price sequence), is_eval (evaluation flag), and model_name. Constants such as γ (discount factor), ϵ (exploration rate), and others are defined to manage the Q-value update and the trade-off between exploration and exploitation in decision-making.

Then, a simple neural network model is created using TensorFlow Keras in the model method. It comprises an input layer sized to the state_size, two hidden layers with 16 and 8 units respectively, utilizing the ReLU activation function, and an output layer employing a linear activation function to match the action_size. The model is compiled using Mean Squared Error (MSE) loss and the Adam optimizer. If in evaluation mode (is_eval=True), the model is loaded from a file; otherwise, a new model is created. The structure of the code is listed below:

Parameters	**Agent with actions = 3**
Input Layer	state_size
Hidden layer 1	16 (ReLU)
Hidden layer 2	8 (ReLU)
Output Layer	action_size (Linear)
Implemented using	TensorFlow-keras

Table 1: Neural Network Architecture

The 'Act' method is responsible for the agent's action selection based on the current state. It implements an ϵ-greedy strategy mentioned in Section 2.2.3, allowing the agent to explore the environment randomly with probability ϵ and exploit its current knowledge with probability 1-ϵ. In evaluation mode, the agent exclusively exploits its learned policy.

The 'expReplay' method is an implementation of the experience replay mechanism discussed in Section 2.2.4. It samples a batch of experiences from the agent's memory and updates the model's weights based on the Bellman equation, estimating the optimal action-value function (Q_value). The targets for Q_value update are calculated based on the rewards and the maximum Q_value of the next state given in Section 2.2.1.

After Setting up the agent, we enter into the training Loop. We initialize the agent, stock price data, window_size(the length of historical sequence of stock prices that the agent uses as its input state, allowing the agent to consider patterns over a specified number of previous time steps when making decisions), and episode_count, then iterate through episodes, each representing an entire sequence of trading decisions. The agent interacts with the environment, making decisions to buy, sell, or hold based on its Q-learning policy. The outcomes of each action, including profits and actions taken, are recorded, and experiences are stored in the agent's memory for later learning. The model is periodically saved for later analysis. Note that the model will be trained on the first 1000 rows of data, and will be evaluated on the remaining 259 rows of data.

Lastly, for evaluation, a pre-trained model is loaded. In this phase, the agent engages with the environment under the learned policy without further updating the model, simulating decision-making on unseen market data. All actions and their resultant financial impacts are stored for further analysis.

3.1.2 Different Versions of DQN Model

Based on the general structure above, we've built 3 versions of DQN model, each of which is an improvement of the previous version and addresses part of its limitations.

Model version1 almost exactly follows the algorithm structure above, but it presents two serious issues: the first one is its slow execution speed (it takes about 30 minutes to run only 20 episodes for model training on VS Code); and the second one is that the

model does not include initial account balance and the transaction costs of the stock trading, which makes it harder to evaluate the profit rate generated by the model. Such model setting is unrealistic. Subsequently, Model version2 deals with the first issue, and Model version3 solves both issues. Note that for all 3 versions of our DQN model, profit is calculated by storing the bought price of the stocks when we buy the stock and then using (selling price - bought price) to get profit for the sell.

Model version2 has faster execution speed, and it takes about 6-8 minutes to train 1000 episodes on the same data. We changed the model significantly, which uses the PyTorch package to built the networks, with two hidden linear layers of 64 units each, implemented using RELU activation. PyTorch often has a speed advantage due to its dynamic computation graph and might lead to faster training. Moreover, the model utilized a fixed Q update and a soft update mechanism, which might also contribute to the increase in speed. However, the model still doesn't solve the issue of lacking an initial balance and consideration in transaction costs.

Model version3 is the final version of our model, except from increased speed compared with model version1, we've also incorporated initial account balance of $10,000 and transaction costs of 0.01% per buying and selling action to our model to make it more realistic to the real-world stock trading. Our model is also changed so that it only has 1 dense layer with 256 units, using RELU activation, and we set the window size to be 30 (the agent can have access to more past information during training). This time, the reward is set to profit rate of (current_money - initial_money) / initial_money, and during evaluation we are able to achieve around 0.5%-4% final profit rate after training our model.

3.2 PPO Methodology

3.2.1 General Model Structure

The basic structure is segmented into train/test environment, agent action and learning, memory storage, and loss and advantage calculation.

