Final Project Report

Hang Zheng 520021911347

June 8, 2023

Contents

1 Introduction **2**

2 Preliminaries **2**

2.1 Environment . . . 2

2.1.1 VideoPinball-ramNoFrameskip-v4 . . . 2

2.1.2 Ant-v2 . . . 2

2.2 Value-based algorithms . . . 4

2.2.1 DQN . . . 4

2.2.2 DDQN . . . 4

2.2.3 Dueling DQN . . . 6

2.2.4 Rainbow . . . 6

2.3 Policy-based algorithms . . . 7

2.3.1 DDPG . . . 7

2.3.2 PPO . . . 8

2.3.3 SAC . . . 9

3 Experimental Result **10**

3.1 Value-based algorithm . . . 10

3.2 Policy-based algorithm . . . 12

4 Conclusion **14**

1 Introduction

In this project, we need to use two model-free reinforcement learning algorithms, namely value-based and policy-based, to solve environments with discrete and continuous action spaces, and test them on the Atari and Mujoco environments respectively.

As for value-based algorithms, I implemented DQN, DDQN, Dueling DQN and Rainbow, and tested them on the Atari environment VideoPinball-ramNoFrameskip-v4. For policy-based algorithms, I implemented DDPG, PPO and SAC algorithms, and tested them on the Mujoco environment Ant-v2.

In this report, I will provide a detailed introduction to the principles of the algorithms mentioned above, present my experimental results, and provide an detailed analysis of the results.

2 Preliminaries

In this section, I will give a brief introduction to the gym environments and the principle of the algorithms we mentioned above.

2.1 Environment

2.1.1 VideoPinball-ramNoFrameskip-v4

VideoPinball-ramNoFrameskip-v4 is an Atari game environment in the OpenAI Gym. It is a simulator of the arcade game "Video Pinball" developed by Atari in the 1980s. In this environment, players need to control a paddle to bounce a ball, hitting bricks at the top to score points. There are multiple bricks and obstacles in the game, and players need to avoid collisions and get higher scores by controlling the movement of the paddle and the direction of the ball's bounce. The environment uses the ram format to represent the game state, which means that all pixels in the game are converted into a one-dimensional array of length 128. Therefore, it does not involve image processing. Compared with other Atari game environments, the state representation of this environment is relatively simple, but it still has a certain degree of difficulty and challenge. The NoFrameskip version of this environment means that the action will not be repeated, i.e., each action corresponds to only one game frame, which reduces the difficulty and complexity of the game.

The space setting of 'VideoPinball-ramNoFrameskip-v4':

- observation_space: Box(0, 255, (128,), uint8)
- action_space: Discrete(9)

The diagram of 'VideoPinball-ramNoFrameskip-v4' is shown below in Figure 1.

2.1.2 Ant-v2

Ant-v2 is a continuous control task environment in the OpenAI Gym. In this environment, players need to control a quadruped robot (Ant) to walk on a plane to achieve a preset goal. Ant's limbs can apply forces in different directions to control its movement. The goal of this environment is to make Ant run as far as possible while avoiding obstacles, including walls and changes in terrain height. The Ant-v2 environment provides a continuous 128-dimensional state vector, including information such as the position, velocity, and angle and speed of the four limbs of Ant. The action space of this environment is a continuous 8-dimensional vector, representing the magnitude and direction of the forces applied to the limbs. Compared with other continuous control task environments, the Ant-v2 environment has a higher level of complexity and challenge, requiring more complex control strategies and learning algorithms to complete the task.

The space setting of 'Ant-v2':

- observation_space: Box(-inf, inf, (111,), float64)
- action_space: Box(-1.0, 1.0, (8,), float32)

The diagram of 'Ant-v2' is shown below in Figure 2.

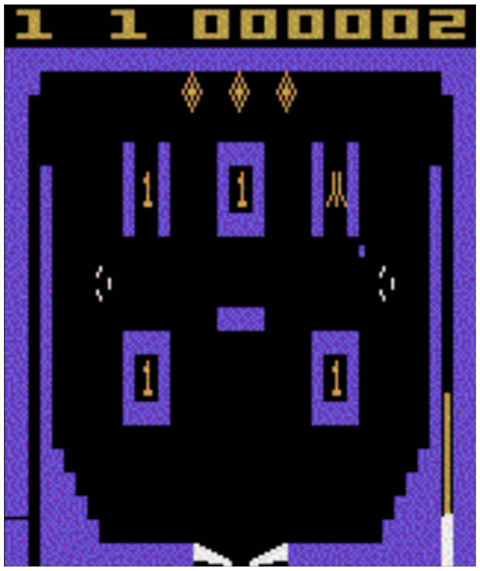


Figure 1: VideoPinball-ramNoFrameskip-v4

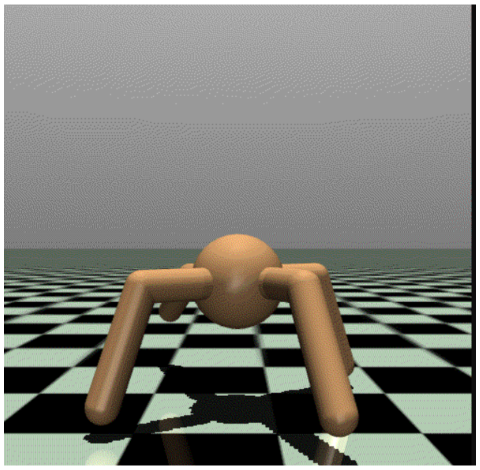


Figure 2: Ant-v2

2.2 Value-based algorithms

Value-based algorithms use value functions to select optimal actions and mainly focus on estimating the value of different states or state-action pairs, i.e., Q-values or V-values. Value-based algorithms are typically used in tasks with discrete actions and finite state spaces, where the value function can be represented as a Q-table or neural network, and the model can maintain high stability and convergence during the learning process.

