Assignment Five Report

Hang Zheng 520021911347

May 11, 2023

Contents

1 Introduction

In the previous assignment, we have applied Deep Q Network (DQN) to solve mountain car problem, which has continuous state space and discrete action space.

In this assignment, I implemented a program that realizes A3C and DDPG, in order to solve the Pendulum problem, a classical RL control environment with continuous action space provided by OpenAI.

2 Preliminaries

In this section, I will give a brief introduction to the Pendulum environment and the principle of the two algorithms we mentioned above, A3C and DDPG.

2.1 Pendulum

The Pendulum environment is a reinforcement learning environment in OpenAI Gym that is designed to train an inverted pendulum controller.

In the Pendulum environment, there is a pole that can rotate around a fixed pivot point at the bottom of the pole, which is attached to a base. The angle of the pole is determined by a single pendulum angle, where a larger angle results in a higher rotation of the pole. The controller's task is to apply torque to the pole to keep the single pendulum angle at 0 degrees in the vertical direction.

The illustration schematic of Pendulum environment is shown in Fig 1.

Figure 1: Pendulum environment

More details about Pendulum environment:

- The pendulum starts in a random position every time, and the done flag returned by the environment will always be False until reaches max_episode_steps.
- The state space of the Pendulum environment is a three-dimensional vector, including sin and cos of the pole's angle and the pole's angular velocity. Denote angle as θ and angular velocity as ω, the initial state of the environment is $\theta \in [-\pi, \pi]$, $\omega \in [-1, 1]$, and the range of the observation space vector is $sin(\theta) \in [-1, 1]$, $cos(\theta) \in [-1, 1]$, $\omega \in [-8, 8]$
- The reward function is:

$$R = -\theta^2 - 0.1\omega^2 - 0.001 \cdot action^2$$

  In general, the more straight upward and stable it is, the better reward it would get.

2.2 A3C

The A3C (Asynchronous Advantage Actor-Critic) algorithm is a reinforcement learning algorithm **based on the Actor-Critic model**, designed to solve reinforcement learning problems in **continuous state and action spaces**.

A3C is an improvement upon the A2C (Advantage Actor-Critic) algorithm, which also uses the Actor-Critic model and updates the Actor and Critic networks using the Advantage function.The Actor-Critic model of the A2C algorithm consists of two neural networks:

- The Actor network and the Critic network. The Actor network learns the policy, i.e., selects an action based on the current state.
- The Critic network estimates the state value function, i.e., estimates the cumulative reward based on the current state.

The A2C algorithm uses the Actor-Critic model and TD (Temporal Difference) error to calculate the Advantage function, which is then used to update the Actor and Critic networks' parameters.The Advantage function is a measure of the goodness of taking a particular action in the current state relative to the average level. Its formula is:

$$A(s,a) = Q(s,a) - V(s)$$

where s represents the current state, a represents the current action, Q(s,a) is the state-action value function, and V(s) is the state value function. The greater the value of the Advantage function, the greater the benefit of choosing that action.

Compared to A2C, the main difference of A3C lies in its use of **asynchronous updates**. A3C trains **multiple asynchronous agents in parallel** with their own experience replay buffers and neural network parameters, which can effectively reduce training time and improve the algorithm's stability and robustness.

Training The training process of A3C consists of two phases: sampling and updating.

1. In the sampling phase, multiple asynchronous agents interact with the environment in parallel and collect a batch of experience data.
2. In the updating phase, the Actor and Critic networks' parameters are updated using the Advantage function based on the sampled experience data.

Asynchronous training can effectively reduce training time and improve the algorithm's stability and robustness.

The pseudo code of the A3C algorithm is as shown in figure 2.

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t,\theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1,\ldots,t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 2: Pseudo code of A3C Algorithm

2.3 DDPG

The DDPG (Deep Deterministic Policy Gradient) algorithm is a reinforcement learning algorithm based on the Actor-Critic model, which is used to handle reinforcement learning problems in **continuous action spaces**. The DDPG algorithm combines **deep neural networks and policy gradient methods**, enabling efficient learning of optimal policies in continuous action spaces.

