SHANGHAI JIAO TONG UNIVERSITY

# 2021 春强化学习课程报告

姓名：吴仕渠

班级：F1803304

学号：518021910665

指导教师：邓妮倩

# REPORT OF CS489 FINAL PROJECT

Wu Shiqu 518021910665
wushiqu123@sjtu.edu.cn

**Abstract.** In this project, I implemented some classical Reinforcement Learning algorithms, which includes: PPO, DPPO, DDPG, DQNs (Double DQN, Dueling DQN, Prioritized DQN). The algorithm's perfomance is checked with OpenAI Gym, Mujoco environments. First I implement DDPG algorithm and test its performence with Ant-v2, HalfCheetah-v2, Hopper-v2 and Humanoid-v2. The result shows that DDPG performs very well in the formal 3 environments where it achieved an average reward of **2232.8, 3848.4, 2532.2** separately for the last 20 training epochs when the step per epoch was set to 1000. The reward scores for these 3 environment are closed to or slightly better than the results in the original paper. Then I implemented PPO algorithm, which is the current RL baseline of OpenAI and it's famous for it's learning ability. PPO did a great job in HalfCheetah-v2 environment and achieved a an average reward of **8470.67** for the last 20 training epochs when the step per epoch was set to 5000. It can be seen from the real-time simulation window that agent with PPO-brain performs better and more stable than agent with DDPG-brain. Finally, I visualized several key comparative experiments and analyzed the causes of the difference. All the codes and experiment results (including demo videos) can be find on Github: CS489 Final Project

**Keywords:** PPO, DDPG, DQN, Mujoco

## 1 Introduction

When action space, state space for a problem is big enough, it's impractical to use a explicit Q-table to store the policy or Q-value. For the fact that it needs tremendous memory to store a Q-table of such size. So scientists introduce Deep Neural Network (DNN) to solve this problem:

1. Construct a neural network, let its input be: $S$, output be: $A$, it can replace the $choose\_action()$ function in Q-learning.
2. Construct a neural network, let its input be: $S, A$, output be: $Q$, it can replace the $policy\_update()$ function in Q-learning.

This is what we called: DQN [5].The core ideas of DQN are **Experience Replay** and **Fixed Q-target**. But there are several drawbacks of DQN, the most important of which are the following three: upward bias problem, random sampling problem, reward bias problem. That's why scientists proposed Double DQN [6], Prioritized DQN [7] and Dueling DQN [3] to solve these 3 problems correspondingly. However, DQNs can't handle the continuous action space problem and simple policy gradient algorithm is updated per-round, which results in slow learning speed and low efficiency. So at this time AC algorithms [8] comes to rescue. AC algorithm combines the advantages of value-based algorithms and policy-based algorithms: it can handle continuous action space problem while keeping the single-step update mechanism, which benefits model's learning speed. But the AC method is sample inefficient and it used on-policy. Thus, more advanced off-policy algorithm like DDPG [4], PPO [9] are introduced. Since DQNs and AC algorithms have been demonstrated in previous assignments, in this project, we focus on the idea of DDPG and PPO and its performance on the Mujoco environment.

## 2 DDPG

### 2.1 Background

Policies are usually stochastic. However, Silver et al. (2014) [1] and Lillicrap et al. (2016) [4] proposed deterministic policy gradient (DPG) for efficient estimation of policy gradients. Silver et al. (2014) [1] introduced the deterministic policy gradient (DPG) algorithm for RL problems with continuous action spaces. The deterministic policy gradient is the expected gradient of the action-value function, which integrates over the state space; whereas in the stochastic case, the policy gradient integrates over both state and action spaces. Consequently, the deterministic policy gradient can be estimated more efficiently than the stochastic policy gradient. The authors introduced an off-policy actorcritic algorithm to learn a deterministic target policy from an exploratory behaviour policy, and to ensure unbiased policy gradient with the compatible function approximation for deterministic policy gradients.

From then on, Lillicrap et al. (2016) [4] proposed an actor-critic, model-free, deep deterministic policy gradient (DDPG) algorithm in continuous action spaces, by extending DQN Mnih et al. (2015) [5] and DPG Silver et al. (2014) [1].

### 2.2 Core Idea

DDPG can be divided into 'deep' and 'deterministic policy gradient', and then 'deterministic policy gradient' can be subdivided into 'deterministic' and 'policy gradient'. It absorbs Actor-Critic's essence of Policy gradient's single step update, and also absorbs DQN's essence of Fixed Q-network and Experience Replay. Thus, DDPG is sample efficient while at the same time keeps good stability and convergence. The algorithm is shown in Fig 1.

