A review on Reinforcement Learning

1 Introduction

Reinforcement Learning (RL) is a branch of machine learning that aims to learn optimal strategies through the interaction between an agent and its environment. The agent takes actions in the environment, receives feedback in the form of rewards or penalties, and adjusts its strategy based on this feedback to maximize long-term returns.

In supervised learning, algorithms are trained using labeled ('supervised') data, with the goal of minimizing the error between predictions and actual labels. In contrast, reinforcement learning does not rely on labels, but instead uses reward signals to guide the agent's decision-making. On the other hand, while both unsupervised learning and reinforcement learning do not require labeled data, reinforcement learning focuses on learning strategies to maximize long-term rewards, whereas unsupervised learning typically aims to uncover hidden structures within the data.

Reinforcement Learning consists of the following major elements.

Agent is the entity that performs actions. Its goal is to select appropriate actions based on the state of the environment and learn through interaction with the environment to maximize cumulative rewards.

Environment refers to the system or external world where the agent operates. The agent interacts with the environment, which responds to the agent's actions by providing feedback (rewards or penalties) and updating the state.

State $\boldsymbol{s}$ is the system or external world where the agent operates and interacts. It responds to the agent's actions, provides feedback, for example, rewards, and updates its state.

Action $\boldsymbol{a}$ is a decision the agent can make in a given state. The agent's task is to choose an optimal action based on the current state to achieve the highest return.

Reward $\boldsymbol{r}$ is the feedback provided by the environment to the agent, evaluating the quality of the agent's behavior. Rewards can be positive (encouraging behavior) or negative (punishing behavior). The agent's goal is to maximize cumulative rewards through a series of actions.

Policy $\boldsymbol{\pi}$ is a function that the agent uses to decide which action to take in a given state. Policies can be deterministic (always choosing the same action in a state) or stochastic (choosing actions with certain probabilities in a state).

Value Function $\boldsymbol{V(s)}$ or **Action-Value Function** $\boldsymbol{Q(s,a)}$. A value function evaluates the quality of a state or a state-action pair. The state value function $V(s)$ measures the expected return the agent can achieve from a specific state. The state-action value function $Q(s,a)$ measures the expected return from choosing a specific action in a specific state. Agents typically learn these functions to optimize their policies.

Discount Factor γ represents the present value of future rewards. Since the goal of reinforcement learning is to maximize long-term returns, the agent usually assigns less weight to future rewards. The discount factor γ is between 0 and 1, with values closer to 1 indicating that the agent prioritizes long-term returns.

Framework of Reinforcement Learning begins with initializing the agent and environment, defining the state space, action space, and reward mechanism. The agent selects an action based on its current policy, performs the action, and the environment updates its state and provides a reward, then updates its policy using the reward signals to improve future returns. This process is repeated until the task is completed or convergence criteria are met.

2 Development of Reinforcement Learning

Reinforcement Learning can be traced back to psychology and behaviorism theories. In the 1950s, Richard Bellman introduced the concept of Dynamic Programming which is the theoretical groundwork for value functions and policy evaluation in RL. Markov Decision Processes were then formalized as a mathematical model to describe the interaction between an environment and an agent.

In 1989, Chris Watkins developed Q-learning, a value iteration method for finding optimal policies by learning state-action value functions. In the 1990s, policy gradient methods emerged to directly optimize policies. Unlike traditional value function methods, policy gradient methods compute the gradient of the policy with respect to long-term returns to optimize policy parameters. However, as state and action spaces grew, classical Q-learning struggled to handle large-scale problems. To address this, researchers began using function approximation techniques, including neural networks, to estimate Q-values. In my project, I am going to investigate the use of neural networks in RL.

In 2013, DeepMind introduced the Deep Q-Network (DQN), which combined deep neural networks with Q-learning to handle large state spaces. DQN demonstrated significant success in complex environments, such as Atari games. DQN employs the technique of Experience Replay to break the correlation between data, thereby improving training stability. Additionally, it introduces a Target Network to address the high volatility in Q-value updates. After this, optimization methods such as TRPO (Trust Region Policy Optimization) and PPO (Proximal Policy Optimization) were proposed, enhancing traditional policy gradient methods to make them more stable and easier to implement. PPO has become one of the most popular algorithms in modern reinforcement learning.

