Reinforcement Learning Final Project
DQN on Atari & DDPG on MuJoCo

Jiale Yuan

521370910130

Abstract

This report presents the introduction to two reinforcement learning algorithms, Deep Q-Network (DQN) and Deep Deterministic Policy Gradient (DDPG), the implementation details of them, and the experimental results on two Atari games (Pong and Boxing) and two MuJoCo environments (HalfCheetah and Ant). As this is the term project of the reinforcement learning course, I not only complete the basic requirements of the project, but also try my best to improve the performance by modifying the original algorithms, test them on more than one environment, and analyze the experimental results in detail. The code and results are available on my GitHub repository: https://github.com/Geral-Yuan/RL_Project.

1 Introduction

Atari games and MuJoCo environments are two popular benchmarks in reinforcement learning. In this term project, I implemented two classical reinforcement learning algorithms: Deep Q-Network (DQN) for Atari games and Deep Deterministic Policy Gradient (DDPG) for MuJoCo environments. Then, I conducted experiments on 2 Atari games (Pong and Boxing) and 2 MuJoCo environments (HalfCheetah and Ant) to evaluate the performance of these algorithms. To achieve a satisfactory performance, I made some modifications to the original algorithms and tuned hyperparameters to improve the training stability and efficiency. Finally, I presented the experimental results of the two algorithms on the four environments, analyze the performance of the algorithms, and investigate the impacts of my modifications to DDPG.

The rest of this report is organized as follows. In Section 2, I introduce the DQN and DDPG algorithms, including their main components and the modifications I made to improve DDPG. In Section 3, I present the experimental setup, including the environments, implementation details, and hyperparameters. And then I show the results of the experiments, including training curves, test results, and the ablation study on the impacts of my modifications to DDPG. Finally, I conclude the report in Section 4, summarizing what I have done, obtained, and learned in this term project.

2 Algorithms

2.1 DQN

DQN is a value-based reinforcement learning algorithm that uses a neural network to approximate the action-value function to optimize the policy of the agent. The framework of

DQN is similar to typical Q-learning. The main difference between DQN and classical Q-learning includes following three aspects:

1. DQN uses a neural network to approximate the action-value function, which allows it to handle high-dimensional or continuous state spaces efficiently.
2. DQN uses experience replay to store the agent's experiences and sample a batch of experiences to update the neural network. This helps to break the correlation between consecutive experiences and stabilize the training process.
3. DQN uses a target network to stabilize the training process. The target network is a copy of the main network, and its weights are updated less frequently. This helps to reduce the variance of the Q-value estimates and improve the convergence of the algorithm.

The pseudo code of DQN is shown as follows.

DQN(γ, ε, E, B, C):
Initialize parameter w of the neural network $Q_w(s, a)$
Initialize the target network $Q_{w'}(s, a)$ with weight $w' \leftarrow w$
Initialize replay memory R
for $e = 1 \rightarrow E$:
    Initialize state s
    while s is **not** terminal:
        Sample action a from ε-greedy policy $\pi(a|s)$
        Take action a, observe reward r and next state s'
        Store transition (s, a, r, s') into replay memory R
        $s \leftarrow s'$
        if R contains enough samples:
            Sample a batch of transitions $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1,\ldots,B}$
            Compute the target $y_i = r_i + \gamma \max_a Q_{w'}(s_{i+1}, a)$
            Update Q_w by minimizing$L(w) = \frac{1}{B}\sum_i (y_i - Q_w(s_i, a_i))^2$
            Update the target network $Q_{w'}$ every C steps by $w' \leftarrow w$
return Q_w

2.2 DDPG

DDPG is an actor-critic algorithm that uses two neural networks: an actor network to represent the policy and a critic network to represent the action-value function. DDPG is designed for continuous action spaces and can handle high-dimensional state spaces efficiently. The main components of DDPG include:

1. Actor Network: The actor network takes the state as input and outputs the action to be taken by the agent. The actor network is trained to maximize the expected return by using the critic network to evaluate the action.

2. Critic Network: The critic network takes the state and action as input and outputs the action-value function $Q(s, a)$, which estimates the expected return of taking action a in state s.
3. Experience Replay: DDPG uses experience replay to store the agent's experiences and sample a batch of experiences to update the actor and critic networks. This helps to break the correlation between consecutive experiences and stabilize the training process.
4. Target Networks: DDPG uses target networks for both the actor and critic networks to stabilize the training process. The target networks are updated less frequently than the main networks, which helps to reduce the variance of the Q-value estimates and improve the convergence of the algorithm.

