REINFORCEMENT LEARNING: DEEP DETERMINISTIC POLICY GRADIENT [1]

AKSHAY DHONTHI (1887502)[2]

CONTENTS

LIST OF FIGURES

[1] *Find the implementation codes and results in the repository* https://github.com/akshaydr/DDPG_reinforcement_learning

[2] *Department: Artificial Intelligence and Robotics Engineering, Sapienza University, Rome, Italy*

1 INTRODUCTION

1.1 Definition

Deep Deterministic Policy Gradient is an algorithm for deep reinforcement learning with continuous space. It is used for environments with continuous action space. It is an extension of Deep-Q-Network (DQN) where DQN is from continuous state space and discrete action space. The algorithm concurrently learns a Q-function and a policy. It uses off-policy data and Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

In a continuous action environment finding the maximum over actions very difficult. In other words, it is difficult to solve the optimization problem. However, policy gradient methods and actor-critic methods mitigate the problem by looking for a local optimum by Bellman methods.

1.2 History

DDPG is derived from two algorithms Deep-Q-Network (DQN) and Deterministic Policy Gradients (DPG). DPG is an efficient gradient computation for deterministic policies. DQN has few features such as reply buffer and target-q-network which are used to define DDPG algorithm. Combining these two algorithms will give the DDPG algorithm.

1.3 Architecture

The DDPG has two networks which are actor and critic as shown in the figure. The critic takes sets of inputs which are states and actions and returns Q-value of that state and of that action. Whereas, the actor takes set of states as input and returns a set of actions. All the updates are performed based on stochastic gradient decent.

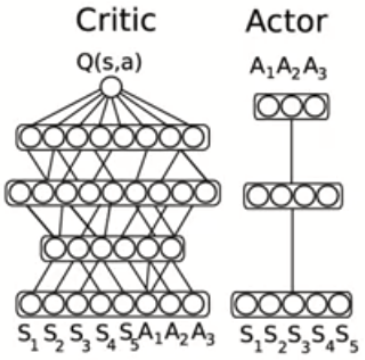

Figure 1: DDPG architecture

.

2 LEARNING METHOD

2.1 Training the Critic

Training the critic is similar to the DQN algorithm. The equations change a little bit since the input is both states and actions while in DQN, the input is just states. To learn the critic, we take set of samples from the buffer called mini-batch. Where buffer is a data structure which stores the history of states and performed actions. Initially when the buffer is empty, we keep adding the state and actions with random parameter initialization until it is possible to get a mini-batch. Now, for each sample in mini-batch, Q-Network should minimize the temporal difference error. .

$$\delta_t = r_t + \gamma Q(s_{t+1}, \pi(s_{t+1})|\theta) - Q(s_t, a_t|\theta)$$

Give a mini-batch of N samples [state, action, reward, next state] and a target network Q', compute,

$$y_i = r_i + \gamma Q'(s_{i+1}, \pi(s_{i+1})|\theta')$$

And then minimize the mean squared loss between updated Q value and the original Q value using the below equation,

$$Loss = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

But the above equation does not work because in supervised learning, the desired value is constant whereas in RL the desired value is a function of Q. So the trick is to not change the Q function for certain number of steps and learn based on that Q-function and later update the Q-function every set number of steps. This is done by defining a target network. We also use reply buffer to shuffle the samples. To do this, we assume each sample is independently and identically distributed.

2.2 Training the Actor

The actor is trained based on deterministic policy gradient. For the policy function, our objective is to maximize the expected return,

$$J(\theta) = \mathbb{E}[Q(s, a)|_{s=s_t, a_t=\mu(s_t)}]$$

To calculate the policy loss, we take the derivative of the above function with respect to the policy parameter. But since, we are updating the policy in an off-policy way with the batch of experience, we take the mean of the sum of gradients calculated from mini-batch,

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N}\sum_i [\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i}]$$

2.3 Summary

Overall, in actor, we give state as input and compute the action. We put the action along with the states in the critic and compute the Q-value of that state and of the action. And then we back-propagate the gradient of Q-value with respect to the actions. This gives the Temporal Difference (TD) error. Then we train the actor with the TD error as loss function.

The actor will improve provided critic is good enough. In the beginning we train the critic faster than the actor and critic gets accurate. And then the actor will get more and more optimal as critic is improving. Similar to DQN, we use the target network which is tuned based on the value 'tau'. It takes the Q-function for certain number of steps and then update the Q-function. We add Ornstein-Uhlenbenk noise for performing exploration.

