|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Network | Policy | Action Space | Operator |
| DQN family | Critic | Off-policy | Discrete | Q-value |
| DDPG | Actor-Critic | Off-policy | Continuous | Q-value/Policy Gradient |
| PPO | Actor-Critic | On-policy | Discrete/Continuous | Policy Gradient |

# DDPG

DDPG combines DPG (Silver et al. 2014) with DQN (Mnih et al., 2015) as a model-free off-policy actor-critic algorithm for continuous actions environment (Lillicrap et al., 2015). Lillicrap et al. (2015) extend DDPG from the discrete DQN into continuous action spaces and utilize the actor-critic network to estimate the deterministic policy model's action value. Basically, the idea of DDPG is that it trains the two-target network: Actor-Network for action selection and Critic Network for action value estimation. The Critic Network will learn and estimate the action value and then connect to Actor-Network to follow optimal gradient. Critic Network and Actor Network work together as following:

1. Actor Network outputs action based on observation received from environment
2. Environment returns rewards of received action also next observation
3. Update Critic Network based on rewards, then update Actor Network

The advances of DDPG over DPG in Lillicrap et al. (2015) are as follows: first, by adopting DQN, DDPG incorporates replay experience and two target networks to stabilize and robust learning process; second, it needs fewer steps to obtain the solution, implying easy-to-apply in more extensive and higher dimensional complex tasks. The article also shows DDPG needs less experience to solve Atari games. Due to all the reasons, we also implement DDPG in our empirical study. But a drawback of DDPG mentioned in the article is that due to the model-free nature, DDPG needs more training episodes to obtain the result.

Another important innovation in DDPG worth mentioning is the soft target update, which is similar to the solution of divergence and instability issue in Mnih et al. (2013) but applies to the actor-critic network. Soft target update means using copy networks to separately calculate action value, then update the target network slowly but frequently to keep stability. Lillicrap et al. (2015) also show that the soft target update generates more efficiency at the cost of slow learning.

## Empirical Study

Since the DDPG is designed to solve problem in continuous action space, we’ve chosen “LunarLanderContinuous-v2” environment to perform empirical study. Same to the discrete action space version of Lunar Lander game, the objective is to land lunar lander in landing pad with possible high score, but the input action changed from 1 of 4 discrete actions into a 2 values vector, ranged from -1 to +1. The first value controls main engine power and the second controls left and right engines.

## Setup

DDPG has many hyperparameters, most important are followings:

* **Gamma**: this parameter affects the weight of rewards from next steps. We expect that the agent considers not only the reward from next steps but also from longer future. The smaller this value, the higher the weight of next few steps. But also, the longer future we take into consideration, the lower the quality of prediction. The balance between them needs to be made, so we set this value to 0.99.
* **Tau**: DDPG uses “soft update” slowly in every step instead of “hard update” after every X steps as in DQN. This value defines how “soft” the update will be. The smaller the value, the “softer” the update. We set it to 0.001.
* **Memory size**: DDPG stores previous experiences and sample from them to train networks. Once memory size is exceeded, we will replace oldest experiences with new ones. We set it to 2 \*\* 17.
* **Batch size**: How many samples we will use to train the model. The lower this value, the faster the training speed, but it also possible to reduce the convergence speed (new more episodes to converge). We set it to 128.

Besides of those DDPG related parameters, we also used following network parameters:

* Actor network:
  + 2-layers Neural Network
    - 512 hidden units in 1st layer
    - 256 hidden units in 2nd layer
  + Learning rate: 5e-5
* Critic network:
  + 2-layers Neural Network for state input
    - 512 hidden units in 1st layer
    - 256 hidden units in 2nd layer
  + 1-layer Neural Network for action input
    - 128 hidden units
  + 1-layer Neural Network with concatenated outputs from state part and action part as input
    - 128 hidden units
  + Learning rate: 5e-4

In noise for exploration part, we used 2 types of noise:

* Ornstein-Uhlenbeck (OU) Noise mentioned in DDPG original work Lillicrap et al. (2015)
* Gaussian Noise

For both noises, the mean is set to 0 and standard deviation is set to 0.2.