DDPG(γ, τ, E, B):
Initialize actor network $\pi_\theta(s)$ and critic network $Q_w(s, a)$
Initialze target networks $\pi_{\theta'}(s)$ and $Q_{w'}(s, a)$ with weights $\theta' \leftarrow \theta$ and $w' \leftarrow w$
Initialize replay memory R
for $e = 1 \rightarrow E$:
    Initialize a random process $\mathcal{N}$ for action exploration
    Initialize state s
    while s is **not** terminal:
        Sample action $a = \pi_\theta(s) + \mathcal{N}$
        Take action a, observe reward r and next state s'
        Store transition (s, a, r, s') into replay memory R
        $s \leftarrow s'$
        if R contains enough samples:
            Sample a batch of transitions $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1,\ldots,B}$
            Compute the target $y_i = r_i + \gamma Q_{w'}(s_{i+1}, \pi_{\theta'}(s_{i+1}))$
            Update Q_w by minimizing $L(w) = \frac{1}{B}\sum_i (y_i - Q_w(s_i, a_i))^2$
            Update π_θ by maximizing $J(\theta) = \frac{1}{B}\sum_i Q_w(s_i, \pi_\theta(s_i))$
            Update target networks: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$, $w' \leftarrow \tau w + (1-\tau)w'$
return π_θ and Q_w

2.3 Improving modifications to DDPG

DDPG is a powerful algorithm, but it can be unstable and sensitive to hyperparameters, especially when tackling complex environments with high-dimensional state and action spaces (e.g., MuJoCo environments). To improve its stability and performance, I search online and refer to a well-known work, TD3 [1], and implement the following modifications:

Delayed Policy Updates. In vanilla DDPG, both the actor and critic networks are updated at every step. However, the frequent updates of the actor network can lead to instability, especially when the critic network is not yet well-trained. To address this issue, I update the actor network each d steps and keep the critic network updated at every step. This makes the

training process more stable by allowing the critic network to catch up with the actor network before the next update.

Target Policy Smoothing. In DDPG, the target policy's output can be overly deterministic, which may lead to overfitting and instability. To mitigate this issue, I add noise to the target action during the update step. Specifically, I use clipped Gaussian noise to perturb the target action, which helps to smooth the target policy and improve robustness.

Delayed Target Updates. Original DDPG updates the target networks at every step, which makes the training process unstable. So, I try to update the target networks less frequently (each C steps) to reduce the variance of the Q-value estimates and improve the convergence of the algorithm. This is similar to the target network update strategy in DQN.

3 Experiments

3.1 Environments

The experiments are conducted on two Atari games (Pong and Boxing) and two MuJoCo environments (HalfCheetah and Ant). The renderings of these environments are shown in Figure 1 and Figure 2. The Atari games have discrete action spaces, so I test my DQN implementation on them. In contrast, the MuJoCo environments have continuous action spaces, which is suitable for DDPG.