3 DDPG ALGORITHM

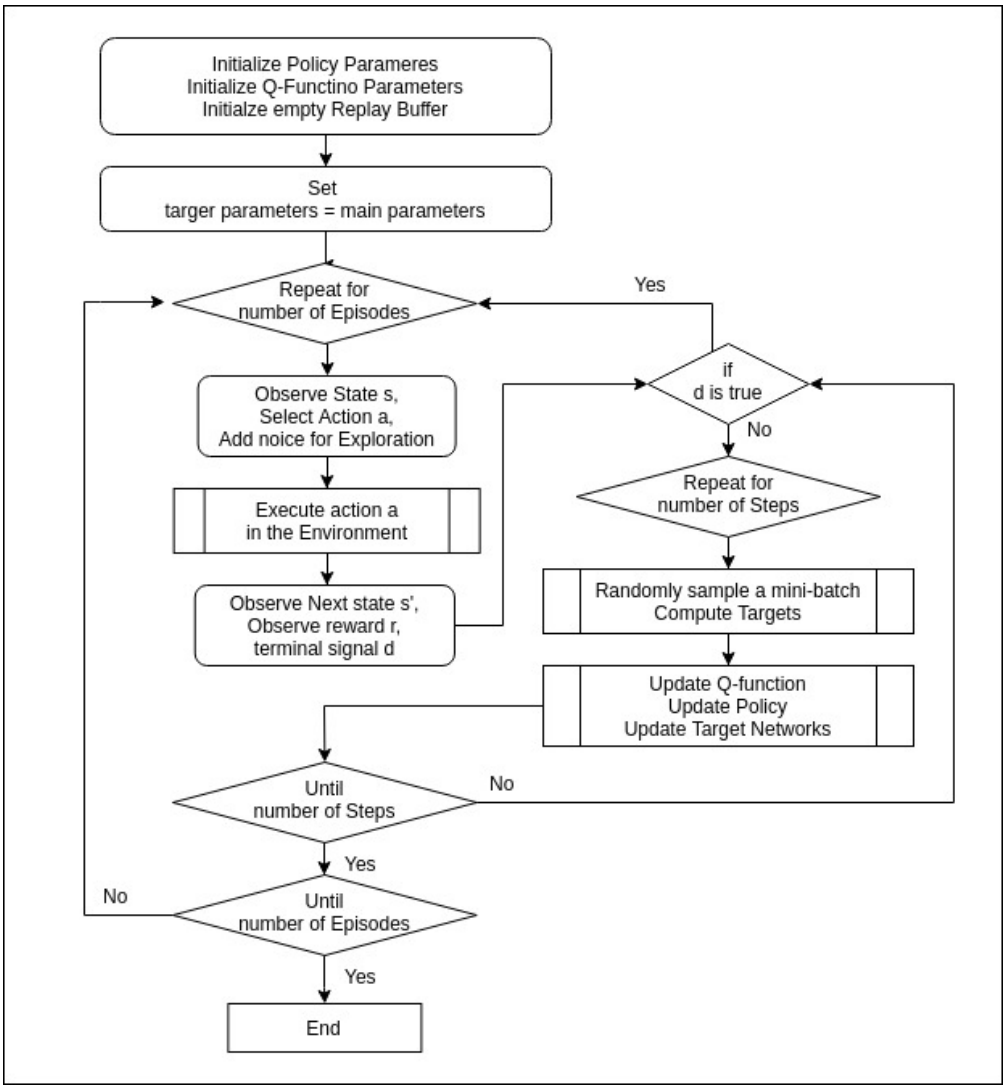

Figure 2: DDPG Algorithm

.

4 RESULTS

4.1 Environment: Mountain Car Continuous

We can see the DDPG agent learns optimal policy for Mountain Car Continuous task as shown below.