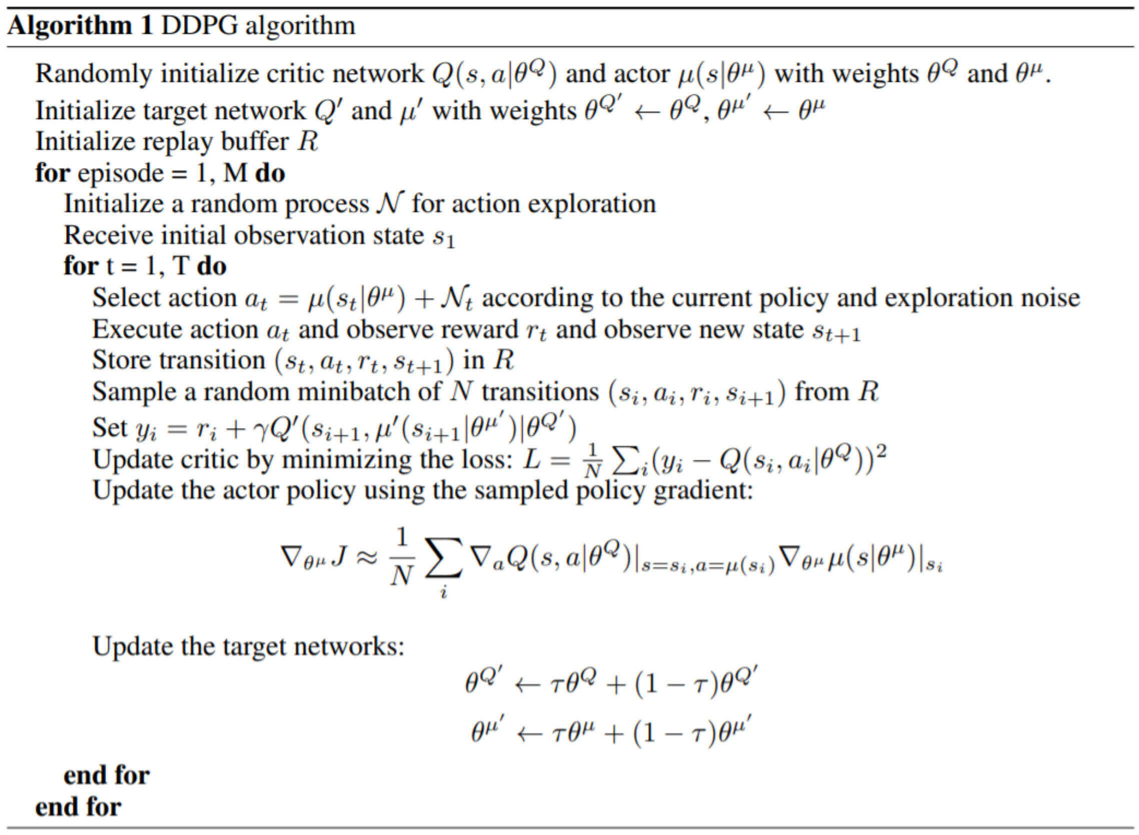
To choose a suitable random process N, in Lillicrap et al. (2016), the author utilized Ornstein-Uhlenbeck process proposed by Uhlenbeck  Ornstein (1930) to generate tempoprally correlated exploration for exploration efficiency in physical control problems with inertia. Furthermore, above random process may sometimes lead to local minimum problem. In Plappert et al. (2017), the author proposed parameter space noise for exploration. Such method add noise directly to the agent's parameters, which can lead to more consistent exploration and a richer set of behaviors. The three key points of this method are:

1. State-dependent exploration: We can still utilize Ornstein-Uhlenbeck process to generate a random process.
2. Perturbing deep neural networks: Noise is added to the parameters of neural networks, here in DDPG, we add noise to the actor network.
3. Adaptive noise scaling: The parameter noise requires us to pick a suitable scale, which is hard to choose. With adaptive noise scaling, such problem will be solved.

## 3 PPO

### 3.1 Background

PPO [9] is proposed by OpenAI in 2017. After that, it's widely used in complex RL settings and it's also the default Rl algorithm in OpenAI newly published papers. As one of the most advanced RL



**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
```

Fig. 1: DDPG algorithm

algorithms, PPO does well in many traditional RL problems and it's easy to implement and debug. The predecessor of PPO is TRPO (Trust Region Policy Optimization) [10], which is also published by OpenAI in 2015. There is another algorithm called ACER (Actor Critic with Experience Replay) [11] that has the similar performance with PPO but is much more complex in implementation. What's more, PPO has a better generalization performance than TRPO and ACER. Compared to A3C, DDPG, the advantage of PPO lies in better sample efficiency and higher robustness.

PPO uses adaptive KL-divergence to constrain the update scope of new policy. This constraint reduces the hypter-parameter sensitiveness of model and enhance the robustness.

### 3.2 Core idea

Policy-based algorithms (e.g. Policy Gradient, AC, A3C, DDPG ...) are sensitive to hyper-parameters (e.g. learning rate, step size). Therefore, their robustness is unsatisfactory. What's more, the sample efficiency of these policy-based algorithms is usually bad, which means that millions of iterations are needed to train a single model in complex problem settings. It will spend a lot of time to train and it needs high computational power. The algorithm is shown as below:

**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
        $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \text{KL}[\pi_{old}|\pi_\theta]$
        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 $\text{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \text{KL}_{target}$ then
        $\lambda \leftarrow \alpha\lambda$
    else if $\text{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \text{KL}_{target}$ then
        $\lambda \leftarrow \lambda/\alpha$
    end if
end for
```

One of the most significant things in PPO is about how to calculate loss. The loss of Critic (c-loss) can be treated as the square of TD-error. However, it's more complex to calculate the loss of Actor (a-loss). Generally speaking, there exists two different ways to calculate a-loss:

1. KL-penalty:
$$L^{KL}(\theta) = \hat{E}[\frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda KL[\pi_{old}|\pi_\theta]]$$
2. Clipped surrogate objective:
$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta) * \hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon) * \hat{A}_t)]$$

Both implementation of a-loss can work well in real test. But as for me, I prefer Clipped method. Because compared to KL-penalty method, it's more simple to implement and it's compatible with Gradient Descent. What's more, according to the original paper [9], clipped method performs better than KL-penalty in average normalized score.