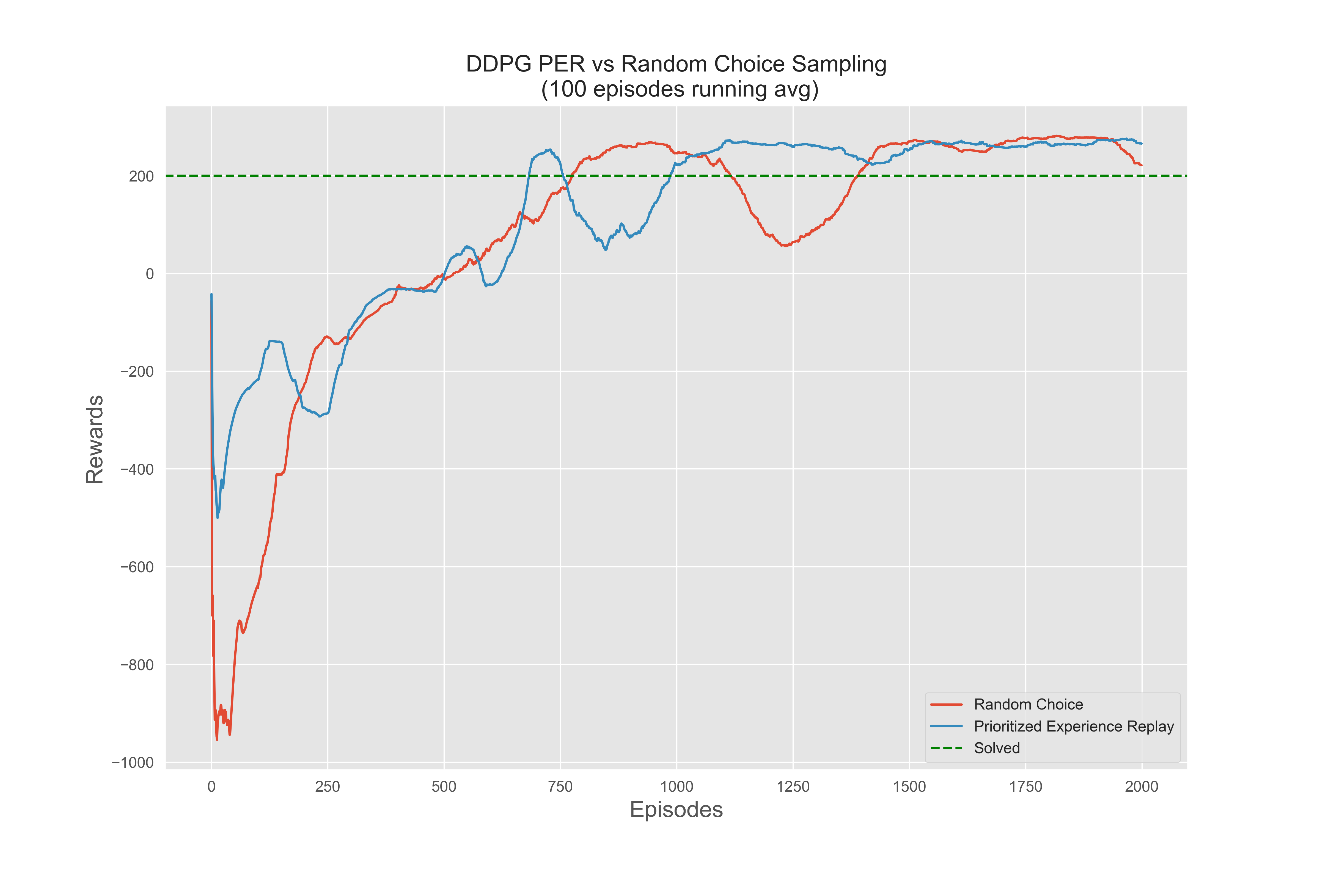
The algorithm is implemented with Tensorflow V2 and executed in Google Colab. Also, to keep all tests possibly equal, we set the environment seed to 123 for all runs.