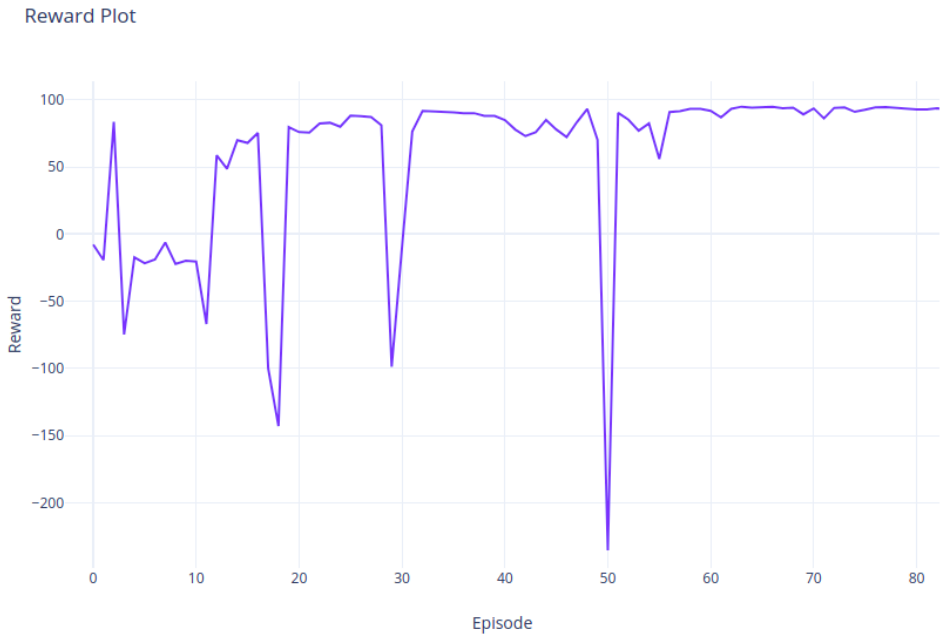

Figure 3: Reward graph: Mountain Car Continuous

.

Applied Hyper Parameters:

- ACTOR LEARNING RATE: 0.0001
- CRITIC LEARNING RATE: 0.001
- CRITIC UPDATE DISCOUNT FACTOR: 0.99
- SOFT TARGET UPDATE PARAMETER: 0.01
- MINI-BATCH SIZE: 64

4.2 Environment: Ant (Mujoco)

We can see the DDPG agent learns optimal policy for Ant environment as shown below. The reward keeps improving.

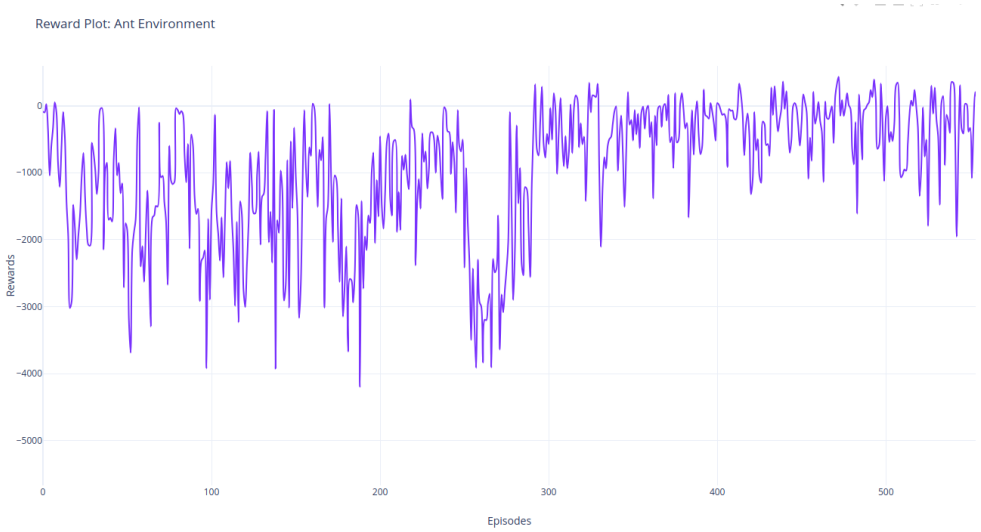

Figure 4: Reward graph: Mountain Car Continuous

.

Applied Hyper Parameters:

- ACTOR LEARNING RATE: 0.0001
- CRITIC LEARNING RATE: 0.001
- CRITIC UPDATE DISCOUNT FACTOR: 0.99
- SOFT TARGET UPDATE PARAMETER: 0.01
- MINI-BATCH SIZE: 64

5 CONCLUSION

DDPG algorithm is mainly for continuous action space. Using neural networks yields exceptional results in solving challenging RL issues across a verity of domains. DDPG was successfully able to learn continuous action space environment. Further by modifying hyper parameters might increase the learning speed and also get accurate.