3.2.2 Environment Setup

The 'StockTradingEnv' class, derived from OpenAI's 'gym.Env', utilizes a custom stock trading environment. This environment incorporates a comprehensive state space representing trading indicators, such as opening, closing, high, and low prices, normalized within a specific range. The action space is defined to enable buying, selling, or holding stock percentages, fostering decision-making based on the current financial situation and market data. Similar to the 'StockTradingEnv' class, the 'TestEnv' class provides an environment for testing purposes, whereas both have similar code with differences in the harshness of the reward. In the training environment ('StockTradingEnv' class), the model utilizes higher transaction costs in the '_take_action' function and steeper rewards to encourage conservative behavior in the step function, where these two changes will increase total performance with a risk of over-fitting the training model if not adjusted properly. The testing environment provides realistic transaction behavior that aligns with the DQN model.

Another crucial function within 'StockTradingEnv' establishes the relationship between the transaction's confidence level and the transaction amount, as demonstrated in

the '_take_action' function. In the function, the amount parameter controls the percentage of available stocks or funds that are used in one action. In the first attempts, an effect is made to relate the amount value with the confidence of the transaction, which is depicted by the discrete probability distribution for the action distribution. Despite intuitively, a more realistically aligned relation for the confidence and the transaction amount should improve the result, it is not backed by training results, hence a fixed percentage of the transaction amount is used in the final model.

3.2.3 Agent Configuration and Learning Process

This part is designed in the 'Agent' class and mainly follows the PPO algorithm discussed in the background section. The 'learn' function in the 'Agent' class sets up the advantage formula denoted by the advantage parameter, and the critic-value, prob-ratio, and clipped probs are all calculated and iterated for each epoch. The 'choose_action' function is the heart of the model, where it is responsible for choosing the action for the transaction. It takes in observation parameter which gives it data feedback from the last action including stock price data and final profit results, inputs relevant data inside the dense layers in the actor network, then outputs a probability distribution model that determines the course of best action, which is loaded to the training environment.

3.2.4 Actor/ Critic Network

Our agent consists of an Actor-Critic architecture shown in class 'ActorNetwork' and 'CriticNetwork', with separate neural networks instantiated for both the actor and critic, facilitating the decision-making process (Actor) and evaluating the actions taken (Critic). Note the structure follows the one provided in the original paper [2]. Using three dense layers, the 'ActorNetwork' outputs a probability distribution over possible actions, and the 'CriticNetwork' generates value estimates for the states encountered.

3.2.5 Memory

Using a PPOMemory class, the agent employs a replay buffer mechanism, integral to on-policy learning algorithms like PPO. This buffer stores transitions including states, actions, probabilities, values, rewards, and done signals. According to Santhosh, the original author of the PPO framework this model is set up in, the memory class can "reduce the correlation between experiences in updating DNN, increase learning speed with mini-batches, and reuse past transitions to avoid catastrophic forgetting."[6] During learning, the agent samples mini-batches from this buffer, mitigating the correlation between experiences and boosting learning efficacy.

4 Results

We'll present the training and evaluation results for models discussed in Section 3.

4.1 DQN Results

For DQN models: here we only list the result for AAPL, but results on PG & TSLA can be found in the Appendix - Section 8. We deliberately chose these stocks for following reason:

- AAPL (Apple Inc.): Medium volatility (the degree of variation in the price of a financial instrument like stock over time). Apple is a large, established company with a solid market presence, leading to relatively stable stock performance compared to more volatile sectors.

- PG (Procter & Gamble Co.): Low volatility. As a consumer goods company, PG offers stability and is less susceptible to market fluctuations, presenting minimal stock price variations.

- TSLA (Tesla, Inc.): High volatility. Tesla's stock is known for its significant price swings due to various factors, including innovation cycles, regulatory news, and market sentiment towards electric vehicles.

