Udacity Deep Reinforcement Learning – Project 3: Collaboration and Competition

Rui La

1. Summary

In this project, we will train a agent to play tennis.

In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

- After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- This yields a single score for each episode.

The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5.

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.

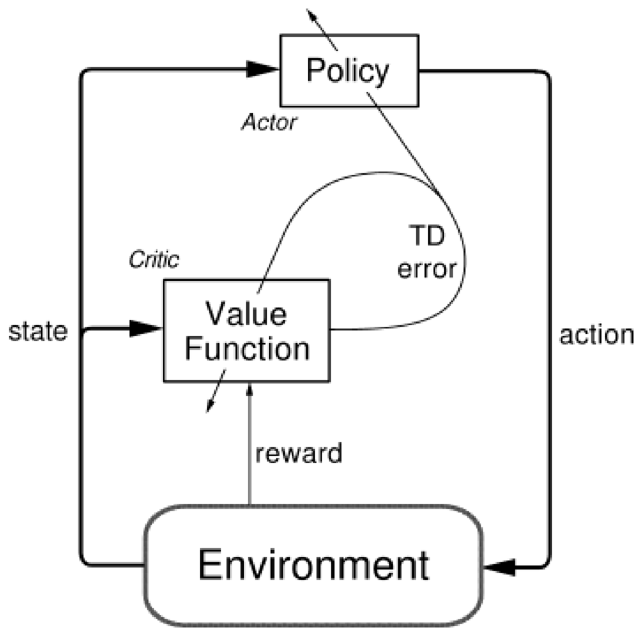


Figure 1. The actor-critic architecture

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}}.