Some traditional value-based algorithms such as **Q-learning** [1] and **SARSA** have achieved good results. Since the introduction of the **Deep Q-Network (DQN)**[2] algorithm, neural networks have been used to learn the Q-function, and techniques such as experience replay and target network have been introduced to improve learning efficiency and stability, effectively improving the performance of value-based algorithms. **Double DQN (DDQN)**[3] introduces two neural networks to estimate the Q-value based on the DQN, which helps to mitigate the overestimation issue of the Q-value in DQN. **Dueling DQN**[4] is an improved DQN algorithm that decomposes the Q-value into a state value function and an advantage function, which can better learn the value and importance of actions. **Rainbow**[5] combines the features of the above algorithms, including DDQN, Dueling DQN, prioritized experience replay, n-step returns, etc., and significantly improves the performance of the DQN algorithm without increasing too much computational cost.

In the following sections, I will give introduction to **DQN, DDQN, Dueling DQN and Rainbow**.

2.2.1 DQN

DQN, short for Deep Q-Network, is a **deep learning-based** reinforcement learning algorithm. Its background lies in addressing the limitations of traditional reinforcement learning algorithms in handling high-dimensional state and action spaces, as well as the difficulties of traditional Q-learning algorithms in dealing with continuous state and action spaces.

The core of the DQN algorithm is to **use a deep neural network to approximate the Q-value function.** Unlike traditional reinforcement learning algorithms that need to maintain a Q-value table, the DQN algorithm uses a neural network to learn the Q-value function, with the current state as input and outputting the Q-value for each corresponding action. This approach can handle problems in high-dimensional state and action spaces, as well as continuous state and action spaces.

The training objective of the DQN network is to minimize the following loss function:

$$L(\theta) = E_{(s,a,r,s') \sim U(D)}[(y - Q(s,a;\theta))^2]$$

where s represents the current state, $a$represents the action to be taken, r represents the reward, s' represents the next state, $U(D)$ represents the samples in the replay buffer, $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$ represents the expected Q-value, θ represents the parameters of the neural network, θ^- represents the parameters of the target neural network, γ represents the discounting factor.

When training a DQN network, two tricks, namely Experience Replay and Target Network, are usually used to improve the stability of the algorithm and speed up the convergence.

The pseudo code of the DQN algorithm is as shown in figure 3.

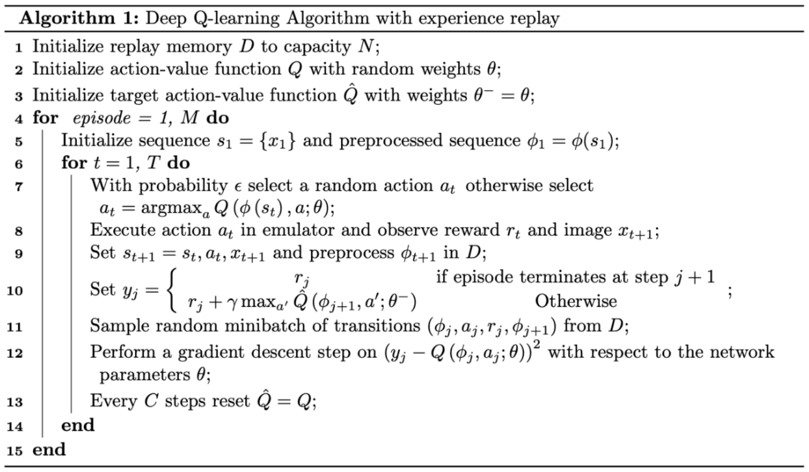
2.2.2 DDQN

Double Deep Q-Network (Double DQN) is a reinforcement learning algorithm that is an improved version based on the Deep Q-Network (DQN) to solve the over-estimation problem in DQN algorithms.

When choosing an action, the DQN network uses greedy policy to select the action with a maximum Q-value and the target Q-value is calculated based on this selected action. Though using such a greedy policy can help the network to converge towards the potential optimal target Q-values, it will also lead to over-estimation on the Q-values for that each time the model estimate a Q-value, it may over-estimate it a little bit and after thousands of iterations, the gap between the ground-truth Q-values and the estimated Q-values will grow wider and wider. Besides, the instability and noise of the neural network may lead to overestimation of the target Q value.

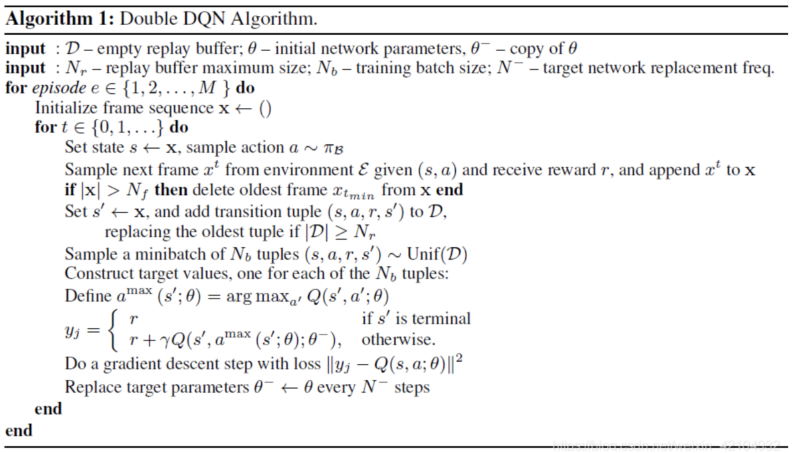
The core idea of Double DQN is to **use two neural networks to estimate the Q-values** of each action in the current state and the Q-value after selecting the optimal action, respectively, in order to reduce estimation bias and improve learning performance.

