Reproducibility report of ”CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING”

Chen Yuyan

20. januar 2022

Indhold

1 Introduction

DQN algorithm is capable of human level performance on many Atari video games using unprocessed pixels for input. However, DQN can only handle discrete and low-dimensional action spaces. DQN cannot be straightforwardly applied to continuous domains since it relies on a finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.
In the paper, the authors present a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces. This algorithm which the authors call DDPG is the combination of DPG and DQN.
The goal in reinforcement learning is to learn a policy which maximizes the expected return from the start distribution $J = E_{r_i,s_i \sim E,a_i \sim \pi}[R1]$.
Many approaches in reinforcement learning make use of the recursive relationship known as the Bellman equation:

$$Q^{\pi}(s_t,a_t) = E_{r_t,s_{t+1} \sim E}[r(s_t,a_t) + \gamma E_{a_{t+1} \sim \pi}[Q^{\pi}(s_{t+1},a_{t+1})]] \tag{1}$$

If the target policy is deterministic we can describe it as a function $\mu : S \rightarrow A$ and avoid the inner expectation:

$$Q^{\mu}(s_t,a_t) = E_{r_t,s_{t+1} \sim E}[r(s_t,a_t) + \gamma Q^{\pi}(s_{t+1},\mu_{s_{t+1}})] \tag{2}$$

The authors consider function approximators parameterized by θ^Q, which they optimize by minimizing the loss:

$$L(\theta^Q) = E_{s_t \sim \rho^{\beta},a_t \sim \beta,r_t \sim E}[(Q(s_t,a_t|\theta^Q) - y_t)^2] \tag{3}$$

where

$$y_t = r(s_t,a_t) + \gamma Q(s_{t+1},\mu(s_{t+1})|\theta^Q) \tag{4}$$

The DPG algorithm maintains a parameterized actor function $\mu(s|\theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action. The critic $Q(s,a)$ is learned using the Bellman equation as in Q-learning. The actor is updated by following the applying the chain rule to the expected return from the start distribution J with respect to the actor parameters:

$$\nabla_{\theta^\mu} J = E_{s_t \sim \rho^\beta}[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)} \nabla_{\theta_\mu}\mu(s|\theta^\mu)|_{s=s_t}] \tag{5}$$

When updating the target network, the authors use soft target updates: the weights of these target networks are updated by having them slowly track the learned networks: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau << 1$. This can greatly improving the stability of learning.
The authors also use batch normalization to learn effectively across many different tasks with differing types of units, without needing to manually ensure the units were within a set range.
The authors construct an exploration policy μ' by adding noise sampled from a noise process N to the actor policy: $\mu'(s_t) = \mu(s_t|\theta_t^\mu) + N$. N can be chosen to suit the environment.