Accordingly, the performance of the DQN agent will be influenced by the volatility of the stock it is trading:

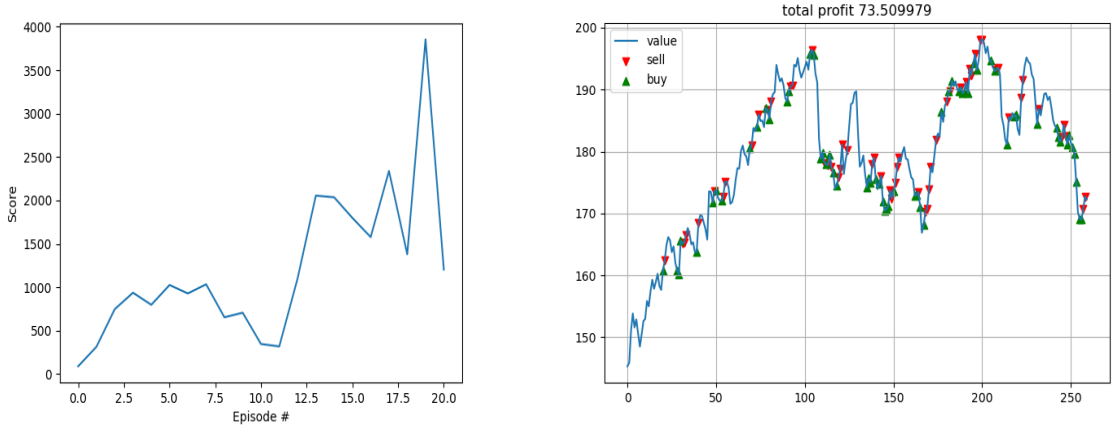
- AAPL: Apple's medium volatility offers a balanced environment for the DQN agent. There are enough price movements for the agent to identify and exploit profitable opportunities, but with less risk compared to high volatility stocks. The agent might achieve a steady performance with a good balance between risk and reward.

- PG: With its low volatility, the agent faces fewer opportunities for large profits due to smaller price movements. However, it's easier for the agent to learn stable and reliable patterns, leading to potentially lower but more consistent profits.

- TSLA: The DQN agent may find more opportunities for profit due to larger price swings, but it also faces higher risk. The agent needs to learn complex patterns to make profitable decisions amidst noise and rapid changes. While high volatility increases potential rewards, it also raises the likelihood of large losses.

To clarify, in plots below, the one on the left (Training Score) represents the total profits earned during each training episode, while the plot on the right (Evaluating Plot) shows the buy & sell action decisions the model made during the evaluation process as well as the final total profit.

4.1.1 DQN Model_1

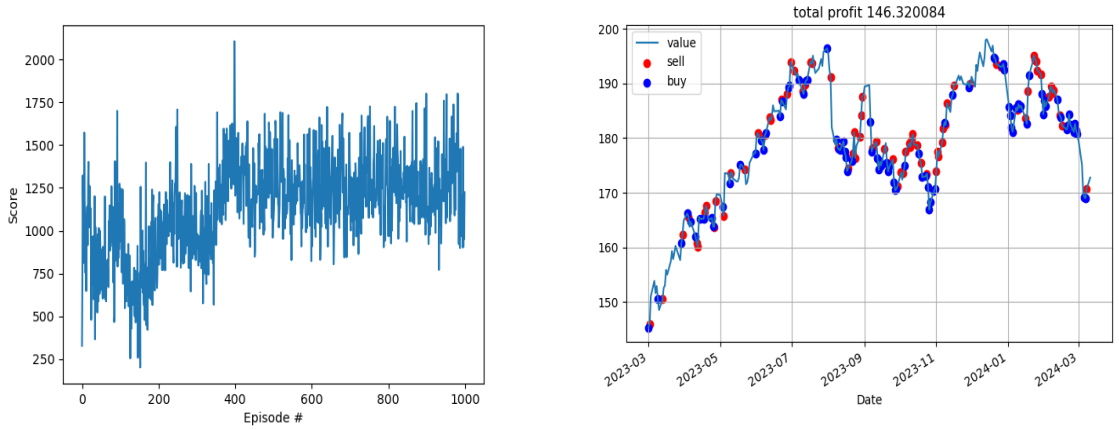


(a) AAPL Training Score-v1 (b) AAPL Evaluating Plot-v1

Figure 5: AAPL Result-v1

As mentioned earlier, due to the execution time constraint, we only train the model for 20 episodes. The training score plot shows that the total profit made during each episode is somewhat unstable, but at least the total profit is positive for all training episodes and the evaluation part. Also, from the evaluating plot, it's evident that the model has learned the basic trading rule: Buy Low & Sell High.