The pseudo code of the DQN algorithm is as shown in figure 4.



```
Algorithm 1: Deep Q-learning Algorithm with experience replay
Initialize replay memory $D$ to capacity $N$;
Initialize action-value function $Q$ with random weights $\theta$;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for episode = 1, M do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$ otherwise select
          $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$;
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$;
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1}$ in $D$;
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{Otherwise} \end{cases}$;
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$;
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network
          parameters $\theta$;
        Every $C$ steps reset $\hat{Q} = Q$;
    end
end
```

Figure 3: Pseudo code of DQN Algorithm



```
Algorithm 1: Double DQN Algorithm.
input : $\mathcal{D}$ – empty replay buffer; $\theta$ – initial network parameters, $\theta^-$ – copy of $\theta$
input : $N_r$ – replay buffer maximum size; $N_b$ – training batch size; $N^-$ – target network replacement freq.
for episode $e \in \{1, 2, \ldots, M\}$ do
    Initialize frame sequence $\mathbf{x} \leftarrow ()$
    for $t \in \{0, 1, \ldots\}$ do
        Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
        Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append $x^t$ to $\mathbf{x}$
        if $|\mathbf{x}| > N_f$ then delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ end
        Set $s' \leftarrow \mathbf{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$,
              replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
        Sample a minibatch of $N_b$ tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
        Construct target values, one for each of the $N_b$ tuples:
        Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
        $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
        Do a gradient descent step with loss $\| y_j - Q(s, a; \theta) \|^2$
        Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
    end
end
```

Figure 4: Pseudo code of Double DQN Algorithm

2.2.3 Dueling DQN

The Dueling Deep Q-Network (Dueling DQN) algorithm is an improvement over the Deep Q-Network (DQN) algorithm, which was proposed by DeepMind to address the instability and high variance of Q-learning in reinforcement learning.

The core idea of the Dueling DQN algorithm is to decompose the Q-value function into two parts, a state-value function and an advantage function.

- The state-value function represents the expected return in that state
- The advantage function represents the additional return that can be obtained by taking action a compared to other actions.

In this way, the algorithm can better estimate the Q-value of each action without increasing the computational cost, thereby improving learning efficiency and stability.

In the Dueling DQN algorithm, the network's output is divided into two parts: the state-value function $V(s)$ and the advantage function $A(s, a)$. Then, the Q-value of each action is calculated by combining them:

$$Q(s, a) = V(s) + A(s, a) - 1/|A| \cdot \sum A(s, a')$$

where $|A|$ represents the number of available actions, and $sum(A(s, a'))$ represents the average of the advantage function for all other actions in state s.

2.2.4 Rainbow

The Rainbow algorithm is a improved version of DQN and combines various reinforcement learning algorithms and techniques, including DQN, Double DQN, Dueling DQN, Prioritized Replay, Noisy Nets, n-step experience and Distributional RL, to achieve better performance and stability. The core idea of the Rainbow algorithm is to use multiple reinforcement learning techniques to solve different problems and combine them together for better performance. The following is a brief introduction to the various techniques used in the Rainbow algorithm:

1. Prioritized Replay: Prioritized Replay is an improved algorithm for experience replay that uses prioritized sampling of experience samples to improve learning efficiency. The Rainbow algorithm uses Prioritized Replay to improve learning efficiency.
2. N-step experience: N-step experience refers to updating a group of consecutive n time steps of experience samples together. Compared with traditional single-step experience, n-step experience can update the value function more quickly and improve sample utilization, thereby accelerating learning.
3. Noisy Nets: Noisy Nets is an improved neural network structure that introduces noise to improve exploration ability. The Rainbow algorithm uses Noisy Nets to improve exploration ability.
4. Distributional RL: Distributional RL is a distributed reinforcement learning algorithm used to learn the distribution of state values. The Rainbow algorithm uses Distributional RL to improve estimation accuracy.

Here I will give a further introduction to the Prioritized Replaybuffer and Noisy Layer techniques.

Prioritized Replaybuffer Priority Replay Buffer is an improved technique based on the traditional Replay Buffer. The main idea is to prioritize the experiences stored in the Replay Buffer so that important experiences can be more focused on during training. The implementation of Priority Replay Buffer is as follows:

- Define priorities: Each experience is assigned a priority, which can be calculated based on the reward value.
- Store experiences: Store experiences in the Replay Buffer according to their priorities. Binary trees or other data structures can be used for fast priority sorting.

- Sample experiences: During training, randomly sample a certain number of experiences from the Replay Buffer according to their priorities.
- Calculate importance weights: To ensure the statistical validity of the samples, importance weights need to be calculated for the sampled experiences. The IS (Importance Sampling) technique is used to calculate the ratio of the probability of each experience being sampled to its priority in the Replay Buffer.
- Update the model: Update the model based on the sampled experiences and the calculated importance weights.

The advantage of Priority Replay Buffer is that it can better utilize the experiences stored in the Replay Buffer, improving training efficiency and stability.

Noisy Layer Noisy Net is a technique used to increase exploration capability, where random noise is added to the weights of a neural network to introduce uncertainty into the output for small changes in input space, making it easier for the agent to explore unknown state and action spaces.

The implementation of Noisy Net is as follows:

- Define noise: Introduce random noise to each weight in the neural network to make the output of the neural network uncertain for small changes in the input space.
- Parameterize noise: Use trainable parameters to control the size and distribution shape of the noise for each weight. These parameters can be updated through back-propagation.
- Forward propagation: During forward propagation, add the noise to the weights to obtain the noisy weights, and then use them to calculate the output of the neural network.
- Back-propagation: During back-propagation, add the gradient of the noise to the gradient of the weights to update the noise parameters.