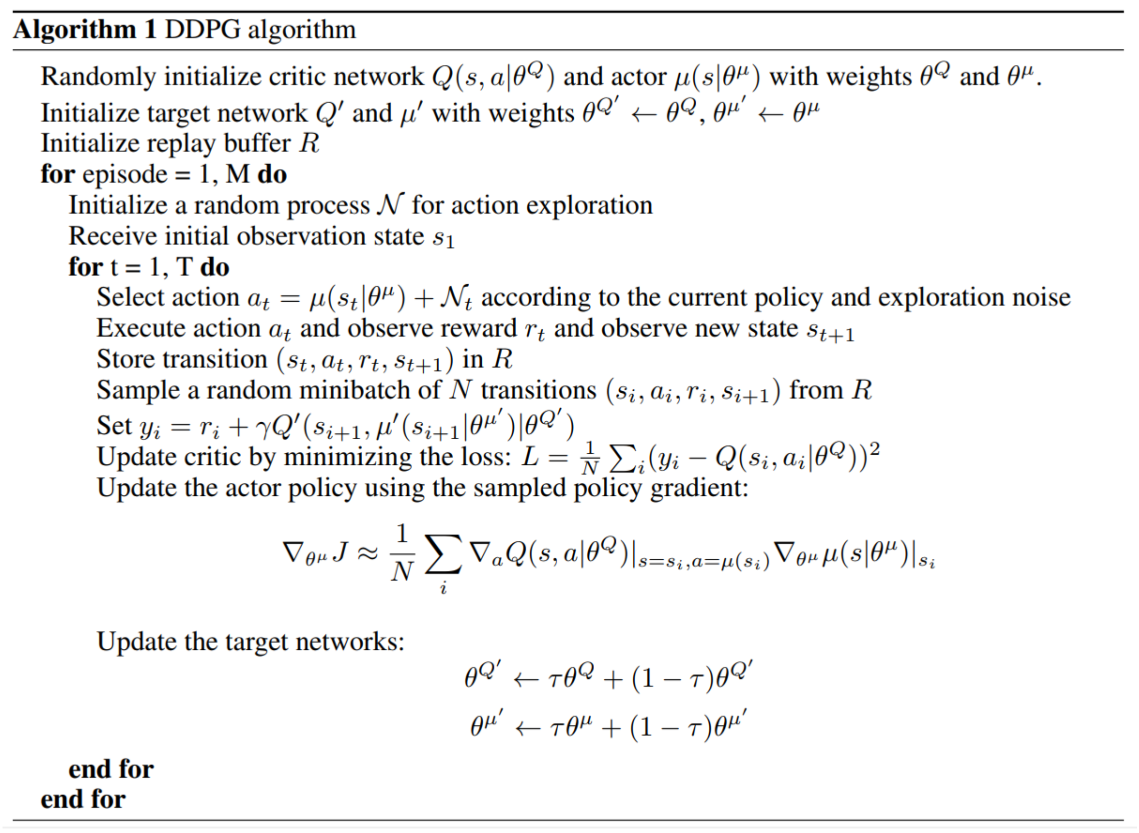
2 Scope of reproducibility

The authors claim that using the same learning algorithm, network architecture and hyper-parameters, their algorithm robustly solves more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving. Their algorithm is able to find policies whose performance is competitive with those found by a planning algorithm with full access to the dynamics of the domain and its derivatives. They further demonstrate that for many of the tasks the algorithm can learn policies "end-to-end": directly from raw pixel inputs.
I test the following concrete claims:

1. DDPG is able to learn good policies on Pendulum game using a low-dimensional state description.

2. DDPG can't learn well without target network.

3 Methodology



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
```

3.1 Model descriptions

My model consists of Actor and Critic. The input of Actor is state and the output of Actor is predicted action. The input of Critic is state and action and the output of Critic is $Q(state,action)$.

Both Actor and Critic are composed of two networks, eval network and target nework. When updating the target network, they both adopt ”soft”target updates, rather than directly copying the weights of eval network.

Actor network has two layers. There are 30 neural units in the first layer and *action_dim* neural units in the second layer, *action_dim* is the dimension of action. The activation function of the first layer is relu, and the activation function of the second layer is tanh. Both layers of networks have bias terms. The initialization mean value of parameter W is 0, the initialization standard deviation is 0.3, the initialization of parameter b is 0.1. The initialization of the parameters of Critic is the same. Since action is predicted and the output range of Actor network acti-

vated by tanh is $(-1,1)$, we need to multiply the output of the Actor network by *action_bound*, so that is the correct action range.
Critic network has two inputs, so the two inputs need to be processed separately. The input of the first part is state, and the input of the second part is action. The two parts are all connected by 30 neural units, and then add the two parts together, add a bias term, and then use the relu activation function, which constitutes the first layer of the neural network. The second layer has only one neural unit with bias term and no activation function.
When updating the Actor network, according to the formula(5), updating the gradient requires two parts, one is $\frac{\partial Q}{\partial a}$, which is given to the Actor network after being calculated by the critical network, and the other is $\frac{\partial \mu}{\partial \theta}$.
When updating the Critic network, calculate y_t according to the formula(4), where $Q(s_{t+1}, \mu(s_{t+1})|\theta^Q)$ is the result of target network. Then minimize $L(\theta^Q)$.
As in DQN, I used a replay buffer to make the samples independently and identically distributed. The replay buffer is a finite sized cache R. Transitions were sampled from the environment according to the exploration policy and the tuple (s_t, a_t, r_t, s_{t+1}) was stored in the replay buffer. When the replay buffer was full the oldest samples were discarded. At each timestep the Actor and Critic are updated by sampling a minibatch uniformly from the buffer. Because DDPG is an off-policy algorithm, the replay buffer can be large, allowing the algorithm to benefit from learning across a set of uncorrelated transitions.