## 4 EXPERIMENTS

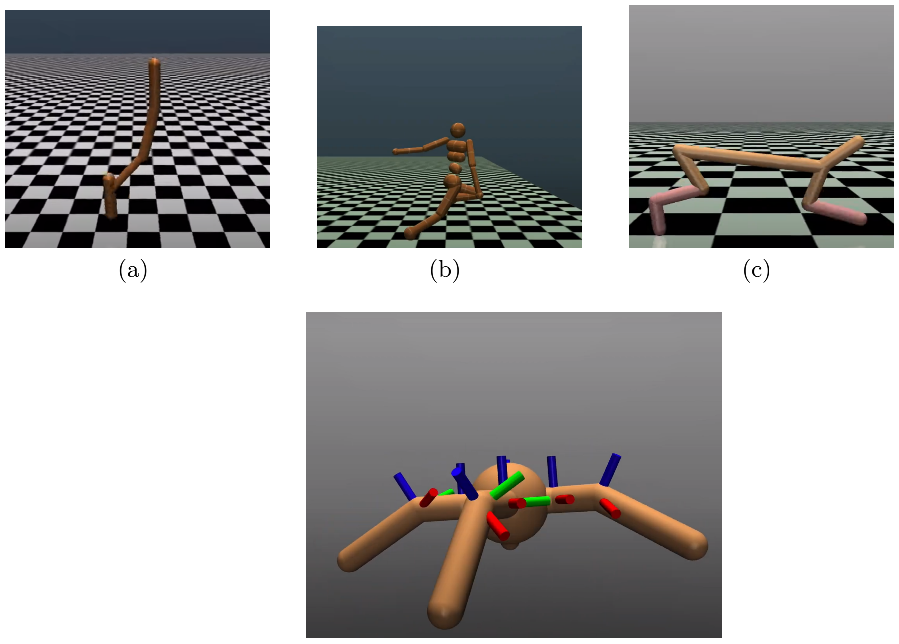
### 4.1 Mujoco

MuJoCo stands for Multi-Joint dynamics with Contact, which is a physics engine aiming to facilitate research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed. In this project, I use the following four Mujoco simulation experiments:

- Hopper-v2: Actions shape: (3,); Observations shape: (11,)
- Humanoid-v2: Actions shape: (17,); Observations shape: (376,)

- HalfCheetah-v2: Actions shape: (6,); Observations shape: (17,)
- Ant-v2: Actions shape: (8,); Observations shape: (111,)

These simulation environments displays as below:



(a) (b) (c) (d)

Fig. 2: Mujoco Simulation Environments

### 4.2 DDPG

In this section, I trained DDPG and it's improvement (DDPG with exploration) on Ant-v2, HalfCheetah-v2, Hopper-v2 and Humanoid-v2. For each environment, I fine-tune the training hyper-parameters by doing tons of serial experiments. And I will show the experimental results of several representative experiments and analyze the causes of the corresponding phenomenon.

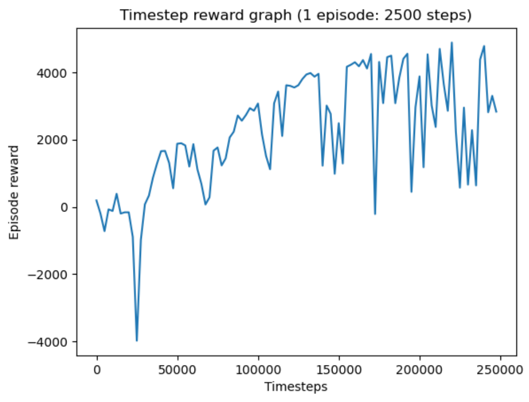
#### Ant-v2

In order to achieve a good result in Ant-v2, I trained DDPG and DDPG with action space exploration. The exploration is achieved by adding Gaussian distribution noise to each dim of the chosen action. The training results of both methods on the two environments is are shown in Figure 3. As we can see from the figures, DDPG and DDPG-explore worked quite good on Ant-v2 (the former can reach about 4200 reward per epoch in certain training states ). The average reward score of the last 20 epochs for DDPG and DDPG-explore reaches **2232.8** and **2085.4** separately when the step per epoch is set to 2500. Though the score of DDPG-explore is slightly lower than the DDPG, it converged more quickly and reached a more stable state. From the simulation window(the video