Just like in A3C algorithm, there are a Actor network and a Critic network in the DDPG algorithm as well. The DDPG algorithm uses Actor-Critic model and **Q-learning** method to update the parameters of the Actor and Critic networks.

Action clipping In the DDPG algorithm, the output of the Actor network is a continuous action vector with a value range of [-1,1]. To ensure that the generated action vector is within a reasonable range, the DDPG algorithm uses a method called "action clipping". Specifically, when the generated action vector by the Actor network exceeds the feasible action value range, it is clipped to the feasible range.

Experience replay and Target network The DDPG algorithm also uses the methods of experience replay and target network to improve the stability and convergence of the algorithm. Experience replay is a method of training using historical experience data, which can effectively reduce sample correlation and improve training efficiency. The target network is a technique used to stabilize training, with its parameters being lagged copies of the parameters of the Actor and Critic networks. The parameter update of the target network is done using exponential averaging, making the changes in the target network's parameters smoother.

Training The training process of the DDPG algorithm consists of two stages: sampling and updating.

1. In the sampling stage, the agent interacts with the environment to collect a batch of experience data.
2. In the updating stage, the Actor and Critic networks are updated based on Q-learning using the methods of experience replay and target network.

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

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' 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 3: Pseudo code of DDPG Algorithm

3 Experimental Content and Result

In this section, I conducted several experiments and all the models are tested for 10 episodes on the Pendulum environment.

3.1 A3C

3.1.1 Baseline

In this experiment, I set the parameters as follows: (a) $num\_of\_workers = 8$, (b)$max\_episode\_steps = 200$, (c) $discounting\ factor\ \gamma = 0.99$, (d) $actor\_lr = 1e-4$, (e) $critic\_lr = 0.002$, and train the A3C model for 1500 episodes.

The A3C model successfully converged and the reward result is shown as in Fig 4.

Figure 4: Reward in each episode with A3C

From the result, we can see that the at about 400 episodes, the A3C model converges and it is not stable enough for that even after convergence, there still exist some episodes with lower rewards than expected.