2.3 Policy-based algorithms

Policy-based algorithms use policy functions to directly obtain the optimal action, focusing on how to optimize the policy function directly, typically using methods such as gradient descent to update the policy parameters. Policy-based reinforcement learning algorithms are commonly used for tasks in continuous action and state spaces, such as robot control. In these tasks, the policy function can be directly represented as a neural network. The advantage of policy-based algorithms is their ability to handle continuous action and state spaces, but they may be difficult to converge and less stable.

The foundation of policy-based algorithms is the **Policy Gradient[6]** algorithm, which directly optimizes the policy function to maximize the long-term reward. Other policy-based algorithms are mostly based on Policy Gradient algorithm for improvement. **Actor-Critic**[7] algorithm combines value function and policy function, using actor and critic networks to represent the policy function and value function respectively, and updating the actor's parameters through gradient ascent. **Deep Deterministic Policy Gradient (DDPG)**[8] algorithm improves on the Policy Gradient algorithm by using a deep neural network to represent the policy function and value function, and using techniques such as experience replay and target networks to improve learning efficiency and stability. **Proximal Policy Optimization (PPO)**[9] algorithm is also a policy gradient-based reinforcement learning algorithm, using a proximal policy optimization method to update the policy function's parameters to ensure that the update magnitude is not too large. **Soft Actor-Critic (SAC)**[10] algorithm is based on the maximum entropy theory, achieving a balance between exploration and exploitation by maximizing the policy's entropy.

In the following sections, I will give introduction to **DDPG, PPO and SAC**.

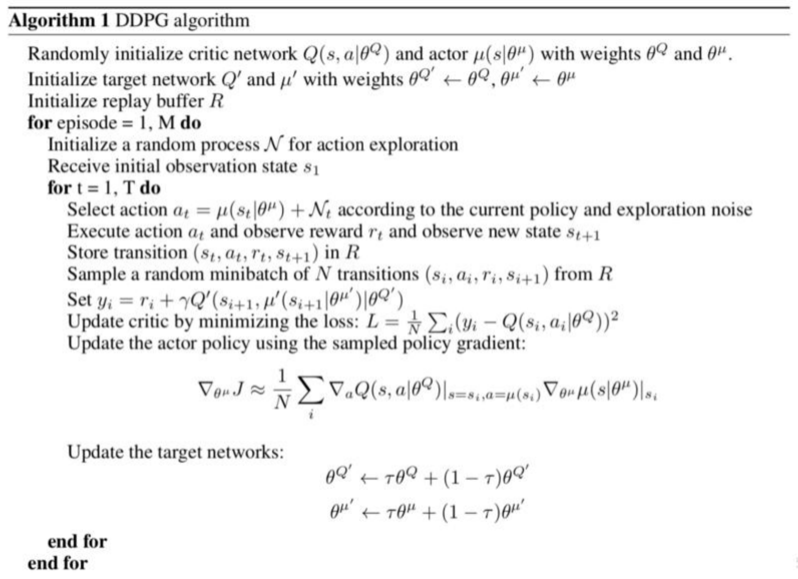
2.3.1 DDPG

The DDPG algorithm is a deep reinforcement learning algorithm used to solve continuous control problems by combining deep neural networks and deterministic policy gradient algorithms. The algorithm employs an Actor-Critic structure, in which the Actor network generates continuous actions and the Critic network estimates the action-value and state-value functions.

The Actor and Critic networks in the DDPG algorithm are both deep neural networks. The Actor network approximates the policy function by taking the current state as input and outputting continuous actions. The Critic network approximates the action-value and state-value functions by taking the current state and action as input and outputting corresponding Q-values. Additionally, the Critic network requires the actions generated by the Actor network to calculate the target Q-values for policy gradient training.

The DDPG algorithm also employs experience replay to improve sample utilization and stability. Specifically, the algorithm stores experience samples in a replay buffer and randomly samples a batch of experiences from the buffer for training. This method avoids correlations between samples and improves sample utilization and stability.

The pseudo code of the DDPG algorithm is as shown in figure 5.



```
Algorithm 1 DDPG algorithm
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

Figure 5: Pseudo code of DDPG Algorithm

2.3.2 PPO

PPO (Proximal Policy Optimization) is a policy gradient-based reinforcement learning algorithm that solves the reinforcement learning problem by optimizing the policy function. The primary focus of PPO is to address the stability issue of policy optimization, which enhances training stability while maintaining performance.

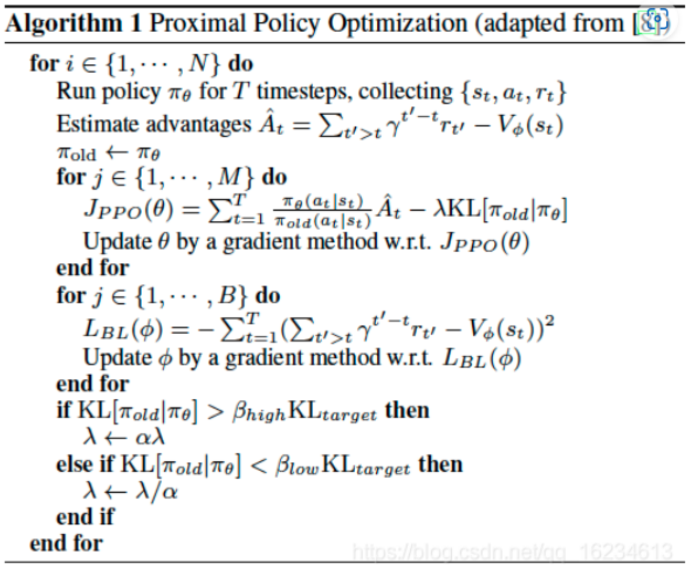
The core idea of the PPO algorithm is to limit the difference between the new and old policies during each policy update, thereby ensuring stability in updates. Specifically, the PPO algorithm uses a method called ”proximal clipping” to restrict the size of policy updates, ensuring that the new policy does not deviate too far from the old policy. The proximal clipping method controls the size of policy updates by limiting the KL divergence between the new and old policies, thereby guaranteeing stability in updates.

The PPO algorithm also utilizes an Actor-Critic structure. It is an off-policy algorithm that uses sample experience to update the policy and value function. Additionally, the PPO algorithm employs a method called Generalized Advantage Estimation (GAE) to estimate the state value function, further improving training efficiency and stability.

The pseudo code of the PPO algorithm is as shown in figure 6.

Here I will give a further detailed introduction about the ”proximal clipping” technique in PPO.

Proximal clipping Specifically, the PPO algorithm can be defined as maximizing the following objective function:



Algorithm 1 Proximal Policy Optimization (adapted from [8])

```
for $i \in \{1, \cdots, N\}$ do
    Run policy $\pi_\theta$ for $T$ timesteps, collecting $\{s_t, a_t, r_t\}$
    Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
    $\pi_{\text{old}} \leftarrow \pi_\theta$
    for $j \in \{1, \cdots, M\}$ do
        $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \mathrm{KL}[\pi_{old}|\pi_\theta]$
        Update $\theta$ by a gradient method w.r.t. $J_{PPO}(\theta)$
    end for
    for $j \in \{1, \cdots, B\}$ do
        $L_{BL}(\phi) = -\sum_{t=1}^{T} (\sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t))^2$
        Update $\phi$ by a gradient method w.r.t. $L_{BL}(\phi)$
    end for
    if $\mathrm{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \mathrm{KL}_{target}$ then
        $\lambda \leftarrow \alpha\lambda$
    else if $\mathrm{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \mathrm{KL}_{target}$ then
        $\lambda \leftarrow \lambda/\alpha$
    end if
end for
```

Figure 6: Pseudo code of PPO Algorithm

$$L(\theta) = \hat{E}t\left[\min\left(\frac{\pi\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}A_t, \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)A_t\right)\right]$$

where θ represents the parameters of the current policy, θ_{old} represents the parameters of the previous policy, $\pi_\theta(a_t|s_t)$ represents the probability of selecting action a_t in state s_t, A_t represents the advantage function at time t, clip(x, a, b) represents a function that clips x to the range $[a, b]$, and ϵ is a small hyper-parameter.

The objective function can be divided into two parts.

- The first part is:

  $$\hat{E}t\left[\frac{\pi\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}A_t\right]$$

  which can be seen as using the KL divergence to measure the difference between the new policy and the old policy, and multiplying it by the advantage function.

- The second part is:

  $$\hat{E}t\left[\text{clip}\left(\frac{\pi\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)A_t\right]$$

  which can be seen as using proximal clipping to restrict the magnitude of the policy update. Specifically, the ratio r_t of $\pi_\theta(a_t|s_t)$ to $\pi_{\theta_{\text{old}}}(a_t|s_t)$ is clipped to the range $[1-\epsilon, 1+\epsilon]$. Finally, the clipped ratio r_t is multiplied by the advantage function A_t to contribute to the second part.

2.3.3 SAC

SAC (Soft Actor-Critic) is a policy gradient-based reinforcement learning algorithm that utilizes maximum entropy policy to address the exploration-exploitation dilemma in reinforcement learning, while also employing offline data replay to enhance sample efficiency.

The core idea of SAC algorithm is to maximize the entropy of the policy, i.e., the uncertainty of the policy itself, to promote exploration. Specifically, SAC algorithm maximizes the sum of the entropy of the policy and the expected return, where entropy is defined as the negative logarithm of the probability distribution of actions generated by the policy. This way, SAC algorithm can learn both a deterministic policy and a stochastic policy, thereby improving exploration performance.

$$\max_{\theta,\phi} E\mathcal{D}\left[\sum t = 0^\infty \gamma^t r_t\right] - \alpha\mathcal{H}(\pi_\theta) + \beta\mathcal{D}\mathrm{KL}(\pi\theta \parallel \pi_{\text{target}})$$

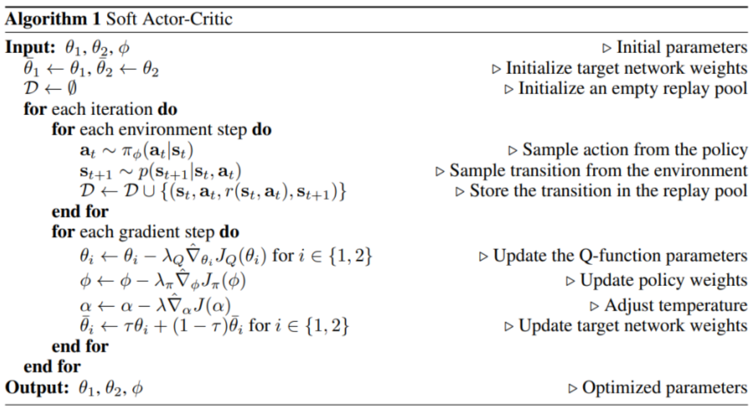
where θ and ϕ denote the parameters of the policy and Q-function, respectively, $\mathcal{D}$ denotes the experience replay buffer, (s, a, r, s') denotes the experience data, γ denotes the discount factor, r_t denotes the reward at time t, α and β denote the weights of the policy entropy and KL divergence, respectively, $\mathcal{H}(\pi_\theta)$ denotes the entropy of the policy π_θ, π_{target} denotes the target policy computed based on the Q-function and normalized to satisfy the properties of a probability distribution, and $\mathcal{D}\text{KL}(\pi\theta \parallel \pi_{\text{target}})$ denotes the KL divergence between the policy π_θ and the target policy π_{target}.

SAC algorithm's objective function consists of three parts.

- The first part is the expected return.
- The second part is the entropy of the policy which is used to measure the diversity of the policy and facilitate exploration.
- The third part is the KL divergence between the policy and the target policy, which is used to control the magnitude of policy updates to prevent large policy changes that may lead to performance degradation.

Another key point of the SAC algorithm is to use two Q-functions to estimate the state-action value function, instead of using the traditional single Q-function. This approach can reduce estimation errors and thereby improve learning efficiency and stability. In the SAC algorithm, a target entropy is used to balance the trade-off between exploration and exploitation, and an adaptive algorithm is used to adjust the value of the target entropy to maintain the exploration performance of the policy.

The pseudo code of the SAC algorithm is as shown in figure 7.