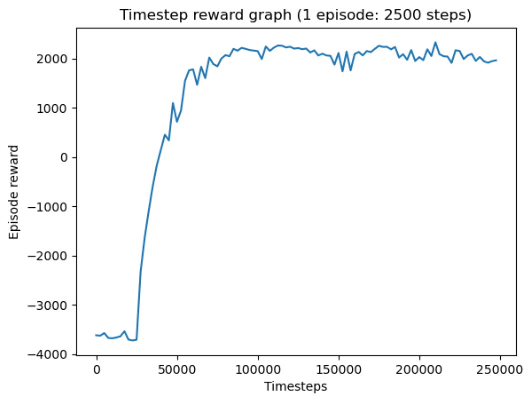
| **Hyper-parameters** | **Value** |
|---|---|
| memory-size | 100000 |
| hidden-size | 256 |
| LR-A | 0.0001 |
| LR-C | 0.001 |
| GAMMA | 0.99 |
| TAU | 0.005 |
| batch-size | 256 |
| epoch | 100 |
| start-step | 25000 |
| step-per-epoch | 2500 |
| use-explore | True/False |
| explore-decay | 0.999995 |
| reward-factor | 10 |

Table 1: DDPG Hyper parameters for Ant-v2

is uploaded to my Github) we can see that the ant agent with DDPG-explore brain walks more smoothly with its two front legs while on the other hand the agent with DDPG brain walks faster with 3 of its legs and keeps 1 leg in the air. However, some times the DDPG agent will stuck in the start point trying to raise one of its legs, which resulted in low reward in the end. Therefore, the results showed the power of action space exploration.



(a)



(b)

Fig. 3: DDPG in Ant-v2

### HalfCheetah-v2

This series of experiments show the importance of choosing a suitable **batch-size**. As we've mentioned before, DDPG updates its parameters by selecting a batch of experiences from memory each time. Thus, a appropriate batch size is significant for the DDPG model to reach a good optimal. As we can see from Fig 4, DDPG-v2 and DDPG-v3 worked quite good on HalfCheetah-v2 while the DDPG-v1 (with inappropriate batch size) did not get a good performance. The average reward

| Hyper-parameters | Value |
|---|---|
| memory-size | 1000000 |
| hidden-size | 256 |
| LR-A | 0.0001 |
| LR-C | 0.001 |
| GAMMA | 0.99 |
| TAU | 0.001 |
| batch-size | 64 |
| epoch | 200 |
| start-step | 5000 |
| step-per-epoch | 1000 |
| use-explore | True/False |
| explore-decay | 0.99998 |
| reward-factor | 1 |

Table 2: DDPG Hyper parameters for HalfCheetah-v2

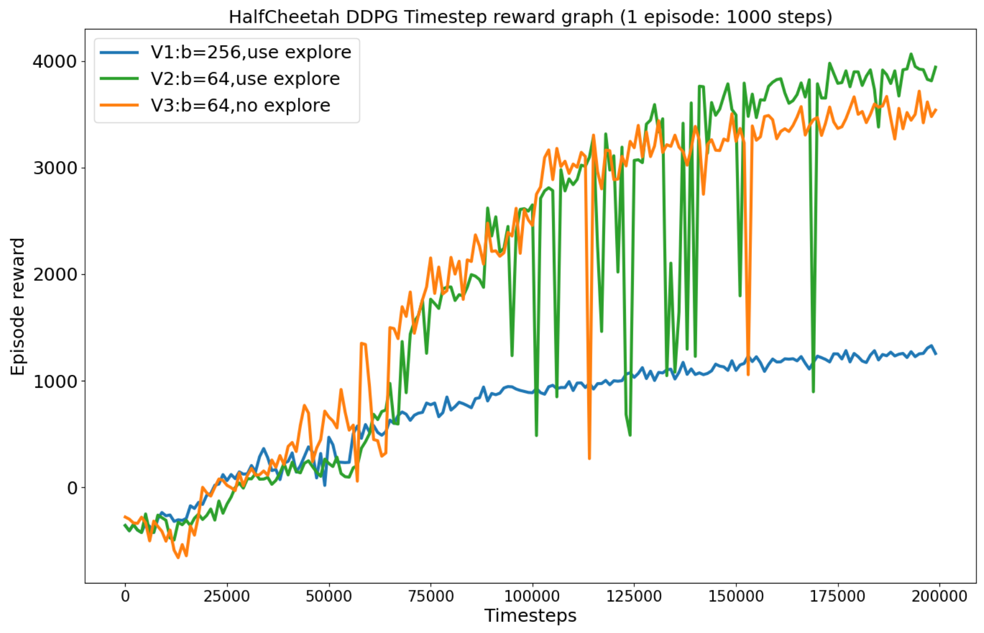
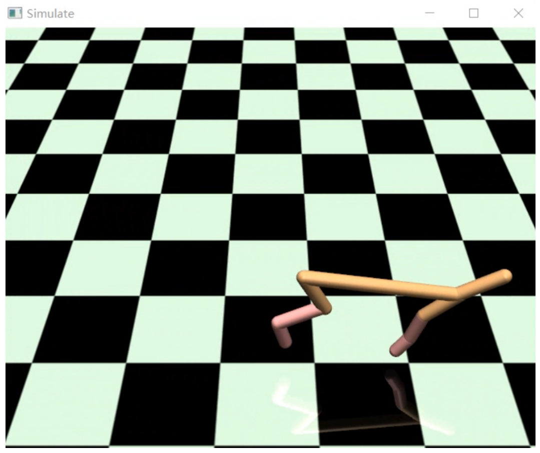
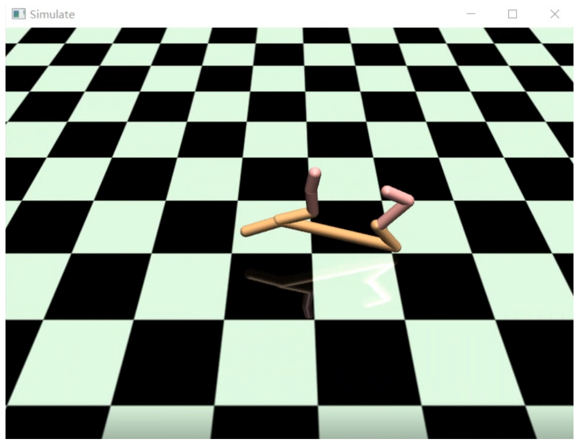