In my implement, I called it Memory.

3.2 Datasets

The environment I used is Pendulum-v0 in gym.

3.3 Hyperparameters

I don't use the same hyperparameters in the paper.
exploration noise: $var = 3$, the noise decreases with the progress of training
test times: $replicas = 5$
game times: $episodes = 400$
step times in one game: $steps = 200$
the learning rate of Actor: $lr\_a = 0.001$
the learning rate of Critic: $lr\_c = 0.001$

discount factor: $gamma = 0.9$
the capacity of Memory: $memory\_capacity = 10000$
Amount of data fed to the network at one time: $batch\_size = 32$

3.4 Experimental setup and code

Training

At the beginning of the game, there is no data in Memory, so agent just plays the game randomly until Memory is full. On each step, I uses *choose_action* to obtain the expected action. Because the model haven't learned effective game strategies at the beginning, so it need more exploration. When the I fetches data from Memory to update the network, var reduces. On each step, I put the record $(s, a, r, s\_)$ into Memory.

Performance after training

After training, I play 5 rounds of games, play 200 steps each time, and record the average and maximum values of 5 rounds of games.

Model without target network

I also report the result with the target network removed. We only need to make some adjustments to the model, as follows.
The target network of Actor and Critic is removed, and the replacement policy is also not required any more. For Critic, *target_q* is calculated using *eval_q*. When learning, there is no need to update the target network.

3.5 Computational requirements

CPU: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx

4 Results

4.1 Results reproducing original paper

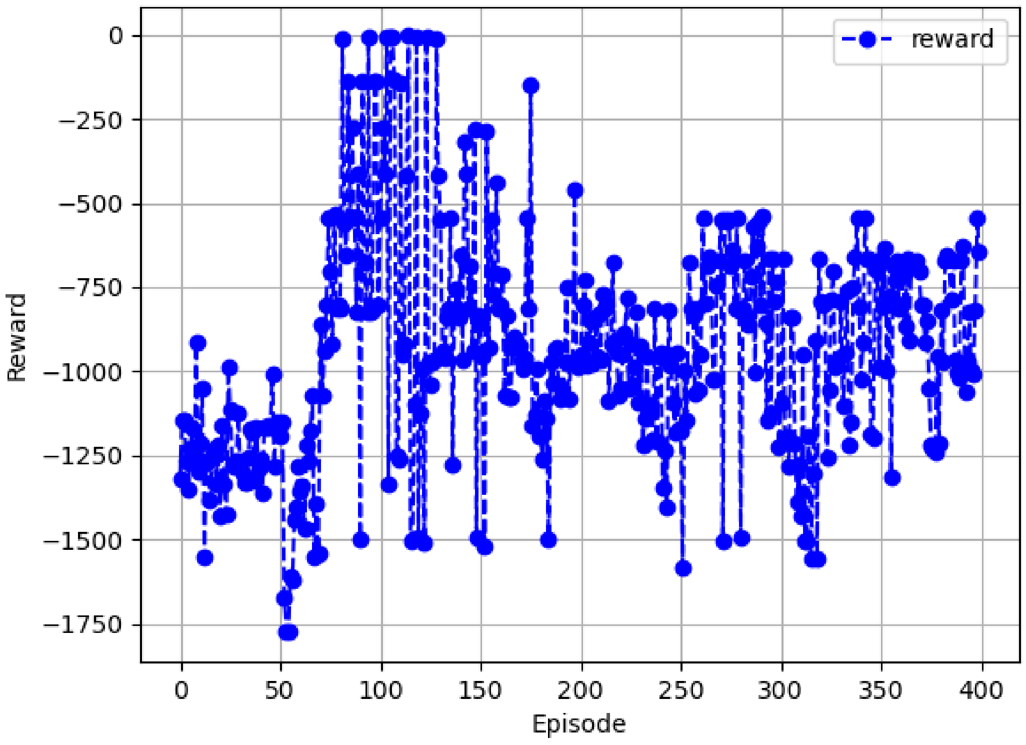
Performance after training

	avg_reward	best_reward
Pendulum-v0	-656	-537

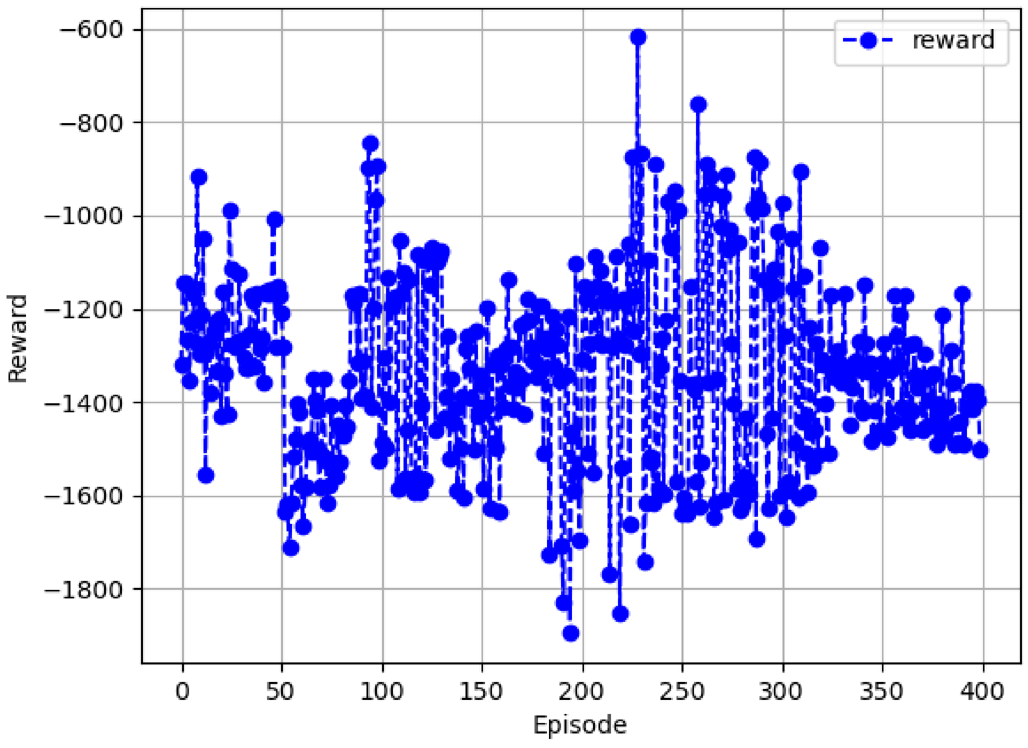
Tabel 1: Performance after training

Model without target network

It can be seen that the model without target network cannot converge. In order to perform well across all tasks, target network is necessary.



Figur 1: Target network



Figur 2: Without target network

5 Discussion

I test the following concrete claims successfully:

1. DDPG is able to learn good policies on Pendulum game using a low-dimensional state description.

2. DDPG can't learn well without target network.

5.1 What was easy

The author provides clear pseudo code. Therefore, I can easily implement the algorithm.

5.2 What was difficult

Neural network parameters are incomplete and unclear, so I can only adjust it by myself.

6 References

github
CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING