Udacity Deep Reinforcement Learning – Project 2: Continuous Control

1. Summary

For this project, we trained a double-jointed arm agent to follow a target location.

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

The task is episodic, with 1000 timesteps per episode. In order to solve the environment, the agent must get an average score of +30 over 100 consecutive episodes.

2. Methods

2.1 Policy-based and value-based network.

The network which learns to give a definite output by giving a particular input to the game is known as Policy network.

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: transition probability i.e. distribution over next state given (state, action) pair
γ : discount factor

Usual Notations for RL environments

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$$

Optimal Policy

The value network assigns value/score to the state of the game by calculating an expected cumulative score for the current state(s). every state goes through the value network. The states which gets more reward obviously get more value in the network.

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

Value Function

The key objective is always to maximize the reward. Actions that results in a good state obviously get greater reward than others.

2.2 Actor-critic methods.

Actor-critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error. This scalar signal is the sole output of the critic and drives all learning in both actor and critic, as suggested by the figure 1.

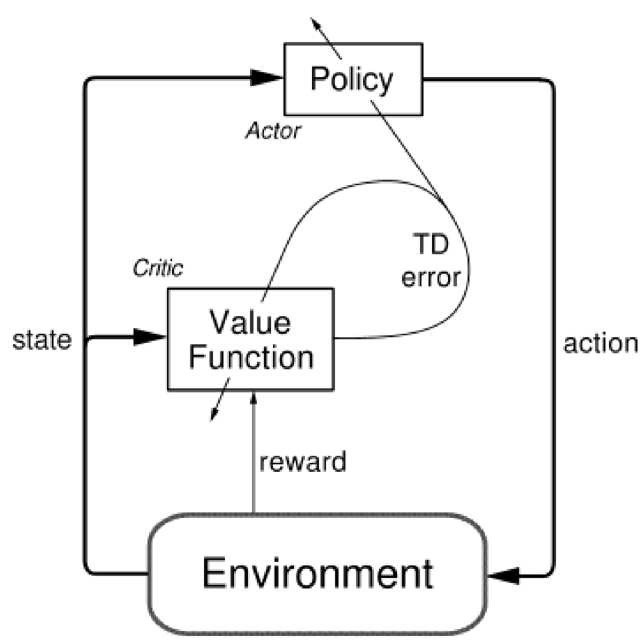


Figure 1. The actor-critic architecture

3. Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function Q*(s,a), then in any given state, the optimal action a*(s) can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a).$$

In DDPG, we use 2 deep neural networks: one is the actor and the other is the critic.

4. Replay Buffer

In the implementation of DDPG, I also used replay buffer which could save the states for training purpose.

5. Target updates

Q-learning algorithms make use of target networks. The term

$$r + \gamma(1-d)\max_{a'} Q_{\phi}(s',a')$$

is called the target, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: \phi. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to \phi, but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted \phi_{\text{targ}}.

In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi,$$

where rho is a hyperparameter between 0 and 1 (usually close to 1

6. Pseudocode

Here is the sudo code of Deep Deterministic Policy Gradient (DDPG)

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\begin{aligned} \phi_{\text{targ}} &\leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi \\ \theta_{\text{targ}} &\leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta \end{aligned}$$