There are some other improved algorithms. Double Q-learning addresses the overestimation problem in standard Q-learning. By using two Q-value functions, it reduces the impact of overestimation and improves stability. Prioritized Experience Replay weights experience replay, allowing experiences with high errors to be sampled more frequently, thus accelerating the learning process. Entropy Regularization is to encourage exploration, where reinforcement learning algorithms introduce an entropy term, rewarding uncertain behaviors and avoiding local optima. For problems involving continuous action spaces, algorithms such as Deep Deterministic Policy Gradient (DDPG) and Actor-Critic methods (A3C) have been proposed. By sharing learned knowledge, such as through Universal Value Function Approximators (UVFA), reinforcement learning models can transfer value functions across multiple tasks.

Reinforcement learning can be divided into the following types. **Value-based reinforcement learning**, such as Q-learning and DQN, learns a state-action value function to find the optimal policy. **Policy-based reinforcement learning**, such as PPO and TRPO, directly learns the optimal policy instead of a value function. Actor-Critic methods, such as A3C and A2C, combine value functions and policy functions, using two networks to learn policies and values separately. Deep reinforcement learning, which uses deep neural networks as function approximators to handle high-dimensional and complex state spaces. Multi-agent reinforcement learning deals with situations where multiple agents work together, with each agent making independent decisions.

Reinforcement learning has achieved significant results in various fields, such as natural language processing, robotic control, and gaming. However, it also faces many challenges, such as low sample efficiency, stability issues, and the balance between exploring unknown strategies and exploiting known strategies.

3 Methodology of Reinforcement Learning

The statistical model of reinforcement learning is often represented using a Markov Decision Process. It is denoted as (S, A, P, R, γ), where $P(s' \mid s, a)$ is the state transition probability, representing the probability of transitioning to state s' after taking action a in state s.

The goal of reinforcement learning is to maximize the agent's long-term return. The cumulative reward from time step t is denoted as

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where R_{t+k+1} is the reward received at time step $t + k + 1$.

The discounted return can also be written as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

which shows the emphasis on future rewards.

The agent aims to find an optimal policy π^* that maximizes the expected return starting from any state. The optimal state-value function $V^*(s)$ represents the maximum expected return starting from state s under the optimal policy

$$V^*(s) = \max_{\pi} V^{\pi}(s),$$

where $V^{\pi}(s)$ is the state-value function under a policy π.

The optimal action-value function $Q^*(s, a)$ represents the maximum expected return starting from

state s, taking action a, and thereafter following the optimal policy

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a),$$

where $Q^{\pi}(s,a)$ is the action-value function under a policy π.

The optimal policy $\pi^*(s)$ can be derived by maximizing the action-value function:

$$\pi^*(s) = \arg\max_{a} Q^*(s,a).$$

Next, I will introduce in detail two types of popular reinforcement learning algorithm that I tried in my project.

3.1 Methodology of Value-based reinforcement learning

Q-learning is a value-based reinforcement learning algorithm that aims to learn an optimal action-value function $Q^*(s,a)$, which represents the maximum expected return achievable from a given state s and action a. Q-learning uses a greedy policy to maximize returns, that is, it chooses the action that maximizes the action value in each state.

The key formula in Q-learning is the Q-value update rule, which is used to estimate the return for a state-action pair

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

where R_{t+1} is the immediate reward received after taking action a_t at time step t and $\max_{a'} Q(s_{t+1},a')$ is the maximum Q-value for the next state s_{t+1}, representing the best possible action to take.

DQN (Deep Q-Network) is an algorithm that combines Q-learning with deep neural networks to address the challenges of applying traditional Q-learning in high-dimensional state spaces, such as images. In DQN, the Q-function is approximated using a neural network, enabling it to handle more complex environments and high dimensional state spaces.

In DQN, the Q-function is represented by a neural network $Q(s,a;\theta)$, where θ are the parameters of the neural network. The Q-value update formula in DQN is similar to traditional Qlearning but uses a neural network for approximation. Specifically, DQN uses a target network Q_{target} to stabilize training. The parameters of the target network are updated periodically every N steps with the update formula

$$y_t = R_{t+1} + \gamma \max_{a'} Q_{\text{target}}(s_{t+1},a';\theta^-)$$
$$L(\theta) = \mathbb{E}_{s_t,a_t}\left[\left(y_t - Q(s_t,a_t;\theta)\right)^2\right],$$

where y_t is the target value, representing the estimated return using the current reward R_{t+1} and the maximum Q -value from the target network, $Q(s_t, a_t; \theta)$ is the output from the current Q-network, $Q_{\text{target}}(s_{t+1}, a'; \theta^-)$is the Q-value output from the target network, which is updated every N steps and $L(\theta)$ is the loss function to measure the difference between the current Q-value and the target value.

Also, DQN uses Experience Replay, where the agent's experiences (state, action, reward, next state) are stored in an experience replay buffer. Batches of experiences are sampled randomly from this buffer for training. This helps reduce correlations between consecutive data points and improves the stability of training.

To avoid rapid fluctuations in Q-values that could destabilize training, DQN uses a target network. The parameters of the target network Q_{target} are updated periodically every N steps as

$$\theta^- \leftarrow \theta.$$

This target network approach helps mitigate the overestimation problem in Q-learning and stabilizes training.

Algorithm 1: Deep Q-learning with Experience Replay

1. **Initialize** replay memory D to capacity N
2. **Initialize** action-value function Q with random weights
3. **For** episode = 1 **to** M **do**:
   4. **Initialize** sequence $\mathrm{s}_1 = \{\mathrm{x}_1\}$ and preprocess sequence $\varphi_1 = \varphi(\mathrm{s}_1)$
   5. **For** $\mathrm{t} = 1$ **to** T **do**:
   6. With probability ε, select a random action a_t
   7. Otherwise, select $a_t = \max, Q^\star(\varphi(s_t), a; \theta)$
   8. Execute action a_t in emulator and observe reward r_t and image x_{t+1}
   9. **Set** $\mathrm{s}_{\mathrm{t}+1} = \mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}}, \mathrm{x}_{\mathrm{t}+1}$ and preprocess $\varphi_{\mathrm{t}+1} = \varphi(\mathrm{s}_{\mathrm{t}+1})$
   10. Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in D
   11. Sample random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from D
   12. **Set** $\mathrm{y_j} =$
       $\mathrm{r_j}$ for terminal $\varphi_{\mathrm{j}+1}$
       $r_j + Y^* \max_s{}' \mathrm{Q}(\varphi_{\mathrm{j}+1}, a'; \theta)$ for non-terminal $\varphi_{\mathrm{j}+1}$
   13. Perform a gradient descent step on $\left(y_j - Q(\varphi_j, a_j; \theta)\right)^2$
   14. **End For**
4. **End For**

3.2 Methodology of policy gradient-based reinforcement learning

TRPO (Trust Region Policy Optimization) is a policy gradient-based algorithm that aims to stabilize policy updates by limiting the size of updates within a trust region, preventing large changes that could

degrade performance. The objective of TRPO is to maximize the objective function while constraining the update within the trust region.

The TRPO objective function is

$$L(\theta) = \mathbb{E}^t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}^t\right],$$

where $\hat{A}^t$ is the advantage function, which represents the relative advantage of taking action a_t in state s_t.

TRPO's idea is to constrain the policy update by ensuring the difference between the new and old policies does not exceed a predefined threshold, typically using KL-divergence

$$\mathbb{E}^t\left[D_{\text{KL}}\left(\pi_{\theta_{\text{old}}}(a_t \mid s_t)\|\pi_\theta(a_t \mid s_t)\right)\right] \leq \delta,$$

Where D_{KL} is the Kullback-Leibler divergence, and δ is a pre-defined threshold that limits the policy update step.

Advantages of TRPO is its stability, ensuring policy updates remain within a trust region, but to compute Hessian matrices and perform inverse operations is computationally expensive for large-scale problems. In this case, PPO was introduced.

PPO (Proximal Policy Optimization) is another policy gradient-based reinforcement learning algorithm that directly optimizes the policy to learn the optimal strategy. It is an improvement over TRPO. PPO uses a clipped objective function to prevent excessively large policy updates, ensuring stable optimization.

A major part of PPO is its clipped objective function, which compares the ratio of the new policy to the old policy's probability during updates. The objective function is

$$L_{\text{CLIP}}(\theta) = \mathbb{E}^t\left[\min\left(r_t(\theta)\hat{A}^t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^t\right)\right],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the ratio between the new policy and the old policy, representing the change between the current and previous policies, $\hat{A}^t$ is the advantage function at time step t, which represents the relative advantage of taking action a_t in state s_t, and ϵ is a small hyperparameter that controls the extent of policy updates to prevent overly large changes. PPO aims to optimize this objective function to maximize long-term rewards while maintaining stable updates.

The advantage function $\hat{A}^t$ represents the difference between the actual return from a state-action pair and the baseline (value function). A common method for estimating the advantage function is Generalized Advantage Estimation (GAE)

$$\hat{A}^t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_t^l,$$

where δ_t is the temporal difference (TD) error at time step t

$$\delta_t = R_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t).$$

PPO has become one of the most popular policy optimization methods in reinforcement learning and is widely used in both continuous and discrete action space problems.

Algorithm 2: PPO in Actor-Critic Style

1. **For** iteration $= 1,2, \ldots$ **do**:
   2. **For** actor $= 1,2, \ldots, N$ **do**:
   3. Run policy π_{tol} in environment for T timesteps
   4. Compute advantage estimates $\hat{HAA}_1, \ldots, \hat{H}A_t$
   5. **End For**
   6. Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
   7. $\theta_{\text{tol}}\,\text{d} \leftarrow \theta$
2. **End For**

4 Implementation of Reinforcement Learning

In this section, I will attempt to complete simple reinforcement learning tasks using Python, based on the methodology of DQN and PPO. The codes for this project can be found on [mingyuzhang2/2024-Fall-5205-Linear-Regression-Model-Final-Project]

4.1 Task and Objective

In the original paper of DQN, *Playing Atari with Deep Reinforcement Learning*, a suite of Atari 2600 games was used as reinforcement learning tasks. Atari 2600 is a challenging RL testbed that presents agents with a high dimensional visual input (210*160 RGB video at 60 Hz) and a diverse and interesting set of tasks that were designed to be difficult for human players. The goal is to train an agent to achieve high scores in diverse games like Breakout, Pong, and Space Invaders, where each game presents a unique challenge, such as hitting a ball with a paddle, winning a table tennis match, or shooting aliens. In the original paper of PPO, *Proximal Policy Optimization Algorithms,* the RL tasks included both Atari games and continuous control tasks. For Atari games, the goal is typically to achieve high scores in video games by learning optimal policies from raw pixel inputs. Therefore, my initial goal was to attempt using reinforcement learning to play an Atari game and compare the results. However, I quickly realized that Atari games are actually very large and complex, requiring millions of environment steps/many hours of training. In this case, I tried to consider a much simpler game for reinforcement learning.

CartPole task is a classic reinforcement learning benchmark in which an agent must balance a pole on a moving cart. The cart moves along a track, and the agent applies forces to the cart to keep the pole upright. The objective is to maximize the total time the pole remains balanced without falling over or the cart moving out of bounds. It is a relatively simple environment often used to test and compare RL algorithms.

For Python implementation, I used the CartPole-v1 environment, created by OpenAI's Gym. This can be created by using a Python library called gym which can be easily be installed to laptop.

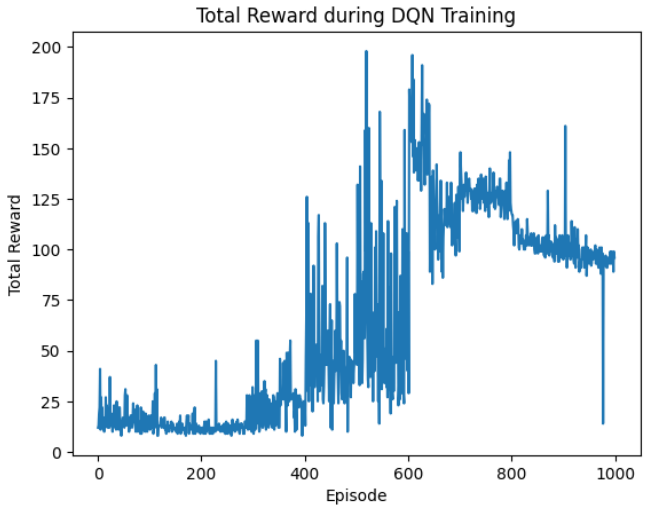
4.2 Implementation of DQN

I first attempted to implement DQN, although I found that the implementation of PPO seemed more

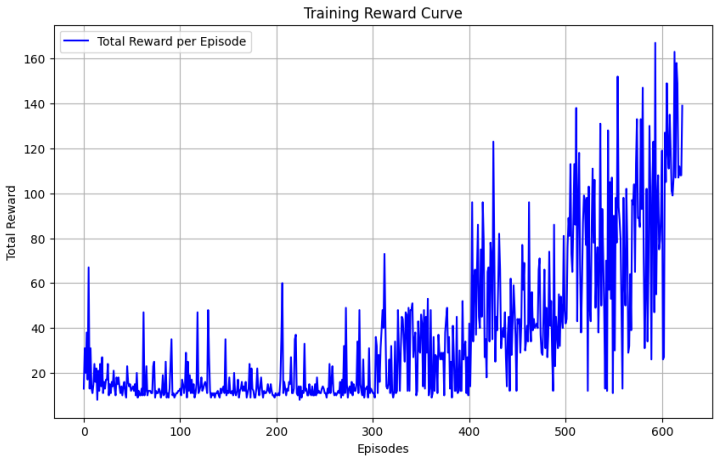
straightforward in comparison.

First is to create the QNetwork. In the original paper convolutional layers were used, but since CartPole is simpler, I only used three linear layers, which did perform well. Then I created a replay buffer, the epsilon greedy policy, and the DQN trainer. Writing loops and passing parameters actually took me a significant amount of time. For hyperparameters, I looked for inspiration from how others implemented their solutions. For the optimizer, I used Adam and a small learning rate.

In the actual training process, I found that waiting for convergence or achieving a good result was far from ideal. Ideally, the rewards should increase steadily or stabilize at a desirable level. In my training, the rewards usually stabilized around 100 of total reward within 1000 episodes, but this did not seem to be a high value. A better result would look something like the chart shown below.



When I set the threshold at 100, the result converged in 622 episodes, although the chart shows some significant fluctuations (but my code indicated that it converged, and the convergence here is not rigorous). I used different colors in the plot without paying attention, but the colors have no actual meaning. Also after completing the plot, I realized that perhaps using a scatter plot might provide a better visual effect.

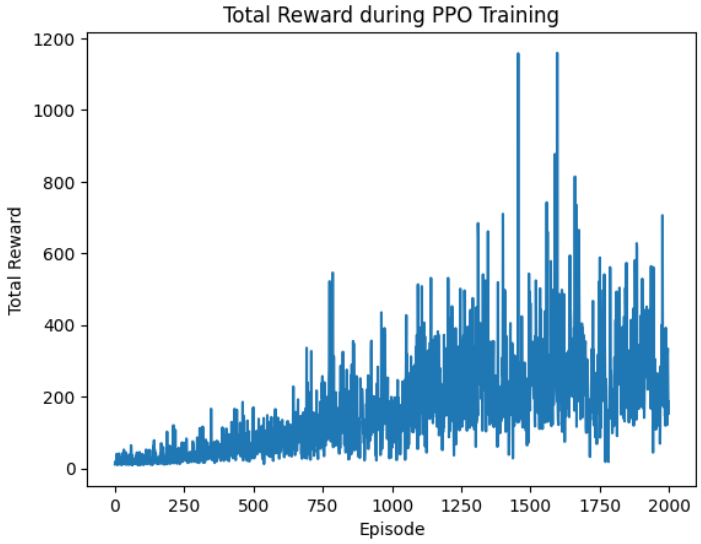


One major issue I encountered during training was the training time. Initially, I used my personal computer, and the training process was extremely slow—1000 episodes took over three hours. After

switching to GCP with 1 NVIDIA T4 GPU, 32 vCPUs (16 cores), and 120GB of RAM, the training speed increased by more than tenfold. However, at the same time, the virtual machine was prone to crashes, and once it crashed, I had to restart the training from scratch. To address this, I added a checkpoint-saving feature in the code to save progress every ten episodes.

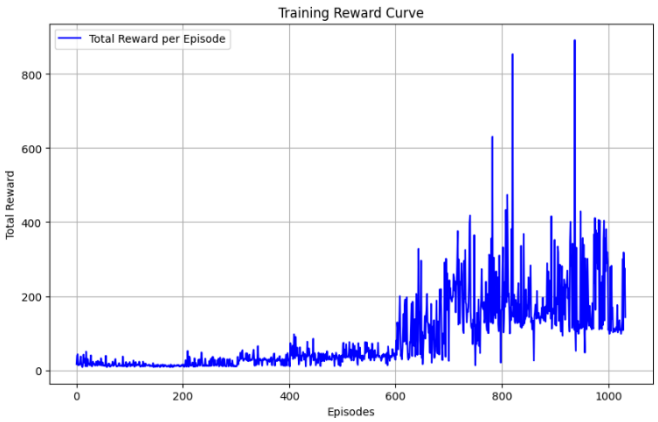
4.3 Implementation of PPO

I then attempted to implement PPO. Compared to DQN, building PPO was more straightforward and simpler, but the challenges during the actual training process took more of my time. Initially, I tried using parameters that were largely consistent with those in DQN. As shown in the figure below, the rewards stabilized around 50 of total reward, which is much lower than the previous result of around 100. To address this, I attempted to increase the number of episodes:

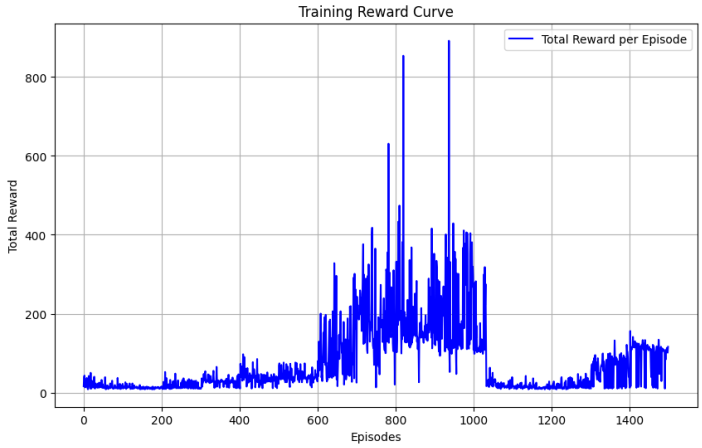


It can be seen that the rewards stabilized around 200, which is significantly higher than the previous value of around 50, which is good. However, there seemed to be substantial fluctuations. I continued to increase the number of episodes, but the improvement was not very significant. At the same time, the training time also kept increasing, since the more the model learned, the longer it could play the game, which led to more time being required. I then tried adjusting the learning rate and batch size, as both of these should help with convergence.

