Assignment 5

BY YUSEN ZHENG

May 11, 2023

Email: zys0794@sjtu.edu.cn

Student ID: 520021911173

1 Introduction

In this study, we implemented and conducted a comparative analysis of two Reinforcement Learning (RL) algorithms: the Asynchronous Advantage Actor-Critic Architecture (A3C) and the Deep Deterministic Policy Gradient Architecture (DDPG). The performance of these algorithms was evaluated on a classic RL control problem, namely the Pendulum, using the testing environment provided by OpenAI Gym. The results demonstrate that, in this task, **DDPG achieves better convergence performance than A3C**, with faster and more stable convergence. Testing over 100 episodes, the evaluation reward of the agent **trained with DDPG is around -150**, while that of the agent **trained with A3C is around -400**.

2 Model Architecture

2.1 Asynchronous Advantage Actor-Critic Architecture

The Asynchronous Advantage Actor-Critic (A3C) is a reinforcement learning algorithm that uses a neural network to approximate both the policy and the value function. It is called "asynchronous" because it allows multiple "workers" to operate in parallel and update the network parameters asynchronously, which can significantly speed up the training process. The "advantage" refers to the use of the advantage function, which estimates the advantage of taking a particular action in a given state over the average action value. A3C avoids the problem of too strong correlation of experience playback, and at the same time achieves an asynchronous and concurrent learning model. Different workers can use different exploration strategies, which also increases diversity.

Let $\pi(a_t|\, s_t;\theta)$ denotes the policy network and $V(s_t;\theta_v)$ denotes the value network. When calculating the advantage function $A(s_t,a_t;\theta,\theta_v)$, k-step rewards are used, i.e. $\sum_{i=0}^{k-1}\gamma^i r_{t+i}+\gamma^k V(s_{t+k};\theta_v)-V(s_t;\theta_v)$. Another trick A3C uses is to introduce the entropy regularization term with respect to the policy parameters takes the form $\nabla_{\theta'}\log\pi(a_i|\, s_i;\theta')(R-V(s_i;\theta'_v))+\beta\nabla_{\theta'}H(\pi(s_i;\theta'))$, where H is the entropy.

The pseudocode for the DDPG algorithm is shown in Algorithm 1. And the model we used in this task is shown in Figure 1.

Algorithm 1

// 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 gradient $d\theta\leftarrow 0$ and $d\theta_v\leftarrow 0$
  Synchronize thread-specific parameters $\theta'=\theta$ and $\theta'_v=\theta_v$
  $t_{\text{start}}=t$
  Get start 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** $t - t_{\text{start}} == t_{\max}$
$R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t; \theta'_v) & \text{for non-terminal } s_t \end{cases}$
for $i \in \{t-1, \ldots, t_{\text{start}}\}$ **do**
$R \leftarrow r_i + \gamma R$
Accumulate gradient wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|\, s_i; \theta')(R - V(s_i; \theta'_v))$
Accumulate gradient wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \frac{\partial (R - V(s_i; \theta'_v))^2}{\partial \theta'_v}$
end for
Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{\max}$