4.1.2 DQN Model_2



(a) AAPL Training Score-v2 (b) AAPL Evaluating Plot-v2

Figure 6: AAPL Result-v2

For model version2, we used 1000 episodes during training. In this case, although the profit score for each episode is still fluctuating, the variation is within certain range as seen in the plot. Also, during evaluation, the model still follows the Buy Low & Sell High rule as before, with higher total profit in the end compared with model version1.

4.1.3 DQN Model_3

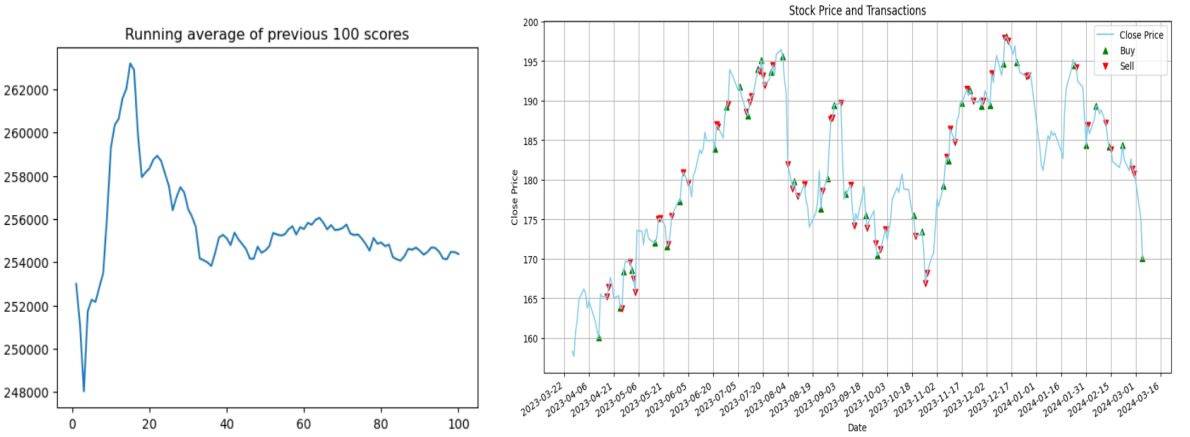
(a) AAPL Training Score-v3 (b) AAPL Evaluating Plot-v3

Figure 7: AAPL Result-v3

For model version3, when we took initial account balance and transaction costs into consideration, which led things to be more complicated and possibly was the reason that the model performance turned to be more unstable and the final profit was lower compared with previous two versions. We predict that the transaction cost incentivises buying more than selling, causing the unbalance between buying actions and selling actions that makes the model volatile. But still, the training scores and the evaluating plot looked quite normal. Due to the time constraint, we may need to investigate the code in more detail in the future to see if there are any problems to cause the such situation.

4.2 PPO Results

Similar to the above DQN Model,the PPO Model takes into account the initial account balance and transaction cost and uses AAPL, PG, and TSLA as train/test stocks. For the above stocks, the training data is selected to be the most recent five years' stock data less the most recent year, and the test data will be the most recent year.



(a) AAPL Training Score-PPO (b) AAPL Evaluating Plot-PPO

Figure 8: AAPL Result-PPO

As shown in Figure 8(a), unlike the DQN model training plots, the PPO training plot exhibits a drastic increase in scores in the beginning, followed by a sharp decrease, and the score fluctuates afterwards. The sharp increase can be explained by the dynamic selection of learning rates, whereas the learning rate at first will be selected to be high as it has low local sensitivity, whereas the decrease could be a potential overshooting, where the model leaves the optimized region. The consequent fluctuations suggest that the model is in an optimization trap where it is difficult to re-reach the initial higher values in a short period. Despite the overshooting could be prevented by selecting a more prudent learning rate value, in practice the final profit and highest score will be lower compared to the values that result in overshooting. One feasible explanation could be the lower choice of learning rate failed to let the model to approximate the optimized point at all.

As for the final behavior of the testing plot shown in Figure 8(b), we can see that compared to the previous DQN models, the PPO model took a more cautious approach toward trading. This is because of the design of the model, where the training period uses a harsher reward relation that discourages frequent trading by installing 10 percent transaction costs as well as a trading frequency modifier. In the actual trading scenario, the restrictions will be lifted, whereas a more prudent approach will be chosen in the actual transactions. The final profit for the model, with differences in different parameters, ranges from -0.6% to 11.6% profit per year, with consideration of transaction costs. While the profit is not significant, it shows the PPO model is working effectively while has the potential to improve.