    end for
  end if
until convergence

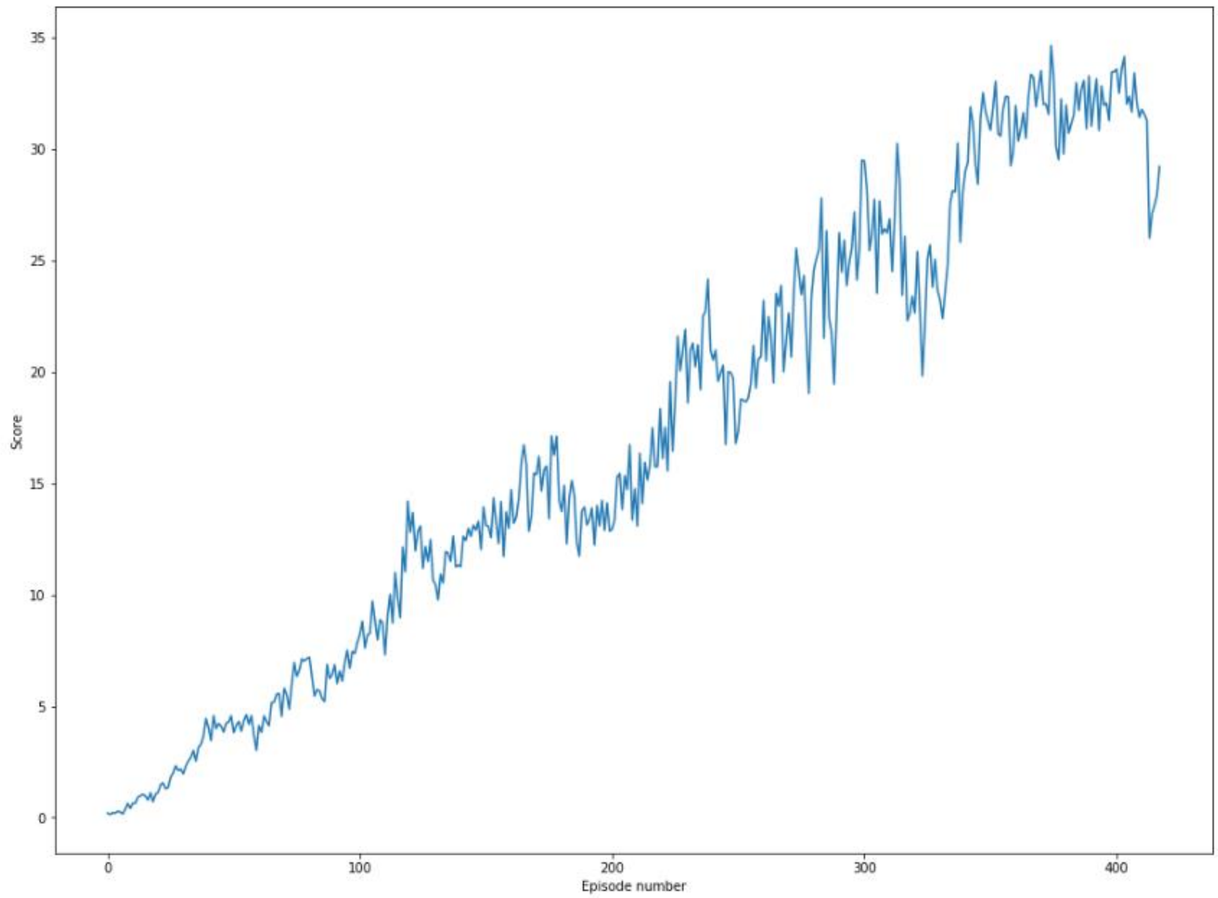
7. Parameter in training

During training, I used hidden layers with 400 and 300 units respectively in model.py. And applied 512 as batch size in ddpg_agent.py. If I use the batch size as large as 1024, the CPU utilization would be too high and the server would crash. I also tried smaller neural network with 200 and 150 units in network but that would lead to very slow increment of average score. Finally the average score reached 30 at 418 episode.

For the other parameters, I used learning rate for Actor as 5e-4 and learning rate for critic for 1e-3. The implementation is similar to the sample code given in ddpg-pendulum from udacity/deep-reinforcement-learning github repository.

8. Results

The agent is trained until it reached at least +30 for at last 100 episodes. My agent reached this target at episode of 418. Here is the score plot of each episodes.




```
CPU times: user 688 ms, sys: 0 ns, total: 688 ms
Wall time: 687 ms
```

The total training time is about 9 hours 16 minutes.

7 Future work

- Try to solve first environment to train only one agent get an average score of +30 over 100 consecutive episodes.