```
**Algorithm 1** Soft Actor-Critic
**Input:** $\theta_1, \theta_2, \phi$                                              ▷ Initial parameters
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$                        ▷ Initialize target network weights
  $\mathcal{D} \leftarrow \emptyset$                                           ▷ Initialize an empty replay pool
  **for** each iteration **do**
    **for** each environment step **do**
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$                          ▷ Sample action from the policy
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$                ▷ Sample transition from the environment
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$   ▷ Store the transition in the replay pool
    **end for**
    **for** each gradient step **do**
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$   ▷ Update the Q-function parameters
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$                  ▷ Update policy weights
      $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$                     ▷ Adjust temperature
      $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$      ▷ Update target network weights
    **end for**
  **end for**
**Output:** $\theta_1, \theta_2, \phi$                                             ▷ Optimized parameters
```

Figure 7: Pseudo code of SAC Algorithm

3 Experimental Result

In this section, I will present the experimental results of using value-based algorithms in the Atari environment and policy-based algorithms in the Mujoco environment.

3.1 Value-based algorithm

I tested six value-based algorithms, including DQN, DDQN, Dueling DQN, Dueling DQN with Priority Replaybuffer, Dueling DQN with Priority Replaybuffer and Noisylayer, and Rainbow, in the 'VideoPinball-ramNoFrameskip-v4' environment, and the experimental results are shown below in Figure 8 . It should be noted that the following figure represents the average reward (averaged over the last 50 episodes), which allows for smoother curves and easier identification of the performance changes of the model.

The reward curves and the average reward curves (averaged over the last 50 episodes) of each algorithm are shown in Figure 9 and Figure 10, respectively.

From the result, we can see that:

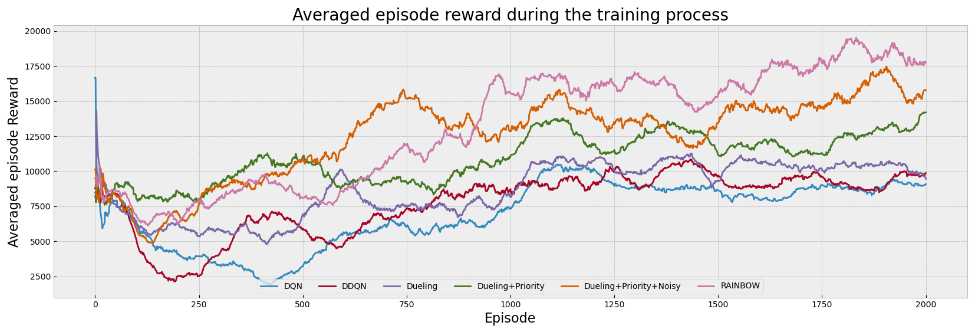
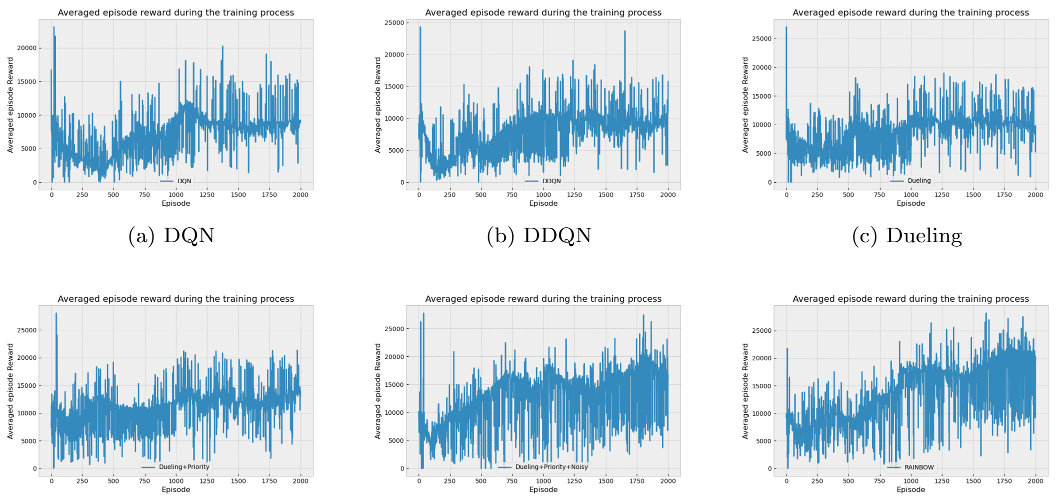


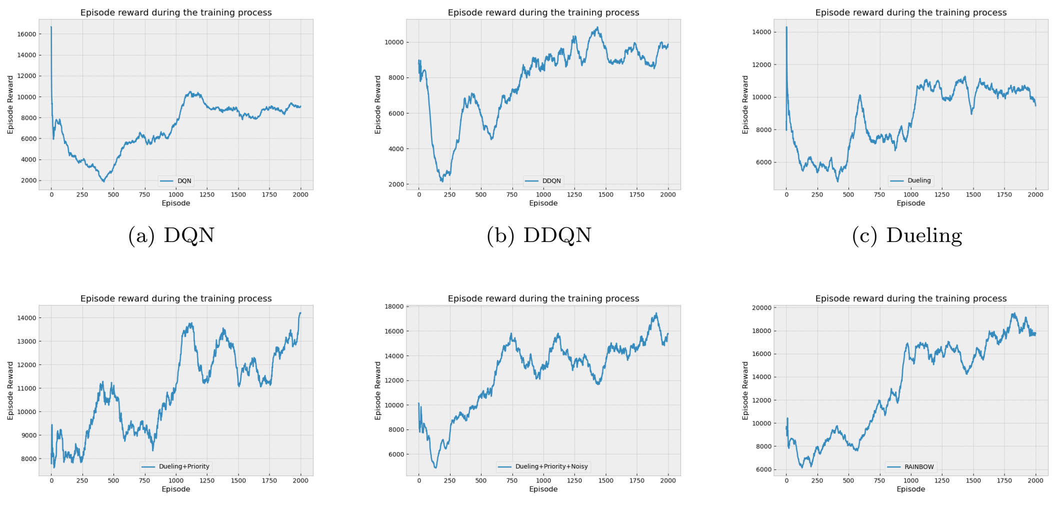
Figure 8: Performance of different value-based algorithms on 'VideoPinball-ramNoFrameskip-v4'



(a) DQN (b) DDQN (c) Dueling

(d) Dueling+Priority (e) Dueling+Priority+Noisy (f) RAINBOW

Figure 9: Reward of different value-based algorithms on 'VideoPinball-ramNoFrameskip-v4'



(a) DQN (b) DDQN (c) Dueling

(d) Dueling+Priority (e) Dueling+Priority+Noisy (f) RAINBOW

Figure 10: Average reward of different value-based algorithms on 'VideoPinball-ramNoFrameskip-v4'

1. The performance of the Rainbow algorithm is the best, and the Dueling DQN with Priority Replaybuffer and Noisylayer also show good performance. The performance of the DQN, DDQN, and Dueling DQN algorithms is similar and relatively poorer.

2. At the beginning of the model training, the performance of all algorithms is initialized at a relatively high level. This is because we explore using a random policy before training starts and pre-fill the experience into the replay buffer (default: 50,000 step). Moreover, during the epsilon-greedy exploration phase at the beginning of training, the value of epsilon is relatively large and tends to favor random exploration. The average reward of random exploration is around 8,000-10,000, and there is a possibility of obtaining a very high reward (greater than 30,000).