5 Discussion and Further Work

5.1 Implication of results

We've discussed the results for all versions of our model's training and evaluating performance correspondingly in the former section. In general, the results are reasonable and consistent with what we expect. The DQN agent is able to learn how to make the correct decisions about whether to buy, sell, or hold at correct time-points, but the performance can be bad if situation gets tricky (like TSLA's highly variable stock price) as well as if we take transaction cost and account balance into consideration. For the PPO model, despite the model has shown improvements in the training process and is able to make a positive profit, it can still be enhanced as the training causes overshooting. Such drawbacks can be partially explained by the limitations listed below and may be solved or at least improved if we expand more on the future research listed below.

5.2 Limitations

For DQN models: only consider single stock portfolio management; use only stock's close price to train the model; only design two kinds of reward function: profit & profit rate made during a single step; the neural network structure is relatively simple; haven't considered effects of hyper-parameters like ϵ and γ on model training; each action only considers either selling or buying a constant amount of stock; When transaction cost is taken into consideration, the model soon becomes very volatile, possibly resulting from the unbalance of buying actions and selling actions.

For PPO model: besides the limitations mentioned above, additional limitations include that the training score curve shows the model is not improving after the first ten epochs and the result keeps fluctuating. This might imply that the hard cap of the clip did not effectively prevent the overshooting of the training model. At the same time, the model uses a basic function that determines each transaction amount that is linked with the probability distribution, and such possibilities are not fully explored to optimize this relationship.

5.3 Future work

For DQN models: try to construct a model used for portfolio management of two or more stocks, where in each step we can have more options about the amount of stocks we can sell & buy; test the effects of different reward functions, such as the Sharpe Ratio: $S_T = \text{mean}(R_t)/\text{std}(R_t)$; incorporate other market indicators other than close price of the stock – e.g.: the daily/weekly average price & information other than the historical data such as the economy or companies themselves since the real market behaves rather independently of past performance; explore the effect of different hyper-parameters on model training and performance; finally, design more robust and more efficient Neural Network algorithms to help achieve better results.
For PPO model: besides constructing a full portfolio to have a wider and diverse array of training data, more types of technical parameters can be introduced for the model to have more features. The reward designation can be further improved, as in the exploration process of the model, it has been discovered that the design of the reward determination is crucial in the model training process, as simply setting the reward value as the total profit brings an unsatisfying result. Future models can implement the improvement checking mechanism introduced in the PPO paper, whereby it could guarantee each epoch update will improve model results.

6 Conclusion

This paper presents a detailed investigation into the use of Deep Q-Learning (DQN) and Proximal Policy Optimization (PPO) models for stock market prediction and trading. Through the application of these models to stocks with different levels of volatility, we observed that both DQN and PPO could learn profitable trading strategies to some extent, among which we find that the success of these strategies was sort of influenced by the volatility of the stock, with medium volatility stocks offering a balanced environment for model performance. The inclusion of transaction costs and initial account balances added a layer of realism to our simulation, while at the same time impacting the models' profitability and stability. Our findings suggest that while RL models hold promise for stock trading, their efficacy is heavily dependent on the characteristics of the stock and the complexity of the trading models. Future work could expand on this foundation by exploring more sophisticated and robust models, incorporating a broader range of market indicators, and testing the models in multi-stock portfolio management scenarios. This research contributes to the growing body of knowledge on the application of machine learning in finance, indicating a promising direction for further investigation and development.

7 Contribution

Project Idea & Motivation: Minghao proposed the project idea, then each of us helped discuss the further arrangement.
Data research: Skeleton code for DQN and PPO were found by Zhiqi and George respectively.
Work planning & organization: We all planned and organized meetings. We always met up as a group and collaborated as a team. Since week 6, we met at least twice a week.
Code writing/analysis/test: Zhiqi and Minghao equally contributed to modify, improve, and rewrite the skeleton code of DQN to build up our new models. Zhiqi selected the stocks, and performed test & evaluation together with Minghao on DQN part. George was responsible for code and test & evaluation on PPO.
Report writing: Zhiqi wrote the Abstract, Introduction, and Conclusion section, half of the Background and Methodology section on DQN, as well as the final proofreading. Minghao wrote another half of the Background section and Methodology section on DQN. Zhiqi wrote the analysis in the DQN Result section. George wrote all texts related to PPO. For Discussion section, all group members shared equal contribution.