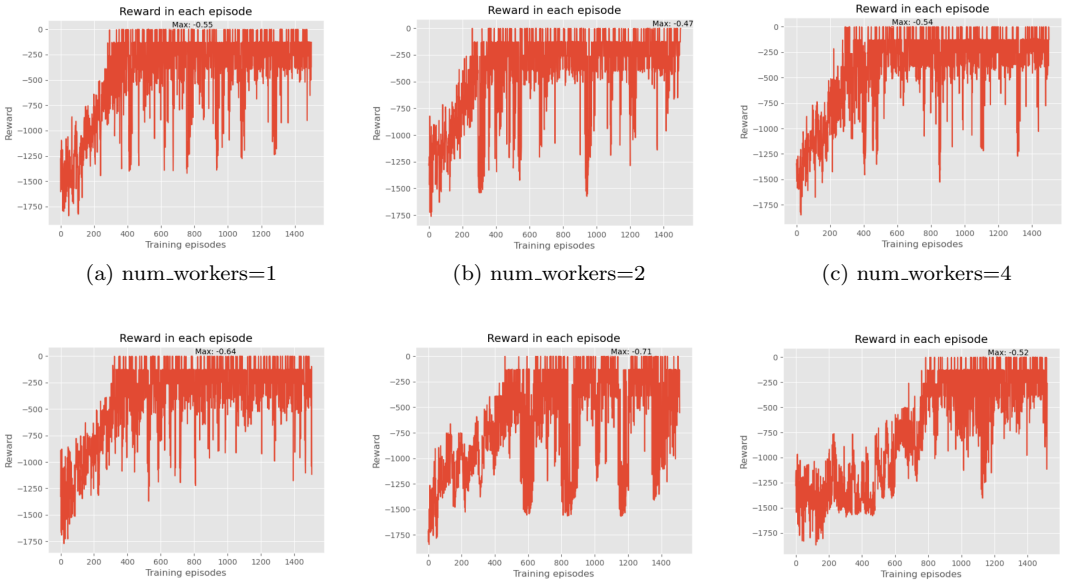
3.1.2 Experiment on number of workers

In A3C, there are multiple asynchronous agents working in parallel and the number of agents will significantly affect the performance. In this experiment, I train the A3C models with different setting of num_workers and test each one for 10 episodes. The result is shown in Table 1 and Fig 5.

Num_workers	reward
1	-252.03862005146235
2	-409.07099300438745
4	-194.4021351322792
8	-183.23864267349603
12	-248.6517300991797
20	-302.0860496613829

Table 1: Average reward in 10-episodes test of A3C with different number of workers

From the result, we can see that:



(a) num_workers=1 (b) num_workers=2 (c) num_workers=4

(d) num_workers=8 (e) num_workers=12 (f) num_workers=20

Figure 5: Reward curves of A3C with different number of workers

1. With the number of workers increasing, the algorithm convergence speed and stability increase at the first stage and then goes down.
2. The reason for the increase of convergence speed and stability in the first stage maybe that with more agents asynchronously sampling experience, the A3C model could see more diverse data and thus has higher training efficiency.
3. The reason for the decrease of convergence speed and stability in the second stage maybe that with too many agents, the local networks of each agent are more different with the global networks, which may result in the oscillations in the training process.

3.1.3 Experiment on gamma

In reinforcement learning, the discounting factor gamma is an important parameter that controls the influence of future rewards.

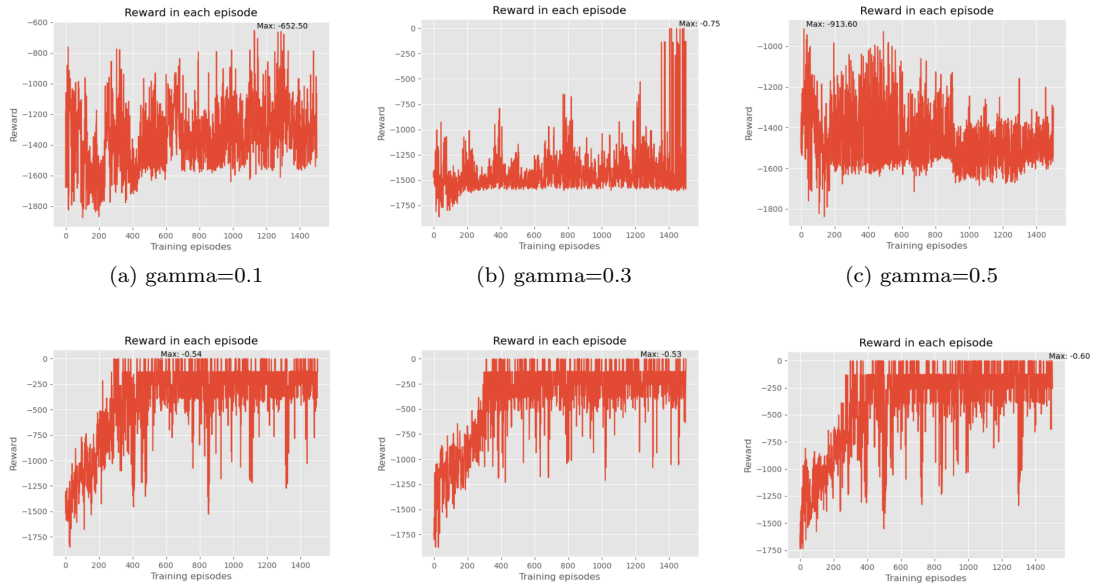
- When gamma is close to 0, the agent focuses more on immediate rewards and tends to take actions that lead to immediate rewards.
- When gamma is close to 1, the agent focuses more on long-term rewards and tends to take actions that lead to long-term benefits.