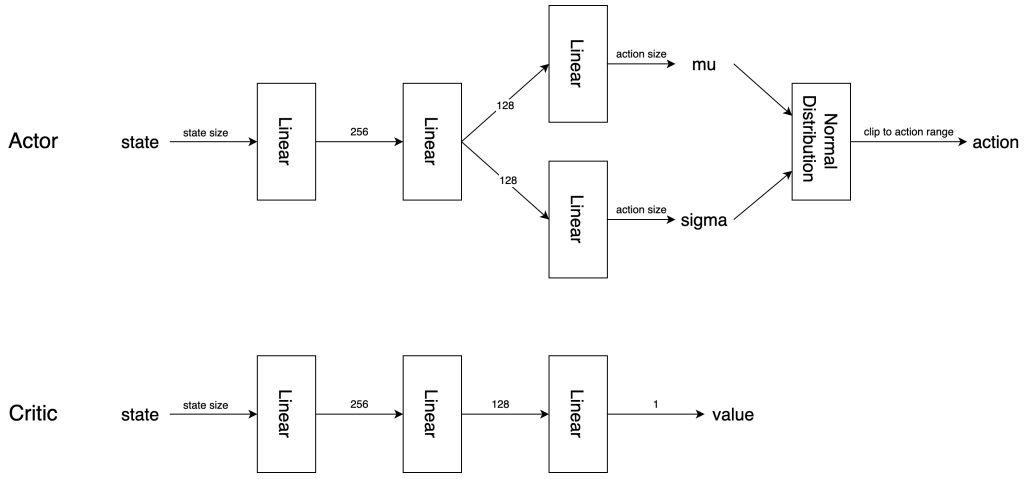

Figure 1. A3C model architecture in `Pendulum-v1` task

2.2 Deep Deterministic Policy Gradient Architecture

The Deep Deterministic Policy Gradient (DDPG) algorithm is an extension of the deep Q-learning algorithm to continuous action spaces. The DDPG architecture consists of two neural networks: an actor network $\mu(s|\, \theta^\mu)$ and a critic network $Q(s, a|\, \theta^Q)$, where θ^μ and θ^Q denote the network weights, respectively.

In DDPG, the actor network is used to determine an optimal policy that maximizes the total rewards in a "deterministic" way. Unlike DQN, we use $\mu(s|\, \theta^\mu)$ to simulate the $\text{argmax}_a Q(s, a)$ function. The critic network is used to calculate the Q-value. When updating the critic network, we minimize the mean squared error (MSE) loss function $\sum_i (y_i - Q(s_i, a_i|\, \theta^Q))^2$, where $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\, \theta^{Q'})$. When updating the actor network, we calculate the policy gradient

$$pg = \frac{\partial Q(s, \mu(s|\, \theta^\mu)|\, \theta^Q)}{\partial \theta^\mu} = \frac{\partial Q(s, a|\, \theta^Q)}{\partial a} \cdot \frac{\partial \mu(s|\, \theta^\mu)}{\partial \theta^\mu}$$

and then use gradient ascent to update θ^μ, i.e. $\theta^\mu \leftarrow \theta^\mu + \alpha \cdot pg$. When updating the target network weight θ', we use a "soft update" method, i.e. $\theta \leftarrow \tau\theta + (1-\tau)\theta'$, with $\tau \ll 1$. We also incorporate exploration in DDPG. To select the policy action a, we add random noise $\mathcal{N}$, i.e. $a = \mu(s|\, \theta^\mu) + \mathcal{N}$.

The pseudocode for the DDPG algorithm is shown in Algorithm 2. And the model we used in this task is shown in Figure 2.

Algorithm 2

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 the 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
  Execute action a_t and observe reward r_t and observe new state s_{t+1}
  Store transtion (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 $y_i = \begin{cases} r_i & \text{for terminal } s_i \\ r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\,\theta^{\mu'})|\,\theta^{Q'}) & \text{for non} - \text{terminal } s_i \end{cases}$
  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

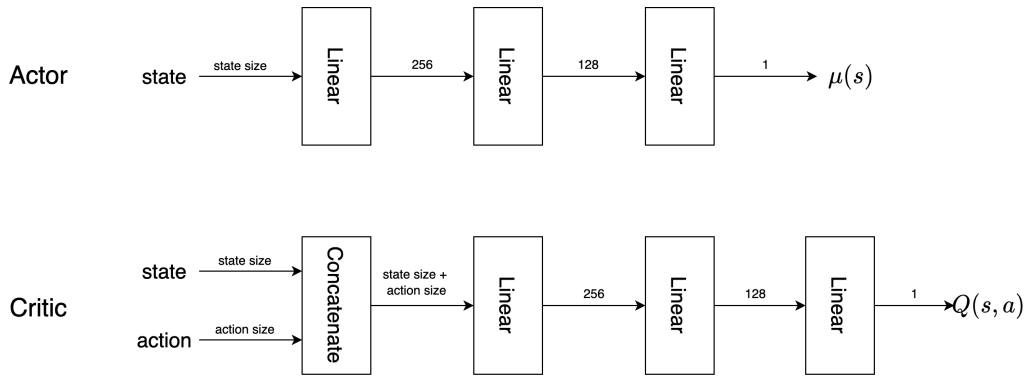

Figure 2. DDPG model architecture in `Pendulum-v1` task

3 Experiments

3.1 Environment

We use the `Pendulum-v1` environment provided by OpenAI gym[1], which is designed to simulate the inverted pendulum swingup problem based on the classic problem in control theory. In this environment, the system consists of a pendulum attached at one end to a fixed point, while the other end remains free. The initial position of the pendulum is randomized, and the objective is to apply torque to the free end in order to swing the pendulum into an upright position. The desired state is achieved when the center of gravity of the pendulum is directly above the fixed point.

The coordinate system of `Pendulum-v1` task is shown in figure 3.

1. https://gymnasium.farama.org/environments/classic_control/pendulum/

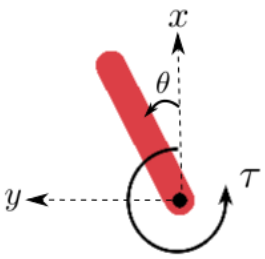

Figure 3. Pendulum coordinate system

Observation Space. The observation space consists of a triplet representing the x and y coordinates of the pendulum's free end (x, y) and its angular velocity ω.

Action Space. The action space consists of a single element representing the torque τ applied to the free end of the pendulum.

Reward. The reward function is defined as:

$$r = -(\theta^2 + 0.1 \cdot \omega^2 + 0.001 \cdot \tau^2)$$

It can be observed that as the angle deviation of the pendulum from the upright position increases, the angular velocity increases, or the applied torque increases, the reward decreases. When the pendulum is upright with zero velocity and no torque applied, the reward reaches its maximum value of 0.

Starting State. The starting state is a random angle $\theta \in [-\pi, \pi]$ and a random angular velocity $\omega \in [-1, 1]$. The gravity g is set to $10.0\,\mathrm{m{\cdot}s^{-2}}$.

Notation. The notations used in this paper is shown in table 1.

	Symbol	Min	Max	Unit
Angle	θ	-π	π	rad
Torque	τ	-2.0	2.0	N·m
x Coordinates	x	-1.0	1.0	m
y Coordinates	y	-1.0	1.0	m
Angular Velocity	ω	-8.0	8.0	rad/s

Table 1. Notation

3.2 Training

A3C. In A3C, we modeled the policy distribution using a normal distribution. The mean and variance of the normal distribution was the output of the actor network, and the variance can encourage exploration. During training, we used 4 workers to update their own local models and update the global model every 20 steps. Each worker ran 10000 episodes. The maximum number of steps per episode for the agent was set to 200 (the default value in gym). The discount factor for the reward was set to $\gamma = 0.99$, and the coefficient for the entropy regularization term was $\beta = 0.01$. The learning rate of the optimizer was set to 0.0001.

DDPG. In DDPG, we ran 1000 episodes with 2 updates to the actor and critic networks per episode, a batch size of 32, and a learning rate of 0.0001. The agent was limited to a maximum of 200 steps per episode (the default value in gym). The first 100 steps of training were considered warm-up, during which only the state, action, and reward were added to the replay buffer. During action selection, a Gaussian noise with a mean of 0 and a standard deviation of 0.005 was added as exploration. The reward discount factor γ was set to 0.9, and the parameter τ for soft weight updates was set to 0.01.