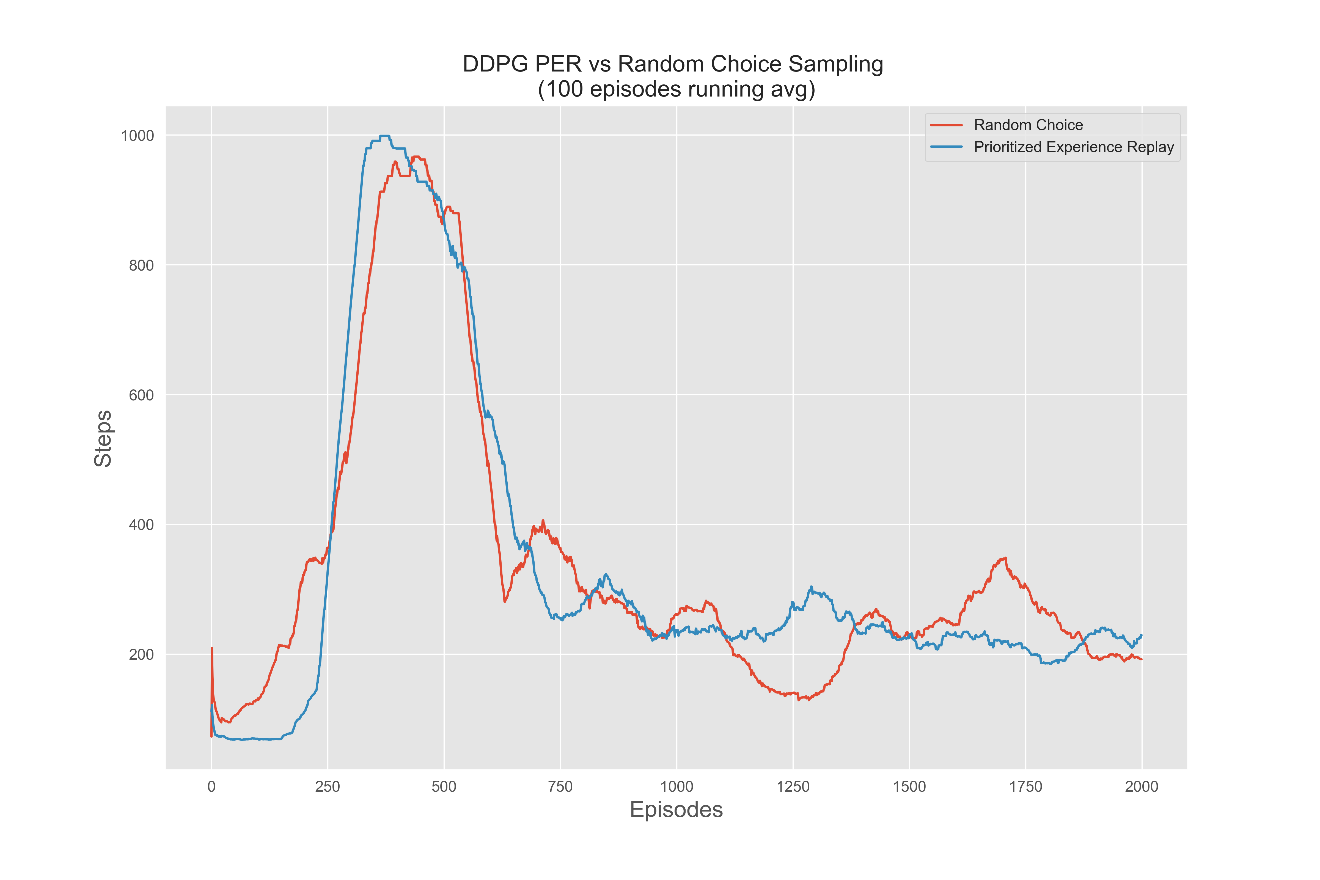
## Results

One major objective of this empirical study is to understand the performance of prioritized experience replay (PER). In original DDPG algorithm, the method of sampling from stored experience is uniformed random sampling, which means every experience in memory has the same probability to be selected. But as common sense tells us, not all the experiences are same important for learning, if we are able to find most helpful experiences, it would be helpful to reduce learning time. PER used TD error to measure importance of experiences and prioritize them while sampling. According to authors (T.Schaul et al. 2016), the PER is able to speed up the convergence and achieve a better performance comparing to DQN using uniform sampling experience replay. Since the Critic part of DDPG is almost the same as DQN, can PER improve the convergence speed and performance in DDPG?

As the rewards figure shows, PER did quickly increase rewards in first 200 episodes, but since the 250th episode, those 2 methods kept almost the same pace in scoring. Although PER version reached Solved (*scored more than 200 point in an episode is considered as problem solved*) line earlier than Random Sample version, it’s not always the case in later runs. So, we cannot prove that PER brings better performance.

Since DDPG updates networks every step, using less steps could reduce the training time and speed up convergence. While checking average steps chart, we can find that, in first 250 rounds, PER version did use less steps than Random Sample version. But for the whole training period, Random Sample version used only 0.2% less steps than PER version. As both models achieved stable performance after 1500 episodes, if we take in consideration only first 1500 episodes, Random Sample version used 3.7% less steps the PER version. In this case, we cannot find enough evidence to support the assumption that PER would bring better performance and convergence speed to DDPG algorithm.