3. All algorithms show a performance drop at the beginning of the training phase, followed by gradual improvement. This may be due to the inappropriate initialization of the model, which may lead to the adoption of specific but unsuitable actions that cause the algorithm's performance to be lower than that of random action.

4. The DQN, DDQN, and Dueling DQN algorithms perform poorly in this environment, and their performance is only equivalent to that of random action after convergence. This may be due to the high complexity of the environment and the use of the ram format to represent a complex state with only 128 dimensions, which compresses the information and makes it difficult for the basic DQN algorithm to learn effective policies.

5. The methods of Priority Replaybuffer and Noisy Layer can effectively improve the performance of the algorithm. It can be observed that the three algorithms that use Priority Replaybuffer are more stable, and there is no obvious performance drop at the beginning of the algorithm compared to those that do not use it. This may be because Priority Replaybuffer effectively selects more appropriate training samples.

6. Although the Rainbow algorithm has the best performance, its training speed is relatively slow and requires a considerable number of episodes to achieve good performance. In contrast, DQN, DDQN, and other algorithms have significantly higher convergence speeds than complex algorithms, which is in line with our expectations.

7. Although all algorithms show relatively good performance in the later stages of training, their rewards still have a large variance and strong fluctuations, which may be due to the complexity of the environment.

3.2 Policy-based algorithm

I tested three policy-based algorithms, including DDPG, PPO and SAC, in the 'Ant-v2' environment, and the experimental results are shown below in Figure 11. It should be noted that the following figure represents the average reward (averaged over the last 20 episodes), which allows for smoother curves and easier identification of the performance changes of the model. Besides, in this environment, once the model has learned enough knowledge, the Ant in the environment can keep running without stopping (done=False) for a long time. Therefore, a maximum step limit of 5000 is set for a single episode by default.

The reward curves and the average reward curves (averaged over the last 50 episodes) of each algorithm are shown in Figure 12 and Figure 13, respectively.

From the result, we can see that:

1. On this task, the SAC algorithm performs the best, followed by PPO, while the DDPG algorithm performs the worst. The DDPG algorithm may not be suitable for this task, as the model's performance is still poor in the later stages of training, and it has not learned much useful knowledge. In contrast, the model performance of the SAC and PPO algorithms improves gradually with training.

2. In the early stages of training, the reward of a single episode is usually negative. However, when the model has trained to a certain degree, negative rewards are almost non-existent, which can be used as an indicator of algorithm effectiveness.

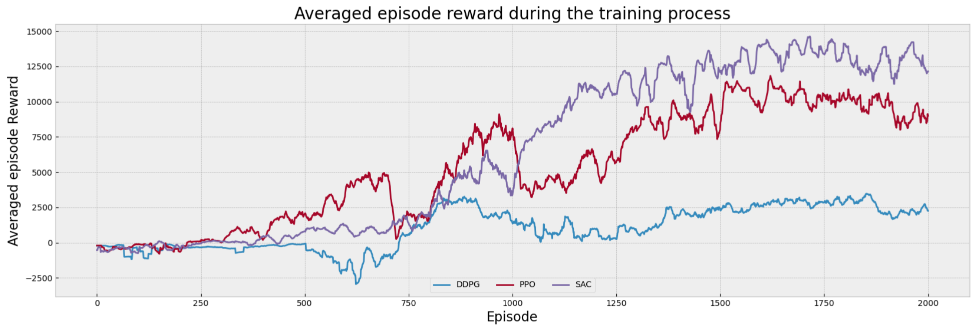
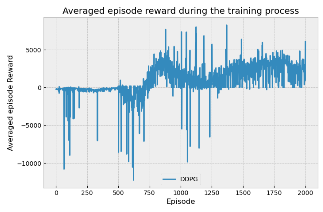