References

[1] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," 2017, `https://arxiv.org/abs/1404.3978`.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms" 2017, `https://arxiv.org/abs/1707.06347`.

[3] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization (TRPO)," *CoRR abs/1502.05477*, 2015, `https://arxiv.org/abs/1502.05477`

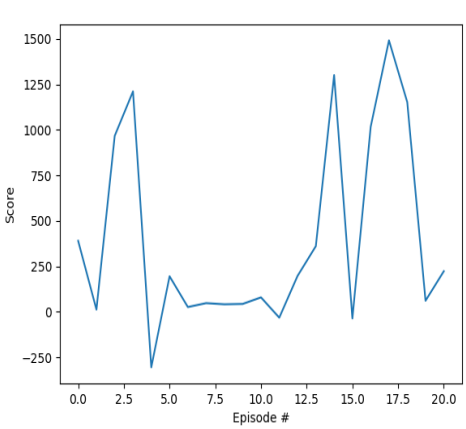
[4] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," *Robotica*, vol. 17, no. 2, pp. 229-235, 1999.

[5] S. Amari, "Natural Gradient Works Efficiently in Learning," in *Neural Computation*, vol. 10, no. 2, pp. 251-276, 1998, doi: 10.1162/089976698300017746.

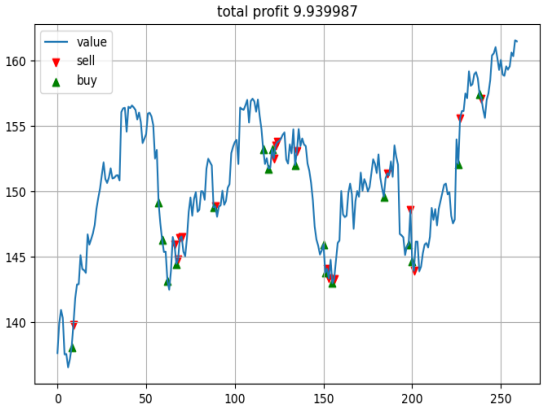
[6] S. Santhosh, "Reinforcement Learning (Part 8): Proximal Policy Optimization (PPO) for Trading," 2023. medium.com/@sthanikamsanthosh1994/reinforcement-learning-part-8-proximal-policy-optimization-ppo-for-trading-9f1c3431f27d.