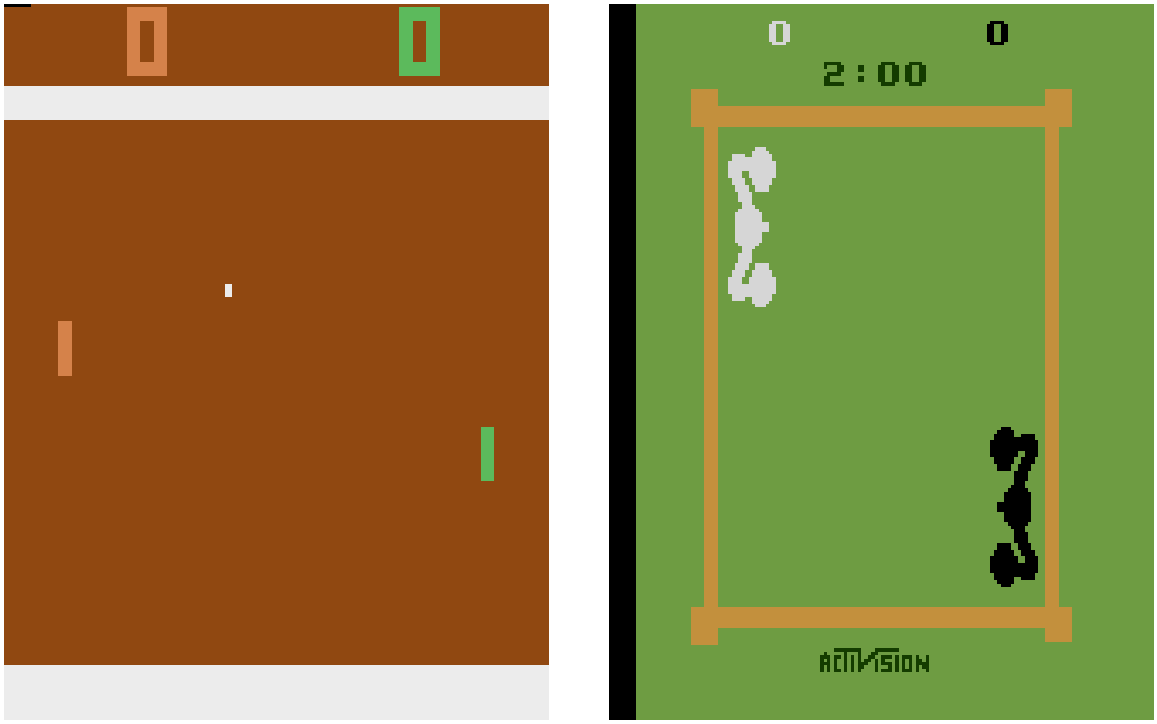


Figure 1: Pong (left) and Boxing (right), two Atari environments used to evaluate DQN

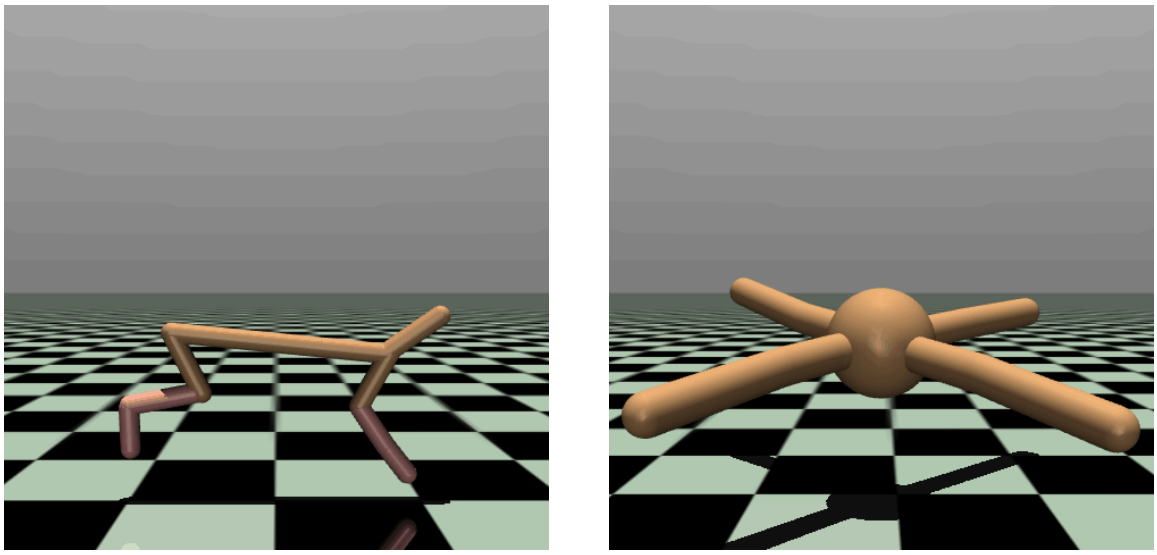


Figure 2: HalfCheetah (left) and Ant (right), two MuJoCo environments used to evaluate DDPG

3.2 Implementation & Hyperparameters

3.2.1 DQN

I use a classic pipeline to handle Atari games with DQN. First, I preprocess the OpenAI Gym environment using the `AtariPreprocessing` function. This function converts the Atari environment's images to grayscale and resizes them to 84x84 pixels, which helps to reduce the dimensionality of the state space. Additionally, it sets the same action to be repeated for 4 consecutive frames, stacking these 4 frames into a 4-channel image as input for the subsequent model. Therefore, I design the DQN model consists of a convolutional neural network (CNN) that takes 4x84x84 images as input. As shown in Figure 3, the CNN has three convolutional layers with ReLU activation functions, followed by a fully connected layer that outputs the Q-values for each action.

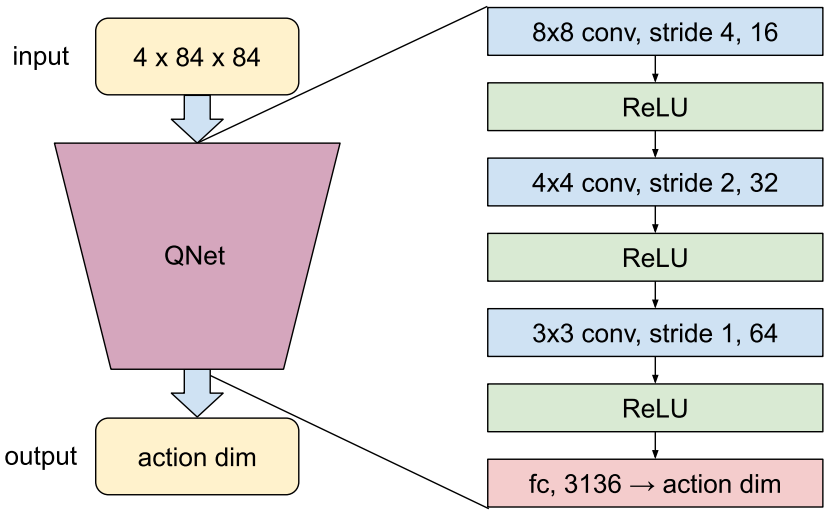


Figure 3: The architecture of the DQN model used for Atari games

After implementing the DQN model and the training pipeline, I train the model on the Pong and Boxing environments. With tens of trials, I tune the hyperparameters to achieve a feasible performance. The hyperparameters used in the experiments are summarized in Table 1.