Fig. 4: DDPG in HalfCheetah-v2

score of the last 20 epochs for DDPG-v2 and DDPG-v3 reached **3848.4**, **3510.5** separately when the step per epoch is set to 1000. But for DDPG-v1 it can only reach **1245.3**. From the simulation window we learned that the agent v2 converged to a state where the HalfCheetah used its back to run. At the beginning of each epoch, the agent turned itself over immediately, which caused its pool performance. The results are shown in Fig 5



(a)



(b)

Fig. 5: (a) v2,v3 ; (b) v1

From figure 4 we also learned that it's helpful to enable the agent with exploration, which improved reward score by around 300. However, compared with a good batch-size, the impact of exploration seemed imponderable.

**Hopper-v2**

| **Hyper-parameters** | **Value** |
|---|---|
| memory-size | 100000 |
| hidden-size | 512 |
| LR-A | 0.0001 |
| LR-C | 0.001 |
| GAMMA | 0.99 |
| TAU | 0.001 |
| batch-size | 64 |
| epoch | 400 |
| start-step | 10000 |
| step-per-epoch | 1000 |
| use-explore | True/False |
| explore-decay | 0.99998 |
| reward-factor | 0.5 |

Table 3: DDPG Hyper parameters for Hopper-v2

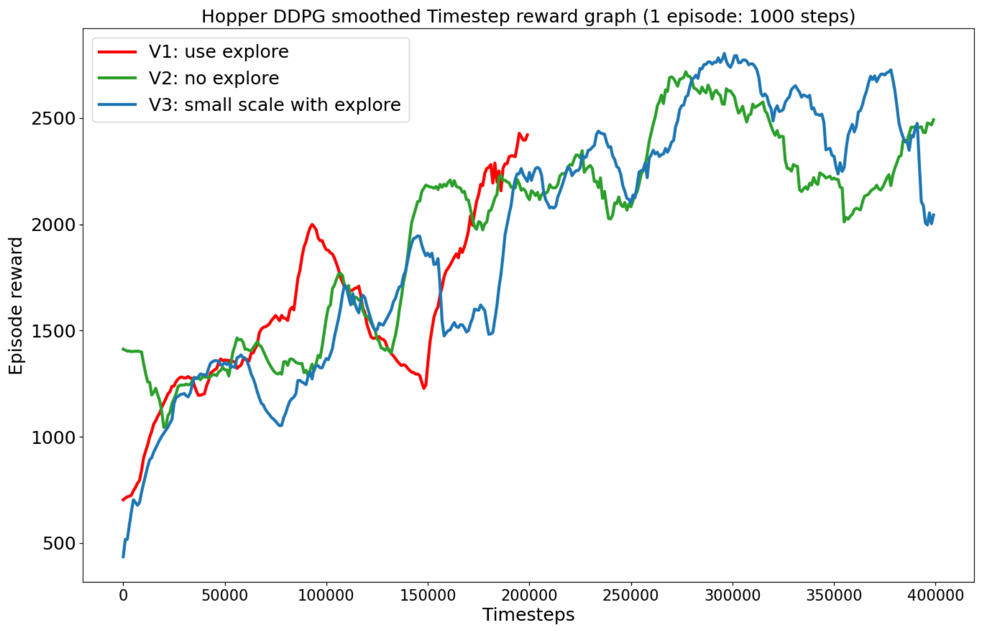


Fig. 6: DDPG in Hopper-v2

In this experiment, I'll show the effect of network scale. As we all know, network scale is very important for Machine Learning models (Reinforcement Learning is a subfield of Maching Learning) to fit the dataset. However, I found that in RL things were slightly different from traditional classification or regression settings, where the parameter scale must match with data scale: the former is larger will cause over-fitting; the latter is larger will cause under-fitting. However, in RL settings, the constraints are not as strict as before. From figure 6 we can see that once core parameters (e.g. batch-size, memory-size, LR-A, LR-C...) were set correct, network scale will not affect the results too much: both DDPG-v2 and DDPG-v3 achieved a good performance in HalfCheetah-v2. The average reward score of the last 20 epochs for DDPG-v2 and DDPG-v3 reached **2532.2**, **2011.6** separately when the step per epoch is set to 1000. And for the begining 200 epochs, we can see that DDPG-v1,v2,v3 all have the similar performance that their reward curves fluctuated in almost the same position. Actually after 400 epochs when all 3 of them converged, their