Reproducibility report of ”Continuous Control with Deep Reinforcement Learning”

20337025 Cui Canming

January 9, 2023

Reproducibility Summary

Scope of Reproducibility

In this paper, I attempt to verify the claims of original paper that DDPG, that is, DPG with target networks, perform better in continuous control tasks continuous control tasks. Further, The author claims that DDPG also has outstanding performance in other aspects. I use the code I wrote according to the paper to replicate several experiments in the original paper and draw conclusions based on these results.

Methodology

I analyzed the model of the paper in detail, and gave the implementation process of the algorithm. This is of great help to my reproduction work.

Results

We validated each of the author’s claim through the experiments given in the original paper and few additional experiments of our own. Overall, we found many experiments yielding identical results. Although the experimental environment cannot be exactly the same as that in the paper, similar experimental results have been obtained in similar experimental environments.

What was easy

Overall, detailed parameter settings, simplified algorithm description and recurrence work of others proved valuable to validate the authors’ claim.

What was difficult

Many MuJoCo experimental environments in the paper were built by the authors themselves, and the authors did not provide building methods or building tutorials. The lack of computing power requires me to spend a lot of time on an experiment.

1 Introduction

In recent years, deep reinforcement learning (DRL) has shown great promise as a method for solving a wide range of control tasks, including tasks that involve continuous, real-valued actions. However, these tasks present unique challenges, as the action space is often much larger than in tasks with discrete actions, and it is difficult to design effective exploration strategies. In this paper[LHP+15], the authors propose a new approach to continuous control with DRL, using actor-critic methods. Actor-critic methods involve training two neural networks: an "actor" network that proposes actions to take in a given state, and a "critic" network that evaluates the proposed actions and provides feedback. They demonstrate the effectiveness of their approach on a range of continuous control tasks, including controlling the movement of robots and simulating physical systems. The results show that their approach outperforms state-of-the-art methods on these tasks.

The model-free approach proposed by the authors is called Deep DPG (DDPG). Authors constructed simulated physical environments of varying levels of difficulty to test the algorithm. This included classic reinforcement learning environments such as cartpole, as well as difficult, high dimensional tasks such as gripper, tasks involving contacts such as puck striking and locomotion tasks such as cheetah (Wawrzy´nski, 2009). They compared the performance of DDPG on these tasks with other algorithms, and drew relevant images and tables.

In this report, I reproduced the DDPG model proposed in the paper based on OpenAI Gym and Tensorflow according to the algorithm content, and verified the author's statement in the same or similar environment. In this work, I will reproduce the conclusive experimental values and image results given in the paper, and try to analyze and expand them.

2 Scope of reproducibility

In this report, we investigate the following claims from the original paper:

1. Adding the target network or batch normalization to the DPG is necessary to perform well across all tasks. In particular, learning without a target network, as in the original work with DPG, is very poor in many environments. To verify this, I will run the same task with my own DDPG model and DPG model, and compare their performance, as in the paper.

2. In some simpler tasks, DDPG can learn strategies from pixels as quickly as using the low-dimensional state descriptor.

3. In terms of comparing the value of Q estimation after training with the real return seen in the test set, DDPG estimates returns accurately without systematic biases. in simple tasks. For harder tasks the Q estimates are worse, but DDPG is still able learn good policies.

I attempt to reproduce the experiments from the paper[LHP+15]and perform exploratory analysis on the above mentioned claims. However, due to different experimental environments and software versions, some experimental results may not be completely restored. I will also explain in detail why these parts are difficult to reproduce.

3 Methodology

The work of TP Lillicrap et al. combines insights from recent advances in deep learning and reinforcement learning, resulting in an algorithm(DDPG) that robustly solves challenging problems across a variety of domains with continuous action spaces, even when using raw pixels for observations.

3.1 Model descriptions

The architecture of DDPG is similar to Actor-Critic. DDPG can be divided into two major networks: the policy network and the value network. The actor network takes in the state of the environment and outputs an action. The critic network takes in both the state and action of the environment and outputs a prediction of the corresponding Q-value, which represents the expected return for taking a particular action in a particular state. DDPG continues DQN's idea of fixed target network, and subdivides each network into target network and real network. However, the update of the target network is somewhat different.