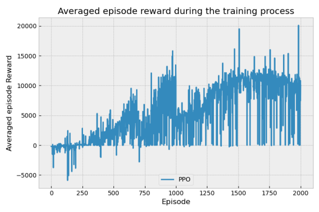


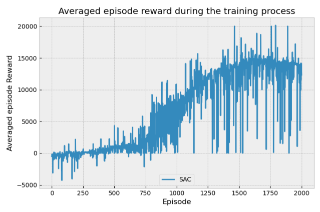
Figure 11: Performance of different policy-based algorithms on 'Ant-v2'



(a) DDPG

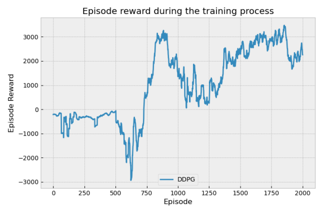


(b) PPO

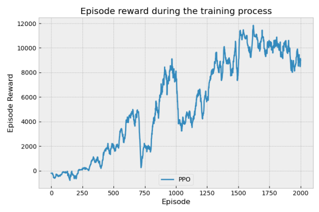


(c) SAC

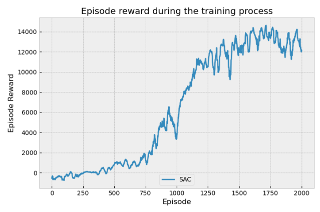
Figure 12: Reward of different policy-based algorithms on 'Ant-v2'



(a) DDPG



(b) PPO



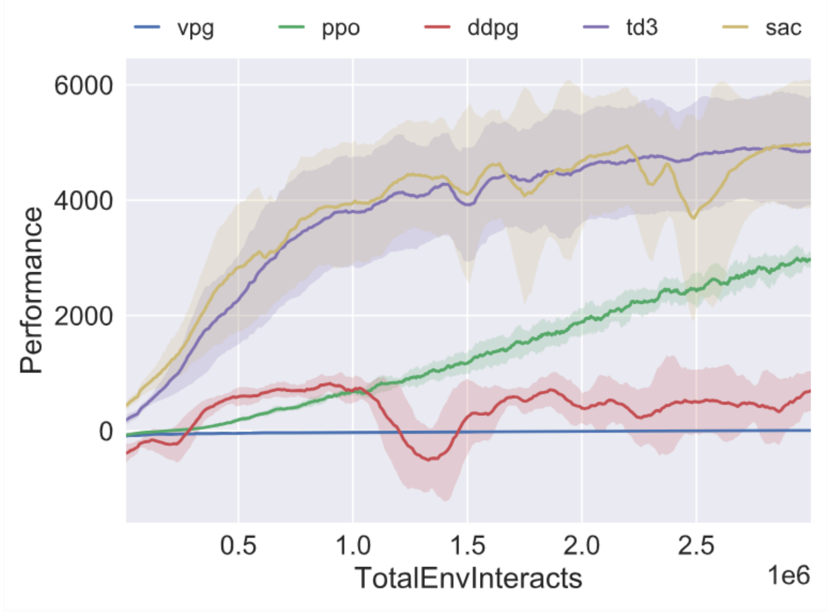
(c) SAC

Figure 13: Average reward of different policy-based algorithms on 'Ant-v2'

3. Compared to the previous Atari environment, the Ant-v2 environment is more stable. When the algorithm is trained to the later stages (especially SAC), the variance of the reward is relatively small, concentrated, and extreme values are less likely to occur.

4. The training speed of the SAC algorithm is slower than that of PPO and DDPG, while the training speed of PPO is slower than that of DDPG. The slow training speed of the SAC algorithm may be due to its pursuit of policy entropy maximization, which tends to do more thorough exploration and thus converges more slowly. The slow training speed of the PPO algorithm may be due to its proximal clipping design, which limits the update speed of the policy, making the algorithm more stable but also limiting its convergence speed to some extent.

In fact, the DDPG algorithm is not suitable for the Ant-v2 environment. I obtained benchmark data for the mujoco environment from the OpenAI website, and the results for the Ant-v2 environment during a 1000-step test are shown below in Figure 14 , indicating that the performance of the DDPG algorithm is indeed poor.



*3M timestep benchmark for Ant-v3 using **PyTorch** implementations.*

Figure 14: Benchmark performance of DDPG on 'Ant-v2' given by OpenAI

4 Conclusion

During the process of this project, I first systematically learned the relevant knowledge of two kinds of deep reinforcement learning, actor-critic model based algorithms, A3C and DDPG, on the basis of the course content, and then implemented the two algorithms on my own to better understanding the implementation details. Furthermore, I conducted several experiments on the parameters of the two algorithms and did some analysis based on the experimental results.

During the completion of this project, I first systematically learned the relevant knowledge of the model-free deep reinforcement learning algorithms, on the basis of the course content. Then I implemented various value-based and policy-based deep reinforcement learning algorithms and conducted experiments on two environments. In fact, my implementation can be applied to different environments, and I did conduct experiments. However, due to the long training time, I did not complete the training for the sake of time cost and cannot show the experimental results here.

If given the opportunity, I will further improve my code, enhance its readability, and implement more deep reinforcement learning algorithms to test them on various environments.

References

[1] Christopher Watkins. Learning from delayed rewards. 01 1989.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

[5] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

[6] Philip S. Thomas and Emma Brunskill. Policy gradient methods for reinforcement learning with function approximation and action-dependent baselines, 2017.

[7] Vijay Konda and Vijaymohan Gao. Actor-critic algorithms. 01 2000.

[8] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.