performance are still similar: DDPG-v1 reach an average score of **2380.1**. Here I only presented the first 200 epochs of DDPG-V1 for the convenience to display the effect of network scale. From the simulation window we can see that the Hopper with DDPG brain successfully learn how to move forward by jumping (If you see the window in step 300000-320000 you can see it works well). However, the Hopper is too eager to move quickly that too much forward tilt caused it to fall to the ground. Unfortunately, using DDPG, the agent could not learn how to stand up again after falling, which resulted in a decline in the performance of the model in the later stage of training

**Humanoid-v2**

| Hyper-parameters | Value |
|---|---|
| memory-size | 100000 |
| hidden-size | 512 |
| LR-A | 0.0001 |
| LR-C | 0.001 |
| GAMMA | 0.99 |
| TAU | 0.005 |
| batch-size | 256 |
| epoch | 100 |
| start-step | 25000 |
| step-per-epoch | 5000 |
| use-explore | True/False |
| explore-decay | 0.999995 |
| reward-factor | 50 |

Table 4: DDPG Hyper parameters for Humanoid-v2

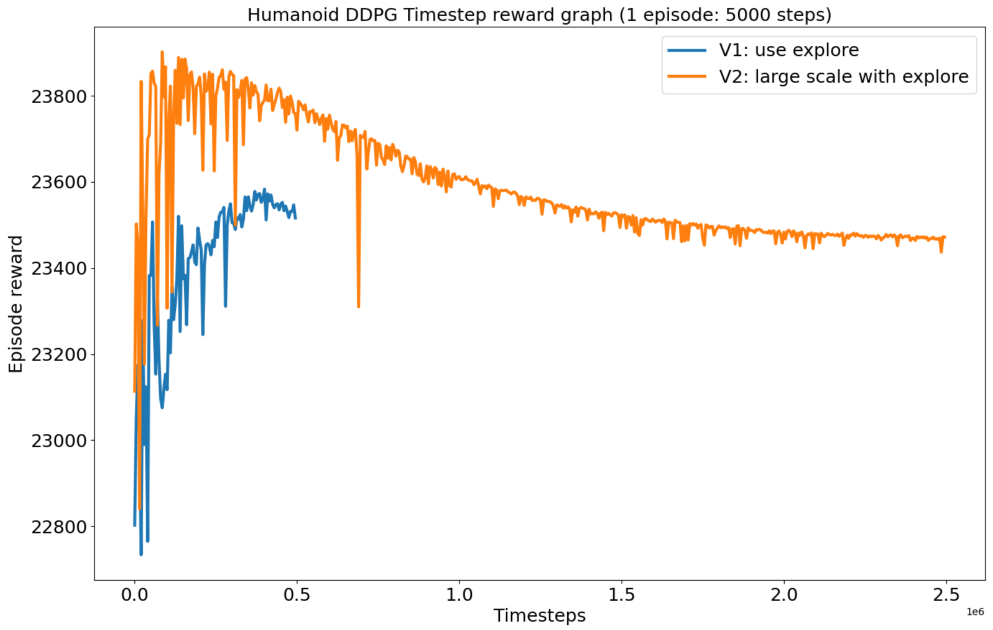


Fig. 7: DDPG in Humanoid-v2

Humanoid-v2 is the most complex one in Mujoco's simulation environments. For the fact that it has an actions shape: (17,) and Observations shape: (376,). From figure 7 we can see that both DDPG and DDPG-large-scale perform bad in this setting. Though I increased the network scale and training steps (hidden-size: 512 to 1024; hidden-layer: 1 to 2 in both Actor and Critic ;memory-size: 100000 to 1000000; Epochs: 100 to 500; explore-decay: 0.999995 to 0.999998) which made it converged, we can see that it reached a bad local optimal. No matter how I tune the hyper-parameter (including revise its reward-factor, add parameter-noise...) and how I change the network architecture, it seems that DDPG just can't adapt to this environment. It shows that the different environment will affect the algorithm somehow.

## 4.3 PPO

In this section, I trained PPO and it's improvement (PPO with advantage value normalization) on HalfCheetah-v2. For each environment, I fine-tune the training hyper-parameters by doing a lot of serial experiments. The experimental results of several representative experiments will be shown below. After that, the analysis of the causes for the corresponding phenomenon will be deployed.

**HalfCheetah-v2**

| **Hyper-parameters** | **Value** |
|---|---|
| hidden-size | 64 |
| LR-A | 0.0001 |
| LR-C | 0.001 |
| epoch | 50 |
| smooth-factor | 0.9 |
| memory-batch | 128 |
| done-step | 1000 |
| step-per-epoch | 5000 |
| use-explore | False |
| explore-decay | 0.99995 |
| reward-factor | 8 |
| action-update-steps | 20 |
| critic-update-steps | 20 |
| beta-low | 1/1.5 |
| beta-high | 1.5 |
| alpha | 2 |
| method-index | KL(0)/clip(1) |
| lam | 0.5 |
| kl-target | 0.01 |
| epsilon | 0.2 |

Table 5: PPO Hypter parameters for HalfCheetah-v2

This series of experiments show the effect of memory-batch size and action-update-steps, critic-update-steps. For the fact that in PPO, each episode the algorithm first run policy $\pi_\theta$ for memory-batch steps to collect experience $s_t, a_t, r_t$, then to update the Critic network and Actor network for critic-update-steps, action-update-steps separately. Thus, the memory-batch size affects the quality of sampling and model's ability of exploration. However, if the memory size is too large, the effective experience may be submerged in many invalid experiences, resulting in the majority of the available experiences. On the other hand, update-steps number affects the update extent of network

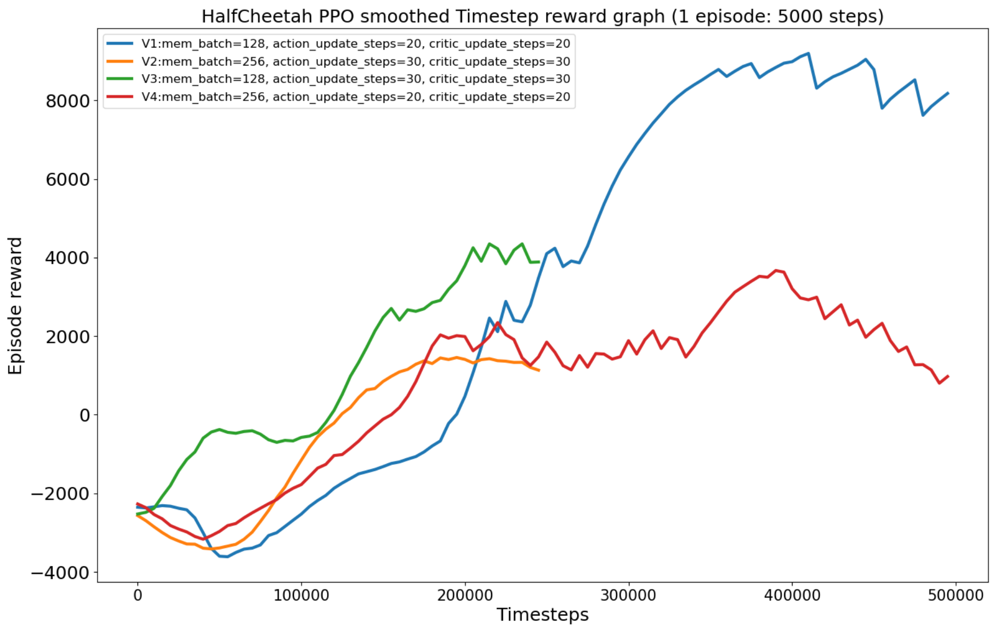


Fig. 8: PPO in HalfCheetah-v2

parameters. The larger the action-update-steps, critic-update-steps, the higher the exploration ability. Therefore, for HalfCheetah-v2 environment, we need a middle size memory-batch and a middle size update-batch.

As we can see from Fig 8, PPO-v1 and PPO-v3 worked quite well on HalfCheetah-v2 while the PPO-v2 and PPO-v4 (with too large memory-batch size) did not get a good performance. The average reward score of the last 20 epochs for PPO-v1 and PPO-v3 reached **8470.7**, **3429.6** separately when the step per epoch is set to 5000. But for PPO-v2 and PPO-v4 they can only reach **1280.6** and **180.8**. From the simulation window we learned that all the agents gained the knowledge of running, but PPO-v1, PPO-v3 ran faster than PPO-v2,PPO-v4. The result tells us about the good learning performance of PPO algorithm comparing with DDPG (no agents in PPO uses its back to run) and the importance to choose an appropriate batch-size (v1 improves the average reward score by around **7200** compared with v4) and update-step (v3 improves the average reward score by around **3200** compared with v4).

**Hopper-v2**

| **Hyper-parameters** | **Value** |
|---|---|
| hidden-size | 64 |
| LR-A | 0.0001 |
| LR-C | 0.0003 |
| epoch | 50 |
| smooth-factor | 0.9 |
| memory-batch | 128 |
| done-step | 1000 |
| step-per-epoch | 1000 |
| use-explore | True |
| explore-decay | 0.99995 |
| reward-factor | 8 |
| action-update-steps | 20 |
| critic-update-steps | 20 |
| beta-low | 1/3 |
| beta-high | 3 |
| alpha | 2 |
| method-index | KL(0)/clip(1) |
| lam | 0.98 |
| kl-target | 0.01 |
| epsilon | 0.2 |
| seed | 0 |
| render-threshold | 6000 |

