Report for Assignment 5

519021910913 黄喆敏

519021910914 孙赫誉

Directory Structure

```
lab5
├── code
│   ├── A3C
│   │   ├── A3C.py                # main function
│   │   ├── Adam.py               # shared adam for stochastic optimization
│   │   ├── a3c.npy
│   │   └── utils.py
│   └── DDPG
│       ├── agent.py              # agent for training, include soft update
│       ├── buffer.py             # replay buffer
│       ├── ddpg_actor.pth        # checkpoint of networks
│       ├── ddpg_critic.pth
│       ├── ddpg_no_noise.npy
│       ├── ddpg_with_noise.npy
│       ├── main.py               # main function
│       ├── model.py              # actor and critic net
│       ├── noise.py              # Ornstein-Uhlenbeck noise
│       └── utils.py
├── docs
│   ├── report.pdf
│   ├── assets
│   │   ├── a3c_code.png
│   │   ├── a3c_detail.png
│   │   ├── a3c_result.jpg
│   │   ├── compare_noise.png
│   │   ├── comparison.jpg
│   │   ├── ddpg_code.png
│   │   ├── ddpg_detail.png
│   │   ├── ddpg_result.jpg
│   │   └── environment.png
│   └── report.md
└── result
    └── result.mp4                # visualization
```

All codes are placed in `./code` directory. You can directly run `DDPG/main.py` or `A3C/A3C.py` to see the results of DDPG model or A3C model.

The package requirement is as follows:

- PyTorch 1.11.0. GPU is not important in this assignment, because neural networks in both two

methods are not very deep.

- Python >= 3.7.0.
- gym <= 0.22.0. Since we use gym.Monitor API to export video, the version can not be higher than 0.22.0.
- pygame 2.1.2.

Introduction

Actor-Critic method reduces the variance in Monte-Carlo policy gradient by directly estimating the action-value function.

In this assignment, we have implemented two improved AC methods – **Asynchronous Advantage Actor-Critic (A3C) and Deep Deterministic Policy Gradient (DDPG)** to solve problem with continuous and high-dimension action space. Besides, we have made some comparisons about these two methods.

Environment: Pendulum-v1

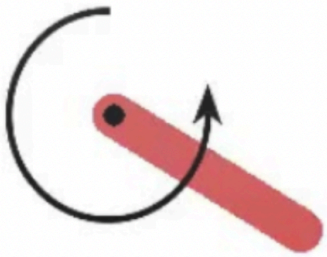


Figure 1: Pendulun-v0

Table 1: Inputs for Environments

Observation	Min	Max
$\sin(\theta)$	-1.0	+1.0
$\cos(\theta)$	-1.0	+1.0
θ_{dt}	-8.0	+8.0

The action *joint effort* ranges in $[-2, 2]$. And the precision equation of reward is

$$R = -\left(\theta^2 + 0.1 * \theta_{dt}^2 + 0.001 * \text{action}^2\right) \tag{1}$$

where $\theta \in [-\pi, \pi]$ and $\theta_d t$ represents the velocity of angle.

The initial state is random angle $\theta \in [-\pi, \pi]$ and random velocity $\theta_{dt} \in [-1, 1]$.

In this assignment, we have used **Pendulum-v1** environment provided by OpenAI Gym. It is a very classic environment in the control literature.

As shown in figure above, the goal is to swing a frictionless pendulum upright so it stays vertical, pointed upwards. It starts in a random position every time and it doesn't have a specified reward threshold nor at which the episode will terminate.

There are three observation inputs for the environment, whch represents its angle and angular speed. The inputs are shown in table 1.

Algorithms

Deep Deterministic Policy Gradient (DDPG)

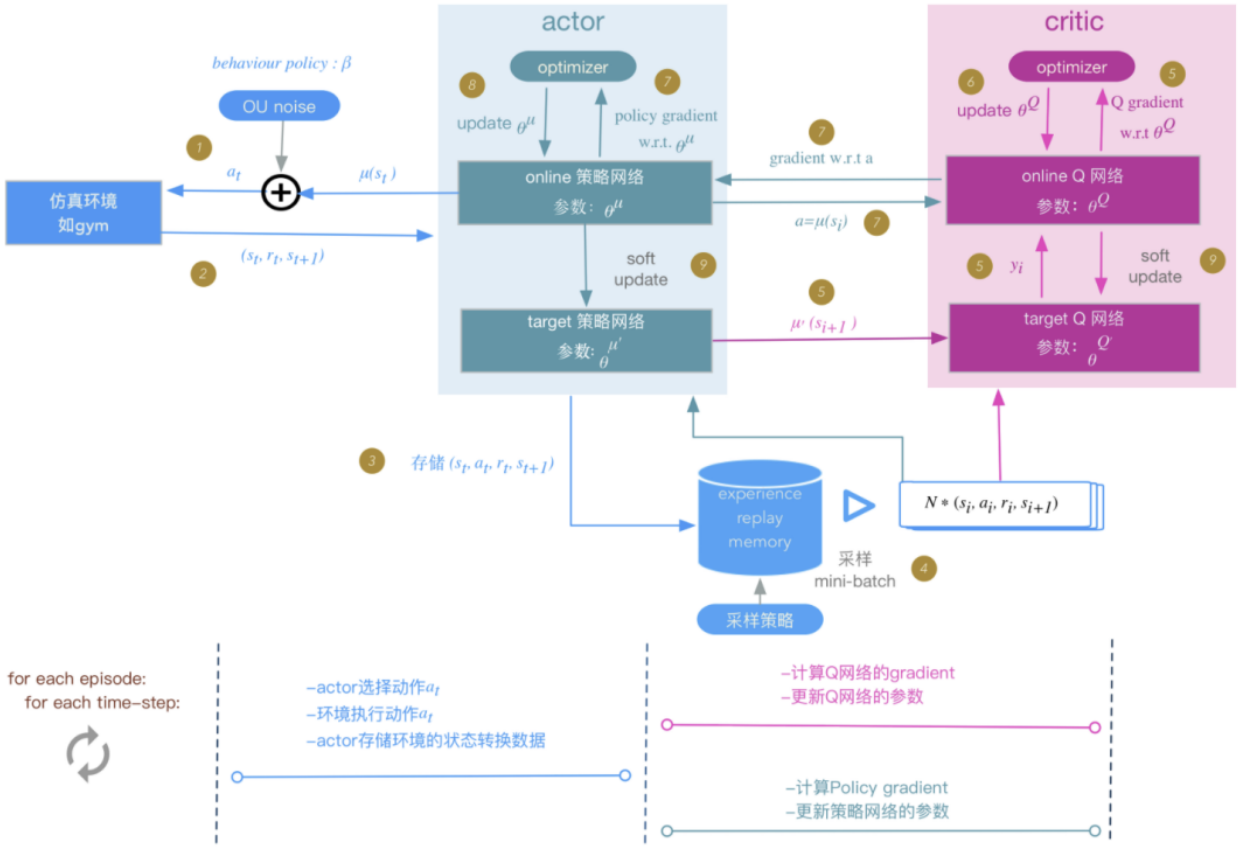
Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. [2] It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

Like DQN, we use **an experience replay buffer** in DDPG to approximate $Q^*(s,a)$. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but **it may not always be good to keep everything**. This may take some tuning to get right.

DDPG also make use of **target networks** similar to DQN. In DQN-based algorithms, the target network is just copied over from the main network every some fixed number of steps. However, DDPG algorithm has made some modifications. The target network is updated once per main network update by polyak averaging, which is also called **soft update** : $\theta^{'} = \tau\theta + (1-\tau)\theta^{'}$.

DDPG trains a **deterministic policy** in an off-policy way. To make DDPG policies explore better, we add noise to their actions at training time. The authors of the original DDPG paper recommended time-correlated **Ornstein-Uhlenbeck Noise**, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well. [5]

In summary, DDPG is a combination of DQN and DPG, and it is an off-policy Actor-Critic based algorithm. The following picture[4] and pseudocode can help you acquire a better understanding of DDPG algorithm.



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

Asynchronous Advantage Actor Critic (A3C)

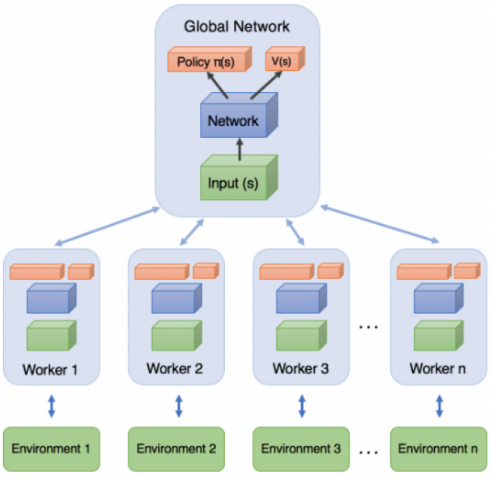
A3C, **Asynchronous Advantage Actor Critic**, is a policy gradient algorithm in reinforcement learning that maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$.[1] It operates in the forward view and uses a mix of n-step returns to update both the policy and the value-function. The policy and the value function are updated after every t_{max} actions or when a terminal state is reached.

The critics in A3C learn the value function **while multiple actors are trained in parallel** and get synced with global parameters every so often. The gradients are accumulated as part of training for stability, which is like parallelized stochastic gradient descent.

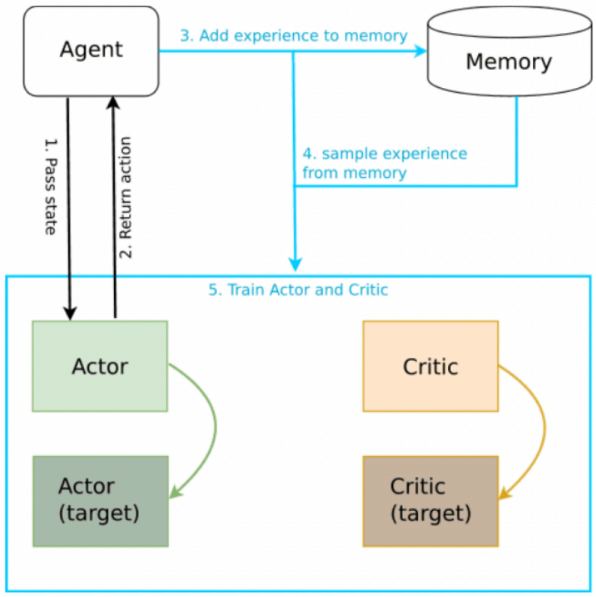
The algorithm details and its pseudocode are as follows.

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 $T > T_{max}$



(a) The network structure of A3C



(b) The network structure of DDPG

Experiment Details

DDPG

- We train 1000 episodes to observe the performance of DDPG. For each episode, the maximum step is 200.
- We use Adam for the optimizer. The learning rate of the actor and critic are 10^{-3} and 10^{-4} respectively.
- For Q we use L_2 weight decay of 10^{-2}, and use a discount factor of $\gamma = 0.99$.
- For the soft target updates we use $\tau = 0.001$.

- For network, We use ReLU to activate all hidden layers, and a tanh layer for the final output layer of the actor. The network has 2 hidden layers with 400 and 300 units. The layer weights and biases of both actor and critic are initialized from a uniform distribution.
- We use an OU process with $\theta = 0.15$ and $\sigma = 0.3$.
- We use a replay buffer size of 10^6. The size of minibatch is 64.

A3C

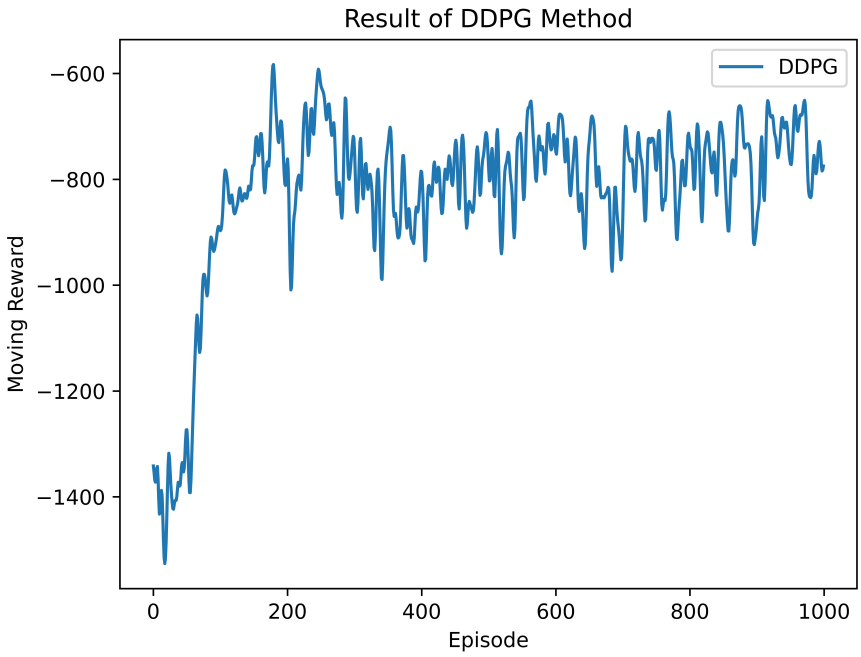
- We train 5000 episodes to observe the performance of A3C and for each episode the maximum steps is 200.
- The number of concurrent threads is 8, which is also the amount of workers.
- The learning rate of both networks are 10^{-4} , and the discount factor is $\gamma = 0.9$.

Result

For DDPG method, We have used `gym.Monitor` library to export video of the final result, which have been put in the `result` directory. For A3C method, you can directly run the code, and watch the visualization through `pygame` window.

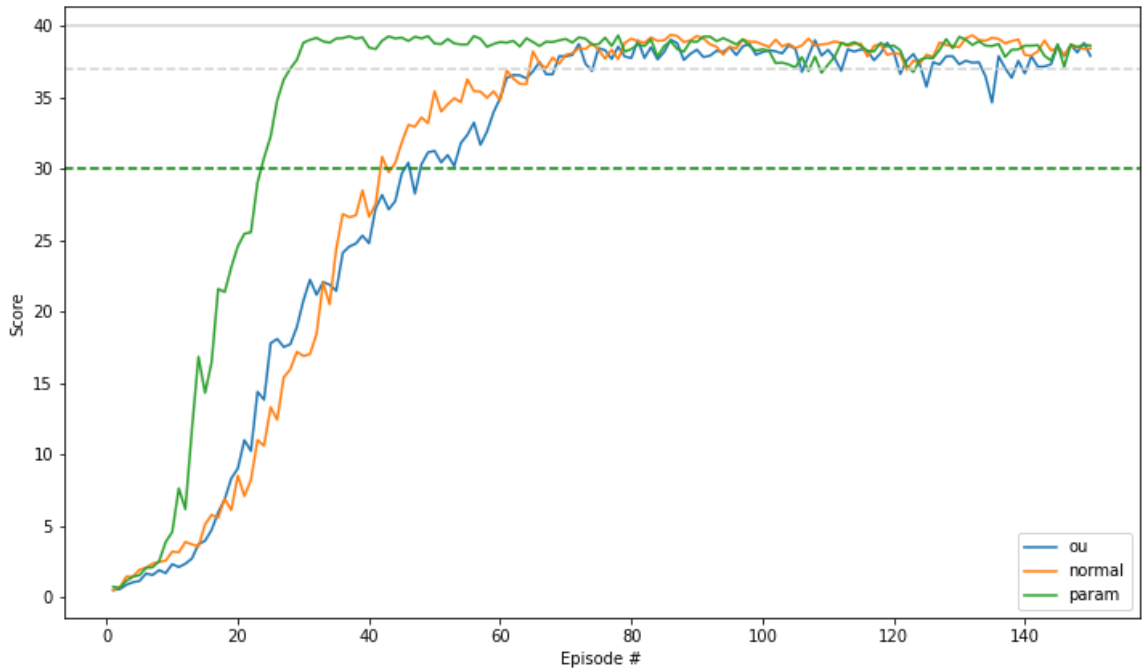
DDPG

The moving reward of each episode for DDPG is as follows. We have used Gaussian Blur to reduce the noise of the result. In our experiment, DDPG converges very fast. After about 200 episodes, the moving reward becomes relatively stable.



Besides, we have also noted that the author used **Ornstein-Uhlenbeck Process** to generate noise. From the author's perspective, it can generate temporally correlated noise in order to explore well in physical environments that have momentum. But as far as we know, Gaussian Noise/Normal Noise is more commonly used and behaves well.

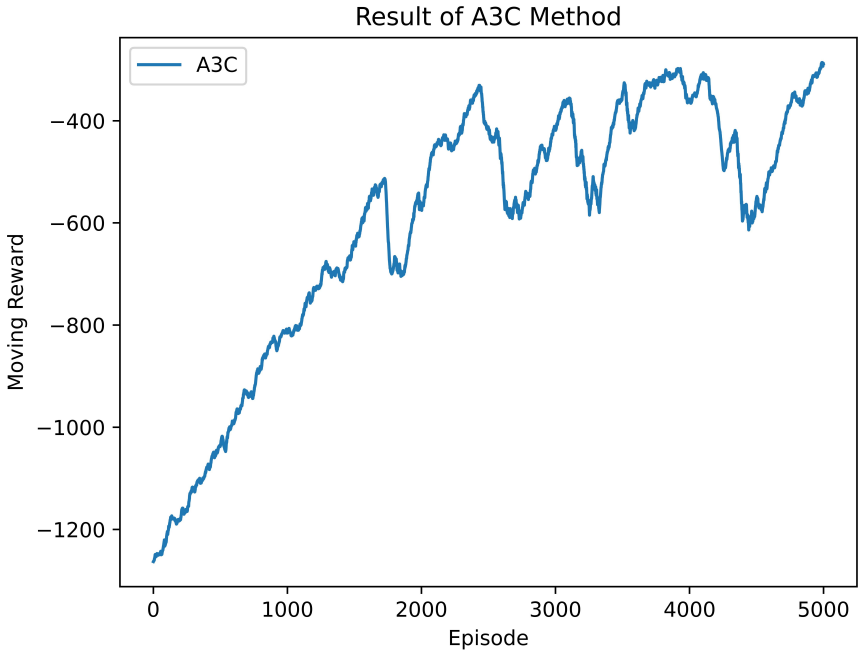
Therefore, we have made some comparison. It seems that OU noise doesn't work better than uncorrelated Gaussian noise. But both of them have satisfying results.



A3C

The method successfully converged. When we render the environment, we can observe that the pendulum will stay vertical after 1500 episodes. After that value, the moving rewards starts to be stable.

There is also a very interesting phenomenon which occurs in A3C that the moving reward will increase a little at the beginning.

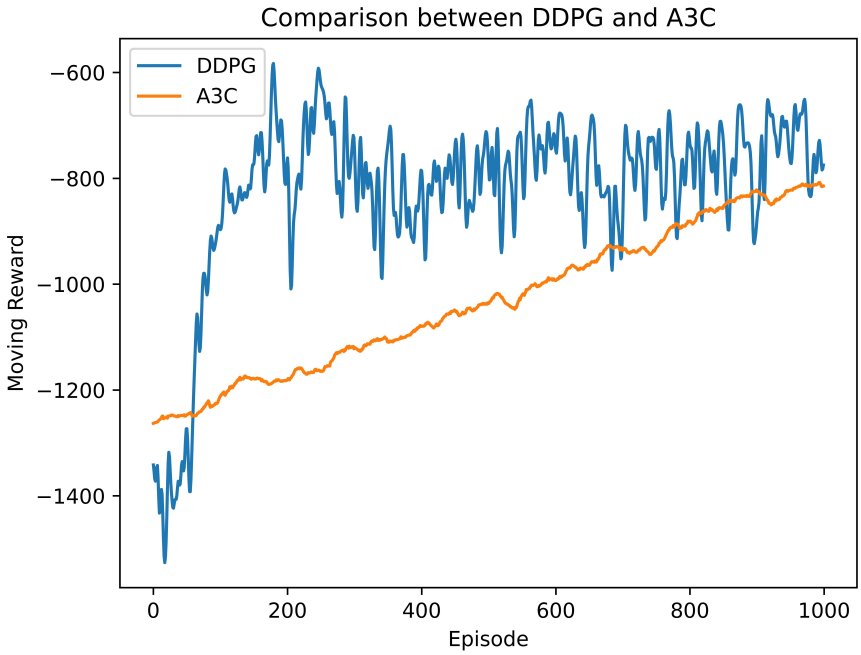


Comparison

Based on the results above, we have made comparison about two methods based on the rewards of first 1000 episodes.

From the figure below, we can see that **DDPG converges faster than A3C**. DDPG achieves a relative high reward about 150 episodes. But it takes A3C about 1000 episodes to achieve the same reward.

However, **the learning curve of DDPG is much nosier than A3C's**, no matter whether the OU noise is added. Therefore, we can use **TD3** as an optimized version of DDPG, which can make the training process more stable.



Conclusion

We have introduced and discussed details of A3C and DDPG methods, as well as implementing two methods in the assignment. Both methods are suitable for continuous actions. They can converge quickly and behave well in the given environment. Furthermore, we should also combine some other methods like PPO and DQN when we are faced with environment with discrete actions.

References

[1] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." *International conference on machine learning*. PMLR, 2016.

[2] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971* (2015).

[3] Hou, Yuenan, et al. "A novel DDPG method with prioritized experience replay." *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, 2017.

[4] https://zhuanlan.zhihu.com/p/111257402

[5] Kenfack, Lionel Tenemeza, et al. "Decoherence and tripartite entanglement dynamics in the presence of Gaussian and non-Gaussian classical noise." *Physica B: Condensed Matter* 511 (2017): 123-133.

[6] https://github.com/openai/baselines

[7] Fujimoto, Scott, Herke van Hoof, and Dave Meger. “Addressing Function Approximation Error in Actor-Critic Methods.” *arXiv preprint arXiv:1802.09477* (2018).