First, let's look at the policy network, namely Actor. Actor outputs a deterministic action. The network that generates this deterministic action is defined as $a = \mu_\theta(s)$. In the past, the policy gradient adopted a random strategy, and each acquisition action required sampling the distribution of the current optimal strategy, while DDPG adopted a deterministic strategy, which was directly determined by function μ. Actor's estimation network is $\mu_\theta(s)$, θ is the parameter of the neural network, which is used to output real-time actions. In addition, Actor also has a target network with the same structure but different parameters, which is used to update the value network Critic. Both networks are output action.

Then look at the value network, namely Critic. Its function is to fit the value function $Q_w(s, a)$. There is also an estimation network and a target network. The two networks both output q-value of the current state at the output end, but differ at the input end. Critic's target network input has two parameters, the observation value of the current state and the action action of the Actor's target network output. The input of Critic's estimation network is the action of the current Actor's estimation network output. The target network is used to calculate Q_{target}

The whole process is shown in Figure 1. We can see that the update of the value network is based on the gradient descent of TD error. As a judge, Critic did not know whether the action output by the Actor was good enough at the beginning. It also needs to learn to give an accurate score step by step. Therefore, with the help of the value Q_w of the target network fitting at the next moment and the real reward r, we can get Q_target. Let Q_target subtract the current Q to get the mean square error, and then we can construct Loss. The updating method is similar to that of DQN. The only difference is that the parameters of the target network are updated slowly in DDPG algorithm, rather than directly copying the parameters of the existing network every N steps in the DQN.

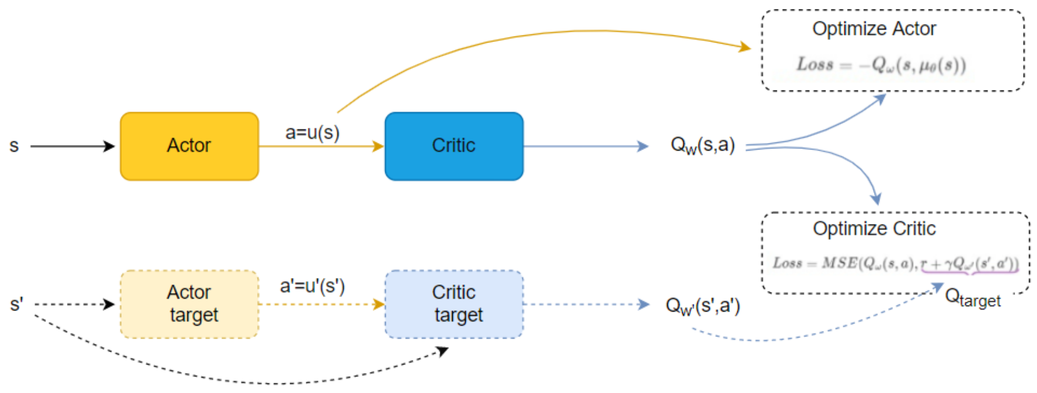


Figure 1: The whole update process.

The update of the policy network (Actor) is based on the gradient ascent, because the goal of the Actor is to find an action a that can maximize the value Q of the output, so the gradient of the optimization policy network is to maximize the value Q of the output of the value network. The Loss function adds a minus sign to minimize the error.

It is worth mentioning that DDQG also draws on the experience replay skills of DQN. DDPG will also store the sequence (s, a, r, s') of a period of time to the experience buffer. During each training, randomly sample a minibatch from the experience buffer to train.

3.2 Algorithm flow

The pseudo code of the algorithm is is shown in *Algorithm 1*. First, initialize Actor, Critic and their respective target networks, a total of four networks and the experience pool replay buffer R.

When the Actor network outputs actions, DDPG implements exploration by adding random noise, which allows agents to better explore potential optimal strategies. Then the skill of experience playback is adopted. Store the data (s_t, a_t, r_t, s_{t+1}) that the agent interacts with the environment in R. Then, a minibatch is randomly sampled from each training.

In terms of parameter update, the critical target network Q' is used to calculate the target value y_i, and use the mean square error of y_i and the current Q value to construct a loss function for gradient update. For Actor's policy network, it is actually to substitute Actor's deterministic action function into Q-function's a, then calculate the gradient, and finally update the target network.

In short, DQN + Actor-Critic = Deep Deterministic Policy Gradient (DDPG). In fact, DDPG is closer to DQN, but it adopts a structure similar to Actor Critic. DDPG absorbs the advantages of single step update of policy gradient in Actor-Critic, and also absorbs the skills of DQN in estimating Q value. The biggest advantage of DDPG is that it can learn more effectively in continuous action.

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:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{\mu'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{\mu'}$$
```

3.3 Code and Experimental setup

The experimental environment is as follows:

- python 3.7.15
- gym[BCP+16] 0.25.2
- tensorflow 1.13.1

The code implementation refers to the works of MOCR[MOC14] and Floodsung[flo14]. The reproduced code includes 12 python files. The code structure is shown in Figure 2. I have also open-sourced the code, which you can find on `https://github.com/91Mrcui/RE_DDPG.git`

Main.py is the entrance of the whole program, where we can see how the program is executed and conduct global settings and training. DDPG.py, DDPG_BN.py and DPG.py are agents, including actor networks and critic networks, and both use replay buffer in memory.py. The noise.py adds noise to the actions.

Next, take main.py as an example to introduce the whole process of running the program. The code of main.py is as follows:

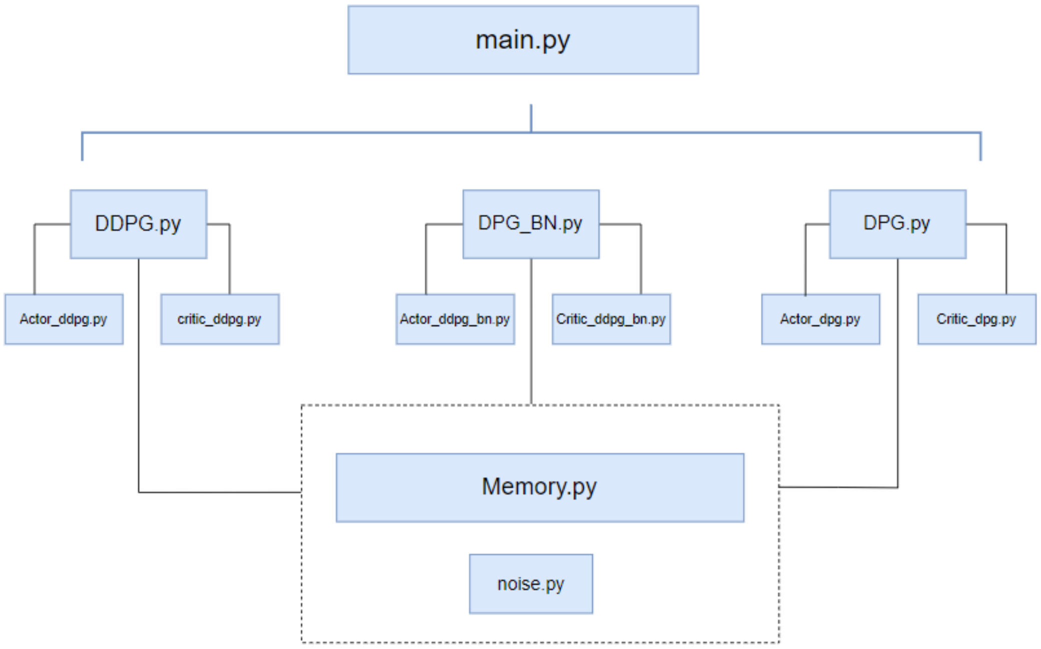


Figure 2: The structure of code.

```
import ddpg
import gc
import gym
import numpy as np
gc.enable()
ENV_NAME = 'Hopper-v4'
EPISODES = 10000
TEST = 10

if __name__ == '__main__':
    # Record the reward to draw the image
    reward_record=open('rw_ddpg.txt',mode='a')
    reward_record.writelines(ENV_NAME+'\n')

    env=gym.make(ENV_NAME)
    env=env.unwrapped

    agent = ddpg.DDPG(env)

    for episode in range(EPISODES):
        state = env.reset()

        # Training
        for step in range(env.spec.max_episode_steps):
            action = agent.noise_action(state)
            next_state,reward,done,_,_ = env.step(action)
            agent.perceive(state,action,reward,next_state,done)
            state = next_state
            if done:
                break
        # Testing:
```

```
        if episode % 100 == 0 and episode > 100:
            total_reward = 0
            for i in range(TEST):
                state = env.reset()
                for j in range(env.spec.max_episode_steps):
      #env.render()
                    action = agent.action(state) # direct action for
                                                         test
                    state,reward,done,_,_ = env.step(action)
                    total_reward += reward
                    if done:
                        break
            ave_reward = total_reward/TEST
            print('episode: ',episode,'Evaluation Average Reward:',
                                             ave_reward)
            #print(str(ave_reward))
            reward_record.writelines(str(ave_reward)+'\n')

    env.close()
    reward_record.close()
```

First define the experimental environment ENV_NAME and the number of training rounds, and then open the file recording the reward. During the training, record the rewards to draw an image. Then call function gym.make to create the experimental environment and call ddpg to create the agent. Then start training, test every 100 rounds, record the reward and output relevant information.

The three files, ddpg.py, dpg.py and ddpg_bn.py define their corresponding agent classes. dpg.py is the original DPG without the target network, ddpg.py uses the target network, ddpg_bn.py adopts target network and batch normalization. Next, take DDPG.py as an example to give a brief introduction. ddpg class are defined in DDPG.py. I define the following functions in the ddpg class:

- init(env): This function is the constructor for the DDPG class and is called when an instance of the class is created. It takes an env parameter which is the environment in which the agent will interact and learn. The function initializes the actor and critic networks, the replay buffer, and the exploration noise process.

- train(): This function trains the actor and critic networks using a batch of experiences sampled from the replay buffer. It calculates the target values for the critic network and updates both the actor and critic networks by minimizing the loss between the predicted and target values. It also periodically updates the target networks for both the actor and critic.

- noise_action(state): Selects an action for the agent to take in the current state, using the current policy and adding exploration noise to the action.

- action(state): Selects an action for the agent to take in the current state, using the current policy without adding exploration noise.

- perceive(state,action,reward,next_state,done): This function is used to put experience (state, action, reward, next_state, don) into the replay buffer. When the size of the replay buffer reaches the set threshold REPLACE_START_SIZE, call the train() function to start training.

Actor.py and Critic.py define actor networks and critical networks respectively. Memory.py define ReplayBuffer class, which is used to store and manage the agent's experiences. The noise.py defines a class called OUNoise that is used to generate Ornstein-Uhlenbeck process noise.

See my github repositories for specific code implementation.

3.4 Hyperparameter search

In the details of the experiment, the author gives detailed settings in the paper, such as the network structure, the number of nodes in each layer of the network, the learning rate, and so on. And I found that the hyperparameters provided by the authors were stable, and so I did not conduct a hyperparameter search in this report. But I reduced the number of training to reduce the training time.

4 Results

A lack of compute power and the same MuJoCo[TET12] environment prevented us from testing on all the environments mentioned in the paper. Therefore, I have conducted experiments in some of the same MuJoCo environments and some of the MuJoCo environments not covered in the paper. For each game environment, the three types of agents train the same number of rounds, record the reward of each agent, and then draw images for comparison.

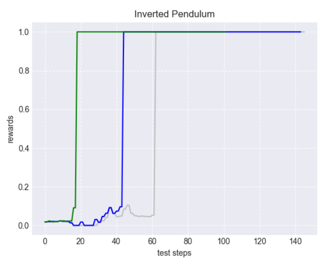
4.1 Results reproducing original paper

Does DDPG perform better in continuous control tasks continuous control tasks? - Figure 3 shows the performance curve for a selection of environments. We can see that adding the target network or batch normalization to the DPG is necessary to perform well across most tasks. In most environments, DDPG performs better than DPG, which can be seen from the convergence speed and reward value. In particular, learning without a target network, as in the original work with DPG, is very poor in many environment. In addition, the performance of DDPG with batch normalization is better than that of DDPG without batch normalization, which also verifies the experimental results in the paper.

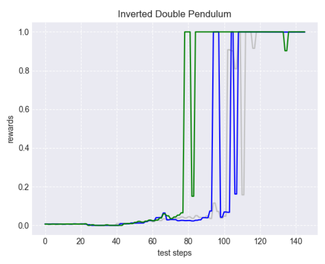
Table 1 summarizes DPG's performance and DDPG's performance across all of the environments.

4.2 Results beyond original paper

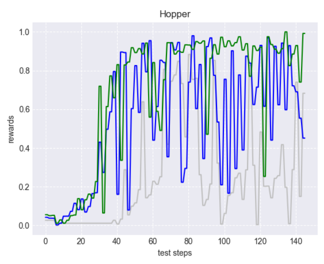
For Additional Result, I added some mujoco environments not in the paper as additional experiments. The experimental results can also be seen in Figure 3.



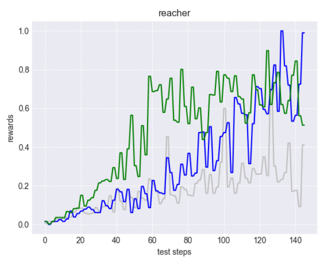
(a) Inverted Pendulum



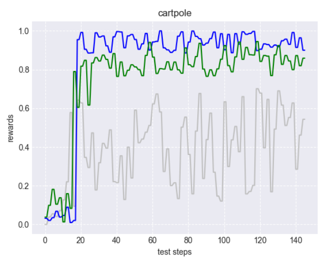
(b) InvertedDoublePendulum



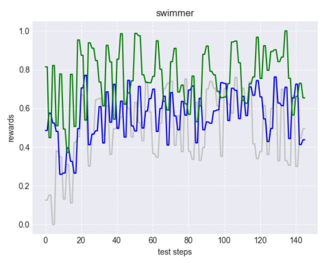
(c) Hopper



(d) Reacher



(e) Cartpole



(f) Swimmer

Figure 3: the performance curve for a selection of environments. original DPG algorithm (light grey), with target network (blue), with target networks and batch normalization (green). The reward is normalized.

environment	$R_{best,dpg}$	$R_{av,dpg}$	$R_{best,ddpg}$	$R_{av,ddpg}$	$R_{best,ddpg\_bn}$	$R_{av,ddpg\_bn}$
Inverted Pendulum	1000.0	591.18	1000.0	704.72	1000.0	829.29
InvertedDoublePendulum	9359.37	2817.55	9357.71	2985.02	9359.89	4332.59
Hopper	3168.46	895.21	3513.78	1939.61	3585.73	2422.57
Reacher	274.0	73.63	395.0	126.71	355.0	188.45
Cartpole	119.27	64.95	170.02	141.75	160.63	128.03
Swimmer	70.02	47.47	70.91	53.55	90.55	69.69

Table 1: Performance after training across all environments for at most 7,500 steps.

5 Discussion

5.1 What was easy

The author gives detailed parameters of the model, which is very helpful for my reproduction. Apart from this, It is easy to set up the experimental environment, although it takes me a lot of time. Reproducibility is a high degree of freedom, and you can use any python library to model your neural network.

In addition, the implementation of DDPG also has a lot of reference work on the Internet, which also provides a great help for me to reproduce the experimental results of the paper.

You can directly use MuJoCo tasks in the Gym environment for experiments without setting up your own. Although the number of environments is not comparable to the number in the author's experiment, it is still sufficient for experimental verification.

5.2 What was difficult

It is difficult to fully realize the experimental results in this paper, mainly due to the following problems:

1. Many MuJoCo experimental environments in the paper were built by the authors themselves, and the authors did not provide building methods or building tutorials. The number of MuJoCo environments in the gym environment is small, so it is difficult to conduct experiments on all environments to verify the performance of the model.

2. Although the architecture of the model is set to be the same as that given by the authors, some experimental results do not meet expectations, and even DPG performs better than DDPG.

3. During the experiment, I encountered a lot of inexplicable errors. For example, the MuJoCo environment can be used normally at first, but after a period of time, the system will report that the environment has not been found and needs to be reinstalled. I haven't solved this problem yet.

4. Because I started late, I still have a lot of work to do. I just did some experiments. In the future, we will almost conduct experiments in the environment where no experiments have been conducted.

5.3 Suggestions for reproducibility

In general, according to the parameter settings of the model in the paper and the experimental environment provided by the author, it is easy enough to generate results similar to those found in the paper. However, in the future, it may be helpful if the authors provide their own MuJoCo environment for use in the paper, or publish the source code of the experiment.

In addition, since I can not repeat all the author's experiments on the MuJoCo environment, I suggest adding additional experiments using the Atari environment to the report for the sake of reproducibility.

References

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[flo14] floodsung. DDPG. `https://github.com/floodsung/DDPG`, 2014.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[MOC14] MOCR. DDPG. `https://github.com/MOCR/DDPG`, 2014.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.