In this experiment, I train the A3C models with different setting of discounting factor γ and test each one for 10 episodes. The result is shown in Table 2 and Fig 6.

gamma	reward
0.1	-1050.2299393489088
0.3	-1298.472530807666
0.5	-1206.324375390614
0.9	-194.4021351322792
0.95	-279.5949821126673
0.99	-285.9356146355459

Table 2: Average reward in 10-episodes test of A3C with different gamma

From the result, we can see that:



(a) gamma=0.1 (b) gamma=0.3 (c) gamma=0.5

(d) gamma=0.9 (e) gamma=0.95 (f) gamma=0.99

Figure 6: Reward curves of A3C with different gamma

1. A gamma smaller than 0.9 will make the algorithm unable to converge.
2. A3C models with a gamma greater than 0.9 have similar performance.

3.1.4 Experiment on updating interval

In my implementation, each agent updates the global network after a fixed interval steps, which is set as a hyper-parameter. In this experiment, I trained the A3C models with different setting of updating interval and test each one for 10 episodes. The result is shown in Table 3 and Fig 7.

updating interval	reward
5	-184.9573184522118
10	-236.38840404276647
20	-430.0341965200955
50	-1030.3653679869399
100	-1227.6064954429776
200	-1185.5872847096148

Table 3: Average reward in 10-episodes test of A3C with different updating interval

From the result, we can see that:

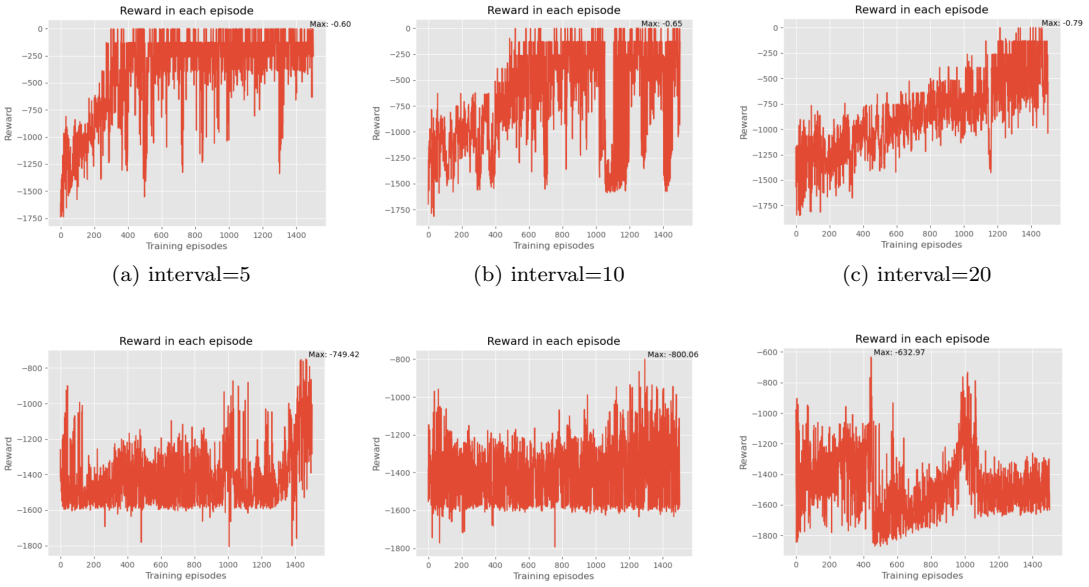
1. The A3C algorithm only converges when the updating interval is small (than 20 in this task).
2. With a updating interval is large, the hysteresis of the updating of local networks will increase, which will lead to a greater difference between the local networks and the global networks, and thus affect the training efficiency significantly.

3.2 DDPG

3.2.1 Baseline

In this experiment, I set the parameters as follows: (a)$max\_episode\_steps = 200$, (c) $discounting\ factor$ $\gamma = 0.99$, (d) $actor\_lr = 0.001$, (e) $critic\_lr = 0.002$, and train the DDPG model for 1500 episodes.