8 Appendix - Results Continued

8.1 PG Results

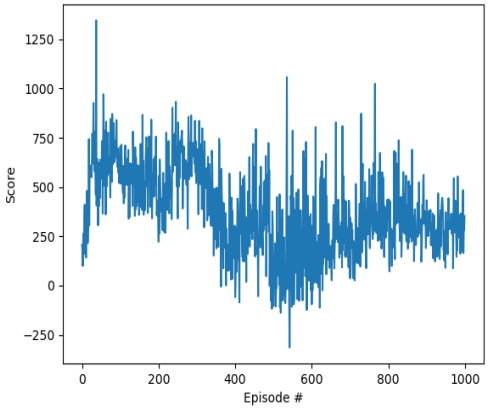


(a) PG Training Score-v1

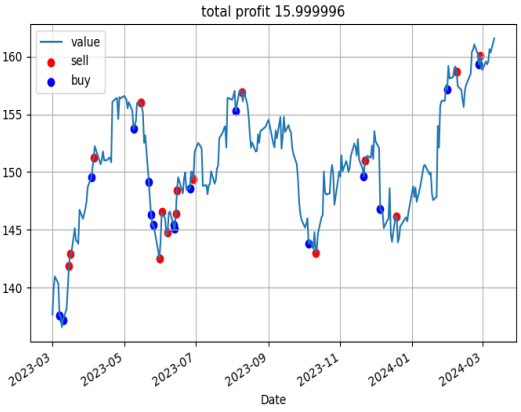


(b) PG Evaluating Plot-v1

Figure 9: PG Result-v1

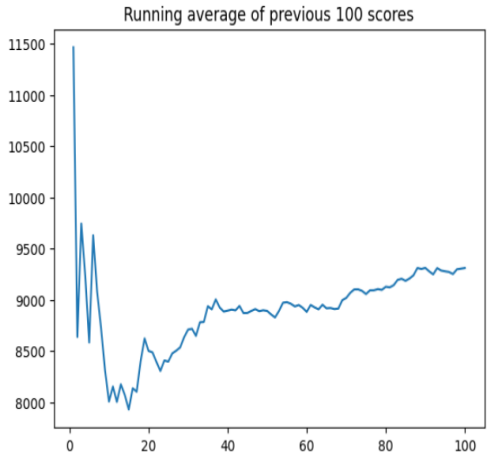


(a) PG Training Score-v2

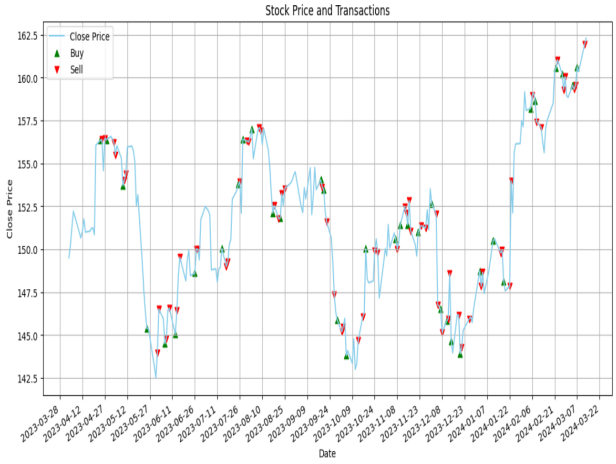


(b) PG Evaluating Plot-v2

Figure 10: PG Result-v2



(a) PG Training Score-PPO

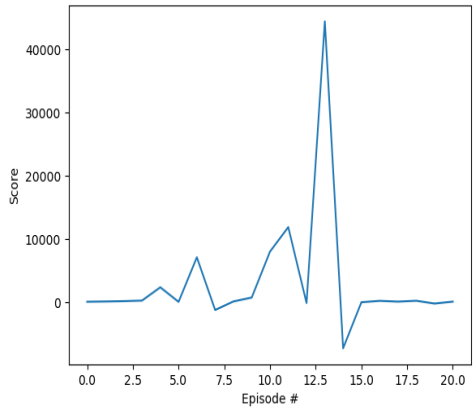


(b) PG Evaluating Plot-PPO

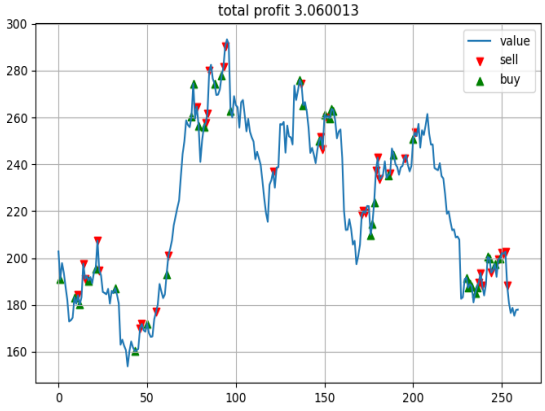
Figure 11: PG Result-PPO

As discussed in Section 4.1, PG has low volatility, which makes it harder for the agent to detect what's the correct decisions to buy and sell since there are not much room to make profits. The evaluating plots for both DQN model version1&2 are consistent with this point, as we can find that the trading frequency is much lower and the profits are much lower compared with AAPL's results from above. We didn't visualize the results of DQN model version3 on PG since the model is too volatile and we couldn't get a visualisation of PG using model version3 that makes sense.
The training score graph shown for TSLA shows a similar characteristic as for AAPL, where it results in a sharp increase in training score, then subsequently followed by a drastic decrease, with the following training scores either increasing at a moderate pace or fluctuating. In this case, we may assume that such observed characteristic is not limited to a single stock, but may be derived from the model itself. Further analysis can be shown in the training dataset for TSLA.