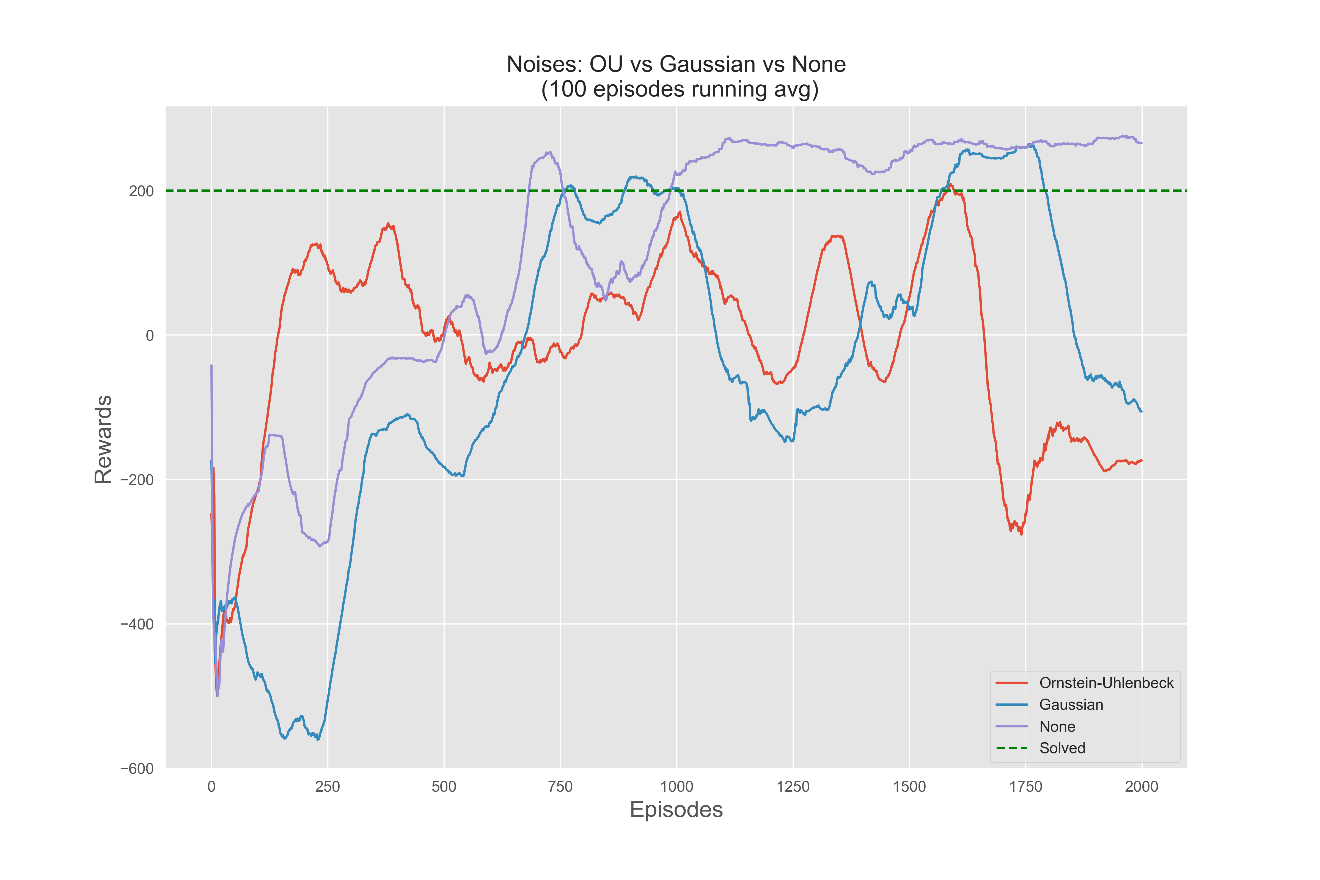




Another objective is to test the performance of different action noises. As mentioned in DDPG original paper Lillicrap et al. (2015), authors used Ornstein-Uhlenbeck Noise in exploration. Is this a better choice than Gaussian noise? What if we don’t use noise in exploration at all?

Described in DDPG paper, OU noise is temporally correlated exporation for exploration efficiency in physical control problems with inertia and it explores well in physical environments that have momentum. One major difference between OU noise and Gaussian noise is that the absolute difference between each continuous two steps in OU noise are not as large as in Gaussian noise. Also, continuous steps have trend to follow the direction of momentum. Landing control in our environment is exactly the system as described: each action will generate feedbacks with inertia.

From Noise graph, we can see that all noise versions were not able to converge, although they all reached solved line, but the performance is not stable at all. Among all three versions, the one has the best performance did not use noise. And OU noise has an even worse performance than Gaussian noise. After multiple rounds of tests, we found that, without noise, each time the model can achieve convergence and get a stable performance. On the contrary, neither OU nor Gaussian noise can get similar results with our configuration (mu=0, sigma=0.2). Due to the usage limit of Google Colab, we were not able to run more tests with different parameters, it could be an interesting subject in future research.