By increasing the batch size, I obtained

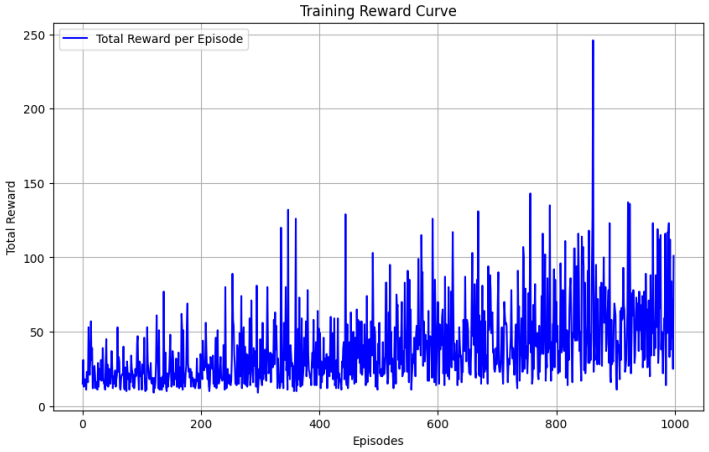


which reached around 200 at about 600 episodes. Interestingly, my kernel crashed during the training process, and when I resumed training, I obtained



which eventually stabilized around 60. This situation, where the results did not recover to the previous performance, has been discussed as potentially due to not setting the seed for reproducible training.

I also changed to a smaller learning rate, and it quickly went around 50.



4.4 Discussion of the Results

First, from my results, it is clear that I successfully built DQN and PPO models, regardless of whether the results were ideal. However, unlike traditional statistical models, which provide results that can be directly obtained, deep reinforcement learning models have uncertainty in each result, and hyperparameter adjustments cannot directly determine the outcome. At the same time, even though I chose relatively simple tasks, the training time was still quite long. This made me realize that this course has provided me with more than just the perspective of linear regression models, which broadened my horizon and taught me a great deal.

5 Future Prospects

From this project, I got to know two very popular reinforcement learning algorithms and personally experienced many of the issues encountered when working with reinforcement learning. In this regard, some prospects related to reinforcement learning can be further considered.

Computational cost of RL is big. Future advancements can focus on improving training efficiency through techniques like parallelization, distributed learning, and GPU optimization. Additionally, more advanced reward shaping and experience replay strategies could help accelerate.

The performance of reinforcement learning heavily depends on the choice of hyperparameters. Automated methods like Bayesian optimization, genetic algorithms, or meta-learning could be employed to find optimal parameter configurations more efficiently. Moreover, adaptive learning rate schedules and dynamic exploration strategies might allow the model to adjust based on the progress of training.

Scalability and Robustness. Techniques such as curriculum learning, where the agent is trained on progressively harder tasks, and meta-reinforcement learning, where the agent learns to adapt to a variety of tasks, could allow the models to generalize better and be more adaptable to new, unseen environments.

There are many algorithm improvements that have already been considered. One popular direction is Double DQN, Dueling DQN, and Prioritized Experience Replay which improve performance in terms of stability and sample efficiency. Model-based reinforcement learning approaches, where the agent learns a model of the environment and uses it to plan its actions, can also significantly reduce the need for extensive interaction with the environment.

Reference

[1] Mnih, V., 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[2] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[3] Schulman, J., 2015. Trust Region Policy Optimization. *arXiv preprint arXiv:1502.05477*.

[4] Arulkumaran, K., Deisenroth, M.P., Brundage, M. and Bharath, A.A., 2017. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), pp.26-38.

[5] Watkins, C.J. and Dayan, P., 1992. Q-learning. *Machine learning*, *8*, pp.279-292.

[6] Li, Y., 2017. Deep Reinforcement Learning: An Overview. *arXiv preprint arXiv:1701.07274*.

[7] Mousavi, S.S., Schukat, M. and Howley, E., 2018. Deep reinforcement learning: an overview. In *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016: Volume 2* (pp. 426-440). Springer International Publishing.

[8] Buşoniu, L., Babuška, R. and De Schutter, B., 2010. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pp.183-221.

[9] AmazingAng. (2022). WTF-DeepRL. GitHub. https://github.com/AmazingAng/WTF-DeepRL