8.2 TSLA Results

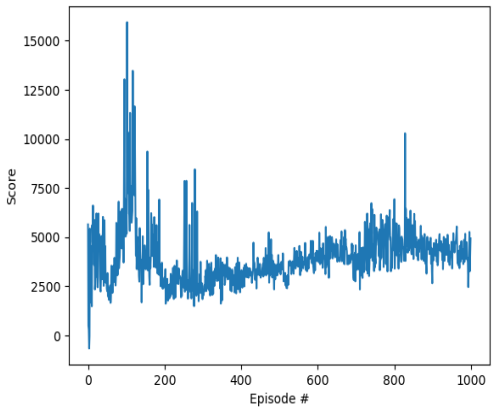


(a) TSLA Training Score-v1

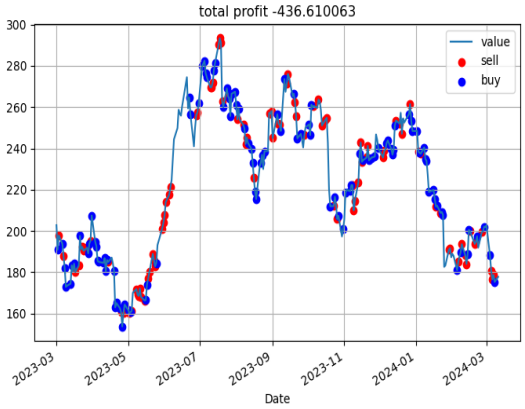


(b) TSLA Evaluating Plot-v1

Figure 12: TSLA Result-v1

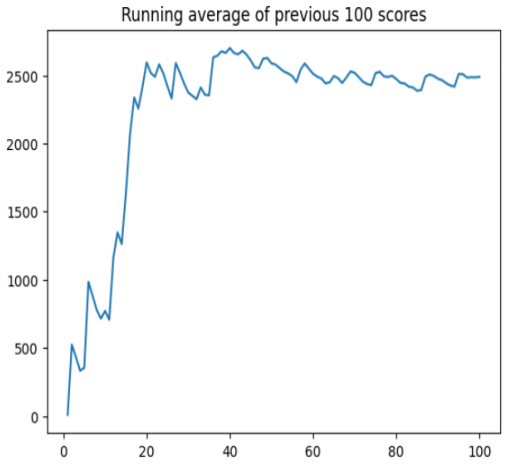


(a) TSLA Training Score-v2

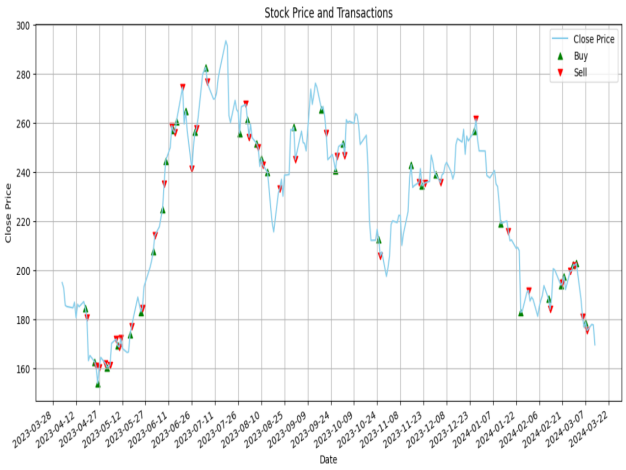


(b) TSLA Evaluating Plot-v2

Figure 13: TSLA Result-v2



(a) TSLA Training Score-PPO



(b) TSLA Evaluating Plot-PPO

Figure 14: TSLA Result-PPO

Results for TSLA also follow the argument discussed in Section 4.1. With its high volatility and large price swings, both the chances to make profit and the risks to lose money increase simultaneously, greatly raising the difficulty for the agent to capture the rapid price change and profit-making policy. As for evaluation, even though the model tries to follow the Buy Low & Sell High rule, the performance is still kind of unsatisfactory (nearly 0 for DQN model v1 and huge loss for model v2). For example, in evaluating plot, within the region between 2023-5 to 2023-7, the stock price increases dramatically and the price difference is about $100, which may not be expected by the agent. We didn't visualize the results of DQN model version3 on TSLA since the model is too volatile and we couldn't get a visualization of TSLA using model 3 that makes sense.
For PPO, it presents the same issue described in the AAPL stock training analysis: the training scores will stop optimizing after the first few epochs but continue to fluctuate around the same score. The reason the peaked score does not approximate the optimized score is because even tested in the trained dataset, the model outputs suboptimal profit, hinting it has the potential to prove on a greater scale.