The DDPG model successfully converged and the reward result is shown as in Fig 8.



(a) interval=5 (b) interval=10 (c) interval=20

(d) interval=50 (e) interval=100 (f) interval=200

Figure 7: Reward curves of A3C with different updating interval

Figure 8: Reward in each episode with DDPG

From the result, we can see that the at about 200-300 episodes, the DDPG model converges and it is much more stable than the A3C algorithm for that after convergence, the reward of each episode is stably under 500.

3.2.2 Experiment on tau

In the implementation of DDPG, the updating of the target network's parameter uses exponential averaging to make the changes in the target network's parameters smoother.

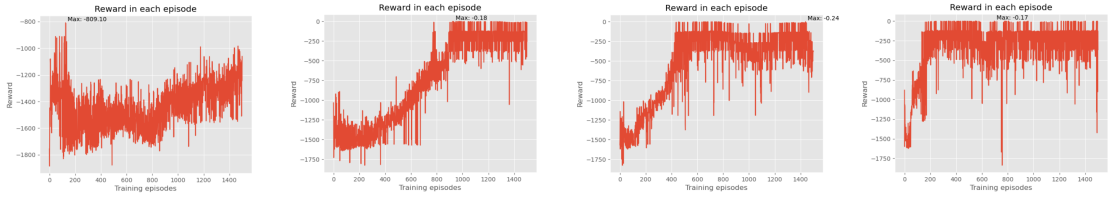
Specifically, the target network parameter update is performed using the following exponential averaging method:

$$\theta_{target} \leftarrow \tau \times \theta_{target} + (1 - \tau) \times \theta$$

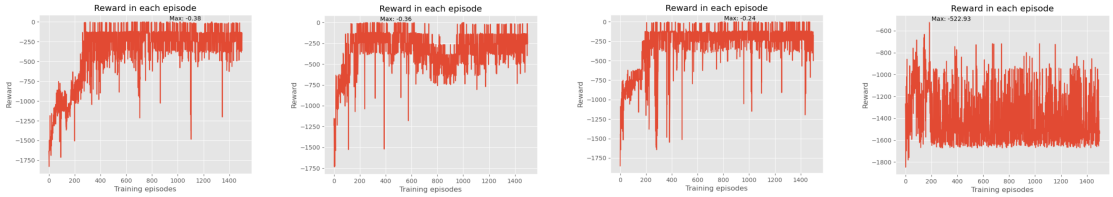
where θ_{target} is the target network parameter, θ is the current network (i.e., Actor or Critic) parameter, τ is a small parameter less than 1, known as the soft update factor. In this experiment, I train the DDPG models with different setting of τ and test each one for 10 episodes. The result is shown in Table 4 and Fig 9.

tau	reward
0.0001	-1385.6568334868616
0.0005	-202.73612635461734
0.001	-188.7943648486581
0.005	-270.84152788699436
0.01	-201.47544444801693
0.05	-165.89744200120774
0.1	-165.40058034074914
0.5	-926.148333039444

Table 4: Average reward in 10-episodes test of DDPG with different value of tau



(a) tau=0.0001 (b) tau=0.0005 (c) tau=0.001 (d) tau=0.005



(e) tau=0.01 (f) tau=0.05 (g) tau=0.1 (h) tau=0.5

Figure 9: Reward curves of DDPG with different value of tau

From the result, we can see that:

1. A tau with value between 0.001 and 0.1 is more suitable for this task.
2. If τ is too small, the updating of θ_{target} is too slow, it may lead to a slower convergence rate of the algorithm (just like that in Fig 9a and Fig 9b).
3. If τ is too large, the updating of θ_{target} is too rapid, it may lead to the instability and decreased convergence of the algorithm (just like that in Fig 9h).

4 Conclusion

During the process of this experiment, 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.