Table 1: Hyperparameters details of DQN training for Pong and Boxing environments

Hyperparameter	Value for Pong	Value for Boxing	Description
γ	0.98	0.98	reward discount factor
lr	1e-4	1e-5	QNet learning rate
B	32	32	size of minibatch
C	25	25	update cycle of target Net
ε start	1.0	1.0	initial exploration noise
ε end	0.01	0.01	final exploration noise
ε decay	0.99	0.99	noise decay rate per episode
E	4000	10000	number of training episodes

3.2.2 DDPG

For the DDPG algorithm, the model architecture is relatively simple. As shown in Figure 4, I use two MLPs with two hidden layers of size 256 as the actor and critic networks. The activation function used in both networks is ReLU. The actor network takes the state as input and outputs the action, while the critic network takes both the state and action as input and outputs the Q-value.

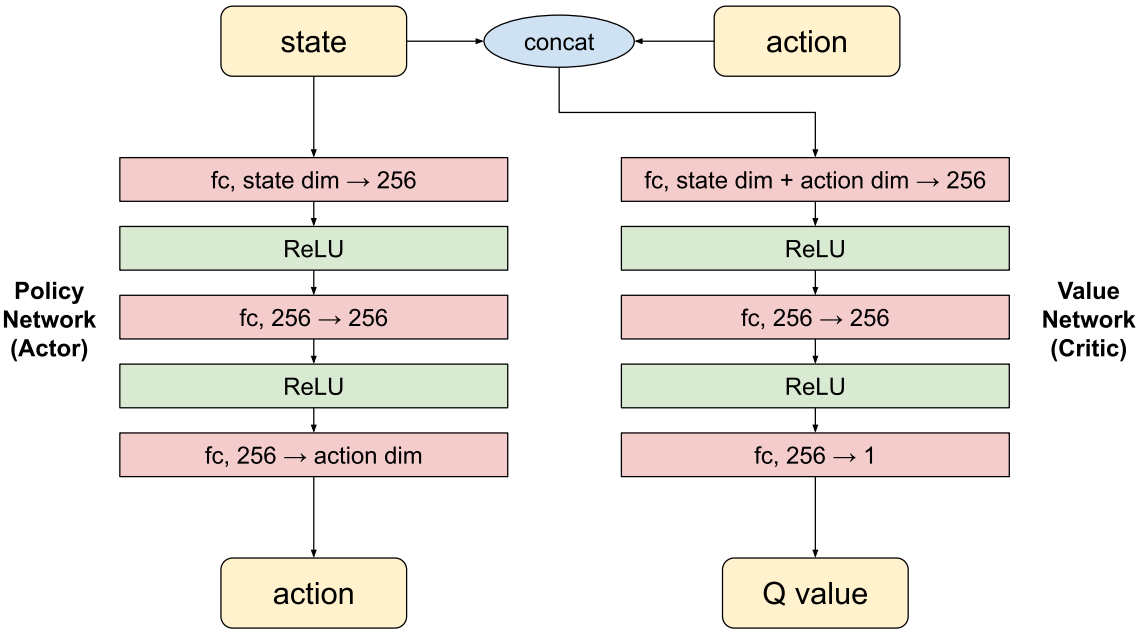


Figure 4: The architecture of the DDPG model used for MuJoCo environments

Although the network architecture of DDPG is simple, the training process is more complex than DQN, especially after I add the modifications mentioned above. I implement the DDPG algorithms with my improving modifications and tuning the hyperparameters when training the model on the HalfCheetah and Ant environments. The final hyperparameters is listed in Table 2, in which $\gamma, \text{actor lr}, \text{critic lr}, B, \sigma, \tau, E$ are from the original DDPG algorithm, while σ_p, C_p, C_t are the hyperparameters I added to improve the training stability and efficiency. While C_p, C_t are easy to understand (update policy network and target networks every C_p and C_t steps, respectively), σ_p is the scale of the Gaussian noise added to the target action during the update step, which helps to smooth the target policy and improve robustness.