3.3 Results

3.3.1 A3C Results

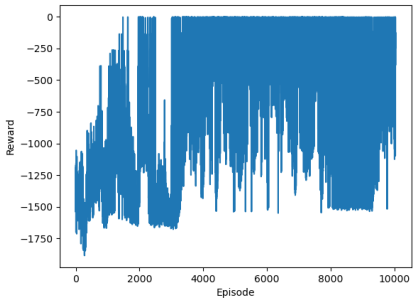

Figure 4. Train (result of one worker)

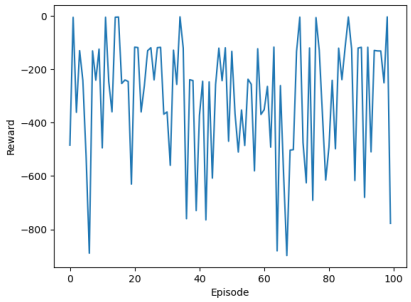

Figure 5. Test (without exploration)

3.3.2 DDPG Results

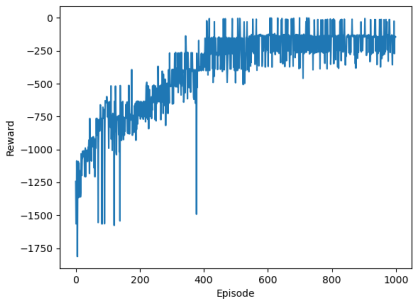

Figure 6. Train

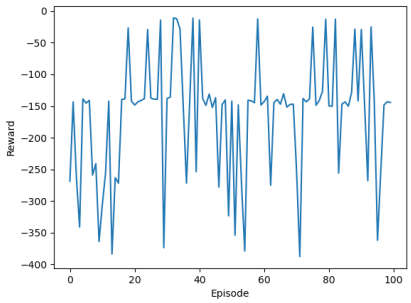

Figure 7. Test (without exploration)

4 Discussion and Conclusion

The results of this study showed that DDPG achieved better convergence results than A3C in the given scenario, with faster and more stable convergence. **During testing over 100 episodes, the evaluation reward of the agent trained by DDPG was around -150, while the evaluation reward of the agent trained by A3C was around -400.**

One possible reason for the better performance of DDPG is that it is more suited to continuous action spaces. A3C, on the other hand, is known to perform better in discrete action spaces. Additionally, DDPG uses a deterministic policy, **which can be advantageous in certain scenarios where consistency is important**.

Another factor that may have contributed to the better performance of DDPG is **the use of experience replay**. Experience replay allows the agent to learn from a wider range of experiences and can improve sample efficiency, which may have allowed DDPG to converge more quickly and with more stability.

Another possible reason for the large fluctuation of the A3C reward is **the use of multiple workers in the training process**. As each worker updates the model independently and asynchronously, there could be a significant variance in the rewards obtained by each worker, leading to a higher overall variance in the training process. On the other hand, DDPG is a single-agent algorithm and updates the model in a more centralized manner, which could lead to a more stable and consistent learning process.

In conclusion, the results of this study suggest that DDPG may be a more effective algorithm for continuous action spaces in certain scenarios. However, further research is needed to explore the performance of these algorithms in different scenarios and to optimize their hyperparameters for better results.

A Codebase

A.1 Train

For training, use the following command:

```
python3 ./run.py --mode train --model [Model Name: A3C or DDPG] --config
[Config Path: ./config/a3c.yaml or ./config/ddpg.yaml]
```

By default, the rewards obtained during training will be plotted and saved in the `images` folder, and the trained model will be saved with a `.pt` extension in the `modelset` folder.

A.2 Test

For testing, use the following command:

```
python3 ./run.py --mode test --model [Model Name: A3C or DDPG] --config [Config
Path: ./config/a3c.yaml or ./config/ddpg.yaml]
```

By default, the rewards obtained during testing will be plotted and saved in the `images` folder.