PPO

Since published, Proximal Policy Optimization (PPO) have been using as default Deep Reinforcement Learning algorithm by OpenAI, and the 1st choice for many reinforcement learning problems because of its simplicity, powerful performance, quick training speed and versatility.

Since the core of policy gradient is to increase the probability of “good” samples and reduce the probability of “bad” samples. So, a common problem that all policy gradient (PG) algorithms are facing is the instability of training caused by the sparseness of rewards in experience and sampling. Imaging for a sample occasionally having a very high rewards, its probability will be increased rapidly and for one having a very low rewards, its probability will be decreased rapidly, this situation generates instability of training.

To increase the stability, different PG algorithms use different methods to reduce the frequency of updates or to limit changes between updates, for example the “soft update” we saw in DDPG. The method used in TRPO is to do an optimization constrained by KL divergence through conjugate gradient, which limits changes between updates. But this method requires to calculate second derivative of KL divergence, which increases the complexity and could use many resources. PPO made an improvement on this method by moving the constraints into loss function, which speeds up computation and reduce computation needs. The optimization in PPO can be described as following:

* If the new policy is improving towards the right direction, once over a certain threshold, the algorithm will stop optimizing the new policy. But if the new policy is getting worse, the algorithm will keep optimizing it.
* If the new state value is closer to target than old one, and over a certain threshold, the algorithm will stop optimizing the state function. But if contrary, the algorithm will keep optimizing it.

## Empirical Study

PPO algorithm works for both discrete and continuous action space, we keep using “LunarLanderContinuous-v2” environment to perform the empirical study. Details of this environment was described in DDPG section.

## Setup

Compared to DDPG, PPO has less hyperparameters, most important are followings:

* **Lambda advantage**: Similar to Gamma in DDPG, this parameter affects the weight of rewards from next steps. Smaller Lambda advantage brings more weights to next few steps. We set this value to 0.99 as in DDPG.
* **Clip epsilon**: this parameter limits the range of changes. As described in introduction, PPO use this parameter as threshold to limit amplitude of updates. We set this value to 0.2.
* **Number of updates**: this parameter controls the number of updates each episode, we set this value to 10.

By the way, although PPO is an on-policy algorithm, it’s still able to learn from experiences from previous episodes. But to compare with off-policy algorithms, we forced PPO only to learn from current episode, which means we don’t maintain an experience memory.

Besides of those PPO related parameters, we also used following network parameters:

* Actor network:
  + 2-layers Neural Network
    - 512 hidden units in 1st layer
    - 512 hidden units in 2nd layer
  + Learning rate: 1e-5
* Critic network:
  + 2-layers Neural Network
    - 512 hidden units in 1st layer
    - 512 hidden units in 2nd layer
  + Learning rate: 1e-3

The algorithm is implemented with Pytorch V1.81 and executed in Google Colab. Also, to keep all tests possibly equal, we set the environment seed to 123 for all runs.

## Results

The major objective of this empirical study is to compare PPO and DDPG, from performance, execution time and other prospects.

First, on comparing the execution time, we found PPO is much quicker than DDPG. In general, it takes PPO 30-40 mins to complete 2000 episodes, but DDPG needs at least 2 and half hours to finish the same run.

Second, from comparison graph we can see that, PPO’s rewards curve increases much more smoothly than DDPG runs. Although DDPGs achieved higher average scores in the last 1000 episodes, but it’s not always the case in different runs. Also, PPO started to increase the average rewards at the very beginning compared to DDPG algorithms and didn’t touch a very negative rewards as DDPGs did during the entire run, which reveals the stability of the algorithm.

Third, although the Lunar Lander games limits by default the maximum steps to 1000 in an episode, we think this value is too high. Although higher steps limit may encourage agent to explorer more, we don’t want a dawdling spaceship to waste much time in exploration. Also, we noticed that, for both PPO and DDPG, the number of used steps per episode increases rapidly in the first half of training, which means the agent was trying to stay in the air to avoid crash penalty, but by the end of training, this number always reduced to less than 300 steps, which means the agent learned that quickly landing on the correct zone is a better strategy.

We designed test runs with different maximum steps limit. From the figure we can see that, maximum 300 steps run shows the least velocity among all tests, but both other two runs reached “resolved” line earlier than it. The fact shows that limiting maximum steps is indeed helpful to achieve a better performance.