Table 2: Hyperparameters details of DDPG training for HalfCheetah and Ant environments

Hyperparameter	Value for HalfCheetah	Value for Ant	Description
γ	0.98	0.98	reward discount factor
actor lr	1e-4	1e-4	Policy Net learning rate
critic lr	1e-4	1e-4	Value Net learning rate
B	64	64	size of minibatch
σ	0.1	0.1	exploration noise scale
σ_p	0.2	0.2	policy noise scale
τ	0.01	0.01	target network update rate
C_p	5	5	update cycle of policy Net
C_t	5	5	update cycle of target Nets
E	5000	5000	number of training episodes

3.3 Results & Analysis

3.3.1 Training Results

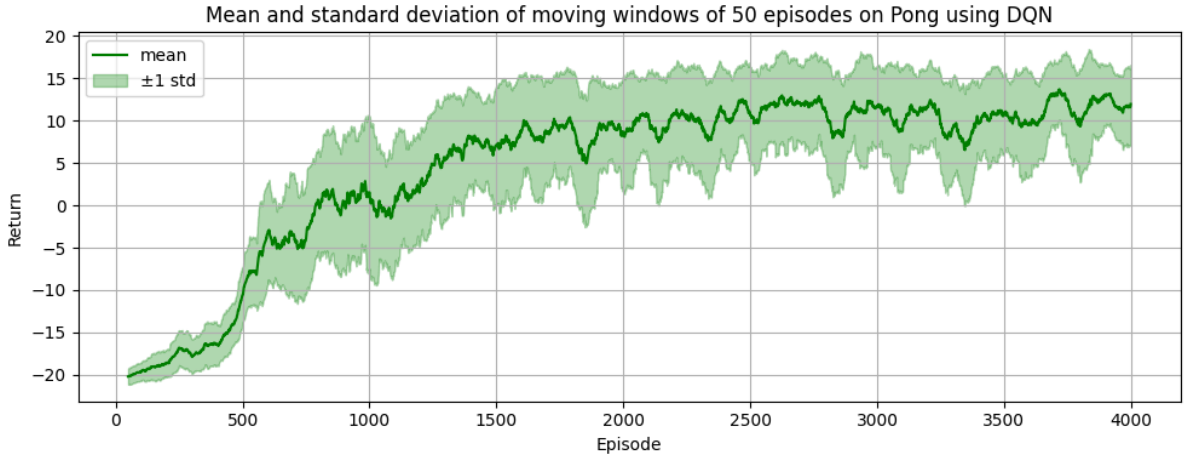
First, I present the training results of DQN on Pong and Boxing as well as DDPG on HalfCheetah and Ant. The training curves of 4 experiments are shown in Figure 5, which plots the average reward over the last 50 episodes at each step and shows the standard deviation as a shaded area. And the maximum, minimum, mean, and standard deviation of the returns of the last 500 training episodes are summarized in Table 3.

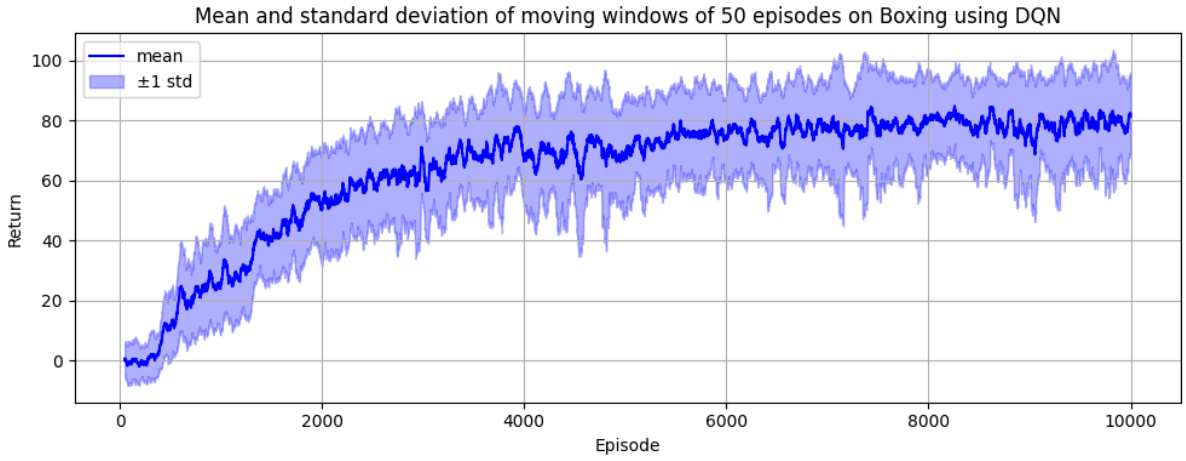
We can see that DQN converges stably and achieve satisfactory performance on both Pong and Boxing, although their final average returns are lower than the optimal performance (21 for Pong and 100 for Boxing). The training curves show that DQN learns to play Pong and Boxing effectively, with the average returns increasing over time. The standard deviation of the returns is relatively small, indicating that the training process is stable.