Table 6: PPO Hyper parameters for Hopper-v2

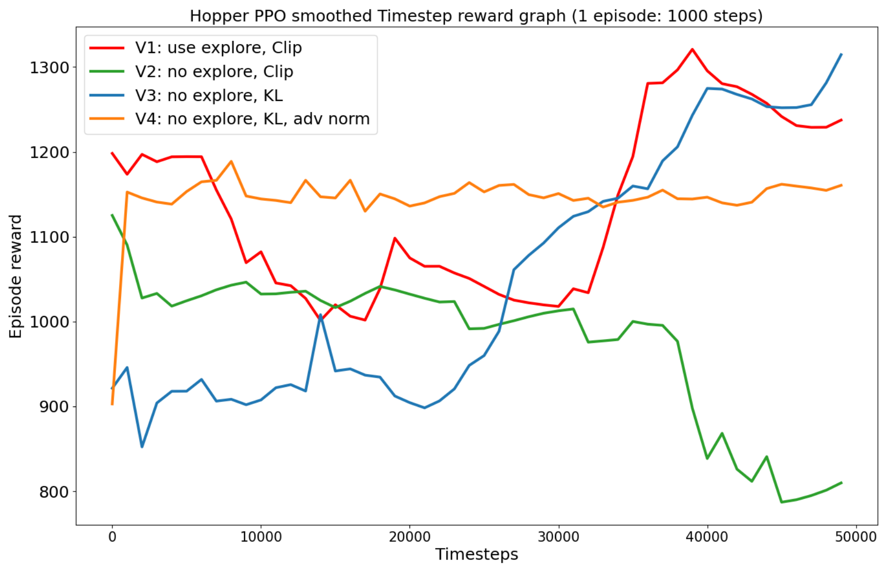


Fig. 9: PPO in Hopper-v2

According to the PPO paper [9], Clipped method performs better than KL-divergence method in most of the text settings. However, in Hopper-v2, I found the KL-divergence method is more suitable for this setting. From figure 9 we can see that PPO-v2 uses Clip method while PPO-v3 uses KL method and the gap between them is about **220** (v2 last 20 epoch: 809.7; v3 last 20 epoch: 1121.0). The reason may be explained from the mechanism of adaptive KL method:

1. When kl-mean is smaller than beta-low, it means that policy update is small, so we need to decrease the KL penalty item.
2. If kl-mean is larger than beta-high, it means that policy update is large, we need to increase the KL penalty item

We can also learn the effect of exploration by compraing the PPO-v1 curve and PPO-v2 curve. The average reward score of the last 20 epochs for PPO-v1 and PPO-v2 reached **1212.4**, **809.7** separately when the step per epoch is set to 1000. The action space exploration improves the average reward by around 400, which is larger than the improvement gained by using adaptive KL method. For PPO with advantage value normalization (PPO-v4), its performance is between PPO-v2 and PPO-v3 when none of them uses exploration. However, adding advantage value normalization makes model converge more quickly and reach a more stable state. From the simulation window we can see that the agent crawling forward steadily.

## 5 Conclusion

Above all, we can conclude that DDPG is quite powerful to control games like Ant-v2, HalfCheetah-v2 and Hopper-v2. It achieve an average training score of 2232.8, 3848.4 and 2532.2 separately. Its improvement DDPG-explore with action space exploration is more powerful and can help to find global optimal solutions. However, DDPG can't perform well with complex environments, for example, it works poorly in Humanoid-v2. What's more, DDPG is sensitive to the batch-size that too larger batch size may lead the agent learn in a wrong way (e.g. HalfCheetah uses its back to run). PPO is a famous reinforcement learning algorithm for its powerful learning ability. It worked quite well on HalfCheetah-v2 and achieved a score of 8470.7 when the step per epoch is set to 5000. And we explore the effect of different update methods (Clipped method, adaptive KL method) and whether to use advantage value normalization. However, all the algorithms mentioned in the project can be affected by the differences among environments slightly or enormously. It means we need to adjust our algorithm in different environments and also carefully choose algorithms.

## References

1. David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller.Deterministic policy gradient algorithms. In Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32, ICML14, pp. I387I395. JMLR.org,2014.
2. George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. Physical review, 36(5):823, 1930.
3. Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger (eds.), Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, volume 48 of JMLR Workshop and Conference Proceedings, pp. 1995–2003. JMLR.org, 2016.

4. Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and DaanWierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun (eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016.
5. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. nature, 518(7540):529–533, 2015.
6. Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double qlearning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI16, pp. 20942100. AAAI Press, 2016.
7. Schaul, T.; Quan, J.; Antonoglou, I. Silver, D. (2015), 'Prioritized Experience Replay' , cite arxiv:1511.05952Comment: Published at ICLR 2016.
8. Volodymyr Mnih, Adri'a Puigdom'enech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. CoRR, abs/1602.01783, 2016.
9. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A. Klimov, O. (2017), 'Proximal Policy Optimization Algorithms.', CoRR abs/1707.06347 .
10. Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M. I. Moritz, P. (2015), Trust Region Policy Optimization., in Francis R. Bach David M. Blei, ed., 'ICML' , JMLR.org, , pp. 1889-1897.
11. Wang, Z.; Bapst, V.; Heess, N.; Mnih, V.; Munos, R.; Kavukcuoglu, K. de Freitas, N. (2016), 'Sample Efficient Actor-Critic with Experience Replay.', CoRR abs/1611.01224 .