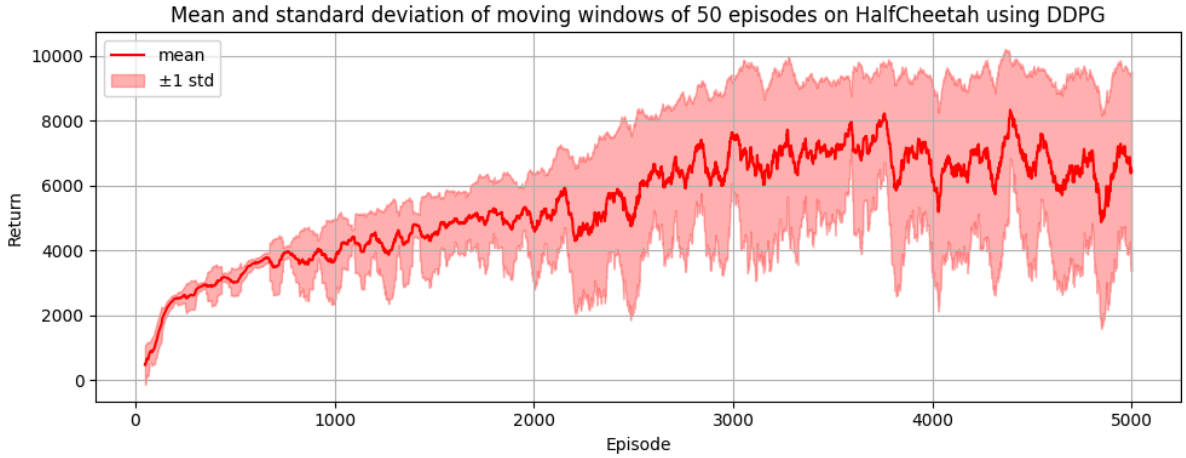
Meanwhile, DDPG also achieves feasible performance on HalfCheetah and Ant, making the agent run and jump fast in two environments, which can be verified by reviewing the rendered output. However, the training curves and the statistics in Table 3 show that training process of DDPG seems unstable and the final average returns are not high enough, far from the optimal performance (about 11000 for HalfCheetah and 5500 for Ant).

Table 3: The maximum, minimum, mean, and standard deviation of the returns of the last 500 training episodes of DQN on Pong and Boxing and DDPG on HalfCheetah and Ant.

Algorithm	Environment	Max	Min	**Mean**	Std
DQN	Pong	20.0	−13.0	**11.48**	5.12
	Boxing	100.0	−9.0	**79.86**	17.23
DDPG	HalfCheetah	9486.12	−586.12	**6448.06**	2914.85
	Ant	4126.43	−1568.64	**1483.38**	1349.12







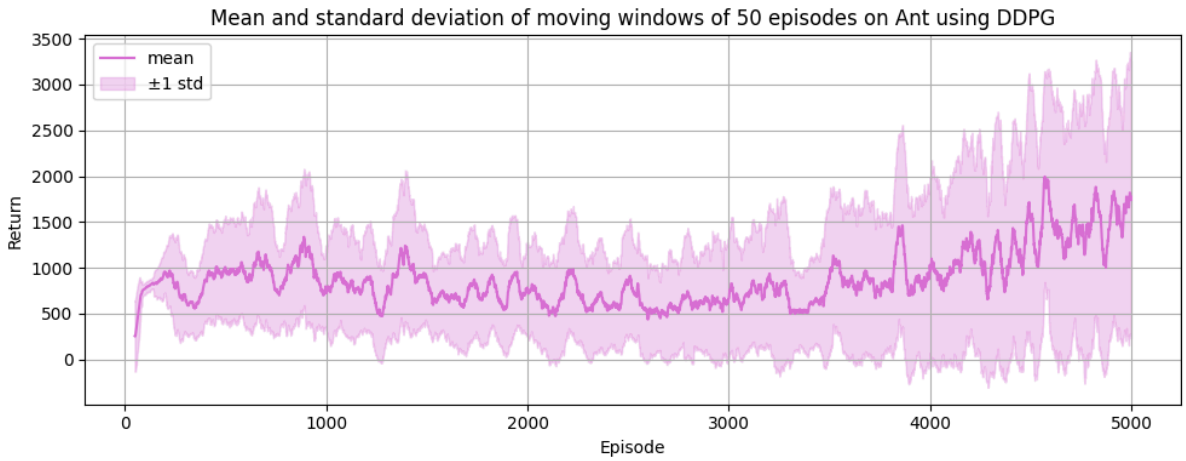


Figure 5: Training curves of DQN on Pong and Boxing (upper two) and DDPG on HalfCheetah and Ant (lower two). The curves show the moving average of the returns over the last 50 episodes, while the shaded area represents the standard deviation of the returns.

3.3.2 Testing Results

To make the results easy to check and reproduce, I store the trained QNet of DQN and the trained actor-critic networks of DDPG as checkpoint files. You can easily run the code I submit in test mode, which will load the saved models and evaluate their performance by default. And here I also present 10 repeated test results and their average of the two algorithms on the four environments in Table 4.

Table 4: 10 repeated test results and their average of DQN on Pong and Boxing and DDPG on HalfCheetah and Ant. The average results are highlighted in bold.

Exp No.	Pong	Boxing	HalfCheetah	Ant
1	10	68	8072.03	4007.62
2	14	90	8162.24	4096.54
3	12	53	7542.91	4093.46
4	10	77	7937.41	1796.35
5	13	92	7857.49	4033.6
6	7	84	6596.72	3970.02
7	15	91	8040.08	4146.32
8	8	78	8075.71	4278.8
9	10	100	8274.28	3680.58
10	16	87	3191.33	3842.76
average	**11.5**	**82**	**7375.02**	**3794.61**

However, when comparing the results with the training statistics in Table 3, I find that while the average returns of DQN on Pong and Boxing are close to the training results, the average returns of DDPG on HalfCheetah and Ant achieve even higher episode returns than the ones during training. After reviewing the code and consulting online resources, I think the

reason is that the target smoothing technique I added to DDPG will make the agent take a noisy trajectory during training, which can lead to a lower average return. When testing, the agent always takes the output of the actor network directly without adding noise, making the agent perform better than during training. This is a common phenomenon in reinforcement learning, where the training process may not reflect the true performance of the agent due to exploration noise and other factors.

3.3.3 Impacts of Improving modifications to DDPG

To evaluate the impacts of the improving modifications I added to DDPG, I conduct an ablation study by comparing the performance of DDPG with and without these modifications. Specifically, I train two versions of DDPG with the same shared hyperparameters: one with the original algorithm and another with the modifications (delayed policy updates, target policy smoothing, and delayed target updates). The training curves of the two versions of DDPG on HalfCheetah (2 experiments for each version) are shown in Figure 6, while the training curves on Ant (1 experiment for each version) with standard deviation shaded area are shown in Figure 7.

From the training curves, we can see that the DDPG with delayed policy updates, target policy smoothing, and delayed target updates results in a more stable training process and higher average returns than the original DDPG. Note that the results of the original DDPG while training and testing are almost the same, but the improved DDPG can achieve a better performance during testing than during training. This indicates the performance improvement brought by the modifications are even larger than these two Figures show.

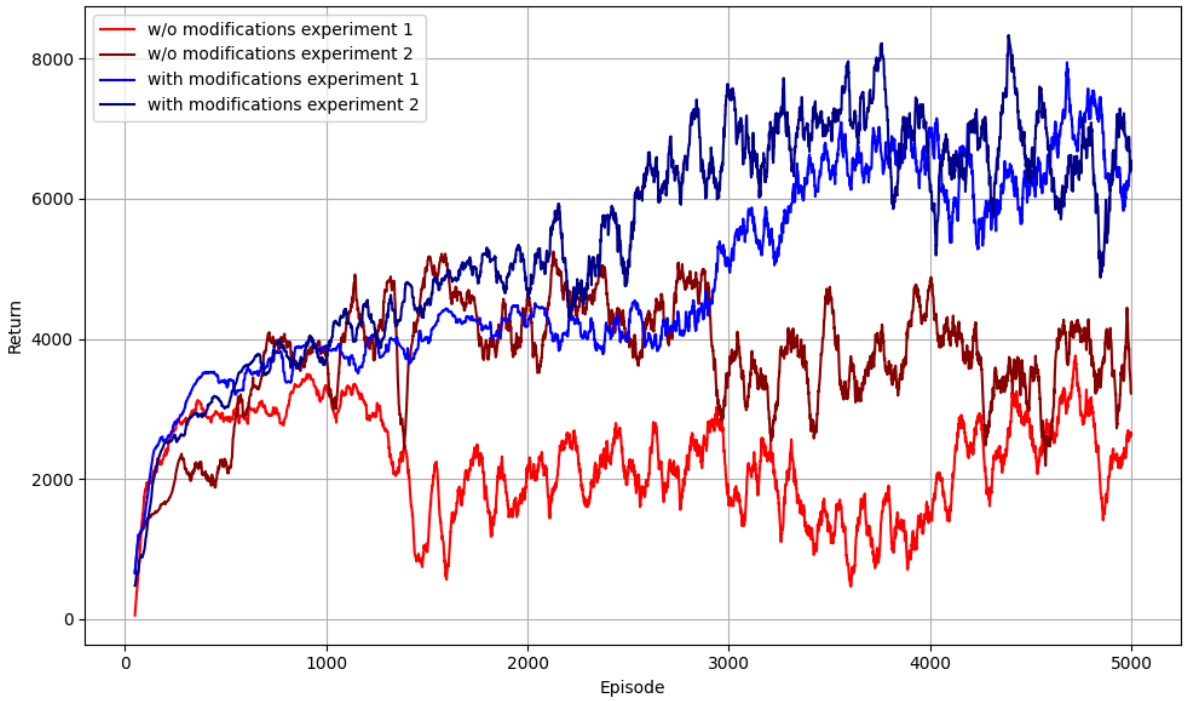


Figure 6: DDPG training curves on HalfCheetah with and without the improving modifications.

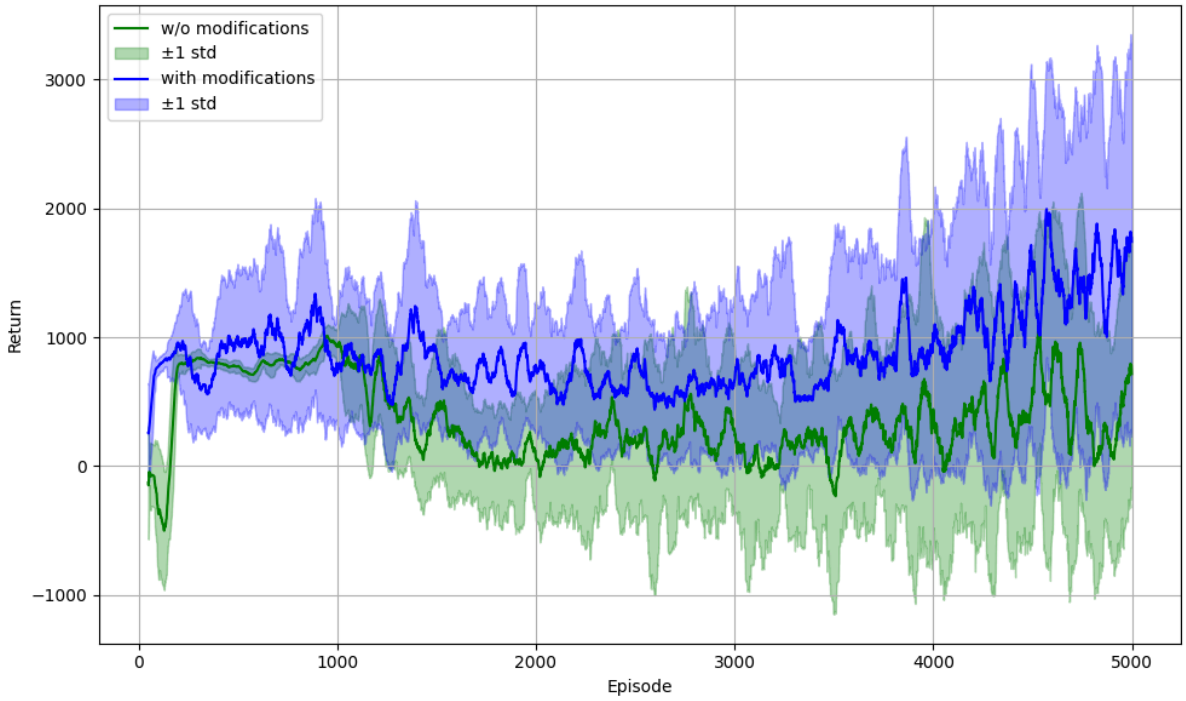


Figure 7: DDPG training curves on Ant with and without the improving modifications.

4 Conclusion

In this term project, I implemented DQN and DDPG algorithms and conducted experiments on two Atari environments (Pong and Boxing) and two MuJoCo environments (HalfCheetah and Ant). The results show that DQN can achieve satisfactory performance on the two Atari games, while DDPG presents a feasible performance with relatively large variance on the two MuJoCo environments. I improved the stability and efficiency of DDPG by adding delayed policy updates, target policy smoothing, and delayed target updates. The ablation study shows that these modifications can significantly improve the performance of DDPG. Overall, this project demonstrates the effectiveness of DQN and DDPG algorithms in solving reinforcement learning problems in Atari games and MuJoCo environments, which serves as a good exercise to deepen my understanding of the principles and challenges of reinforcement learning.

References

[1] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing Function Approximation Error in Actor-Critic Methods,” *CoRR*, 2018, [Online]. Available: http://arxiv.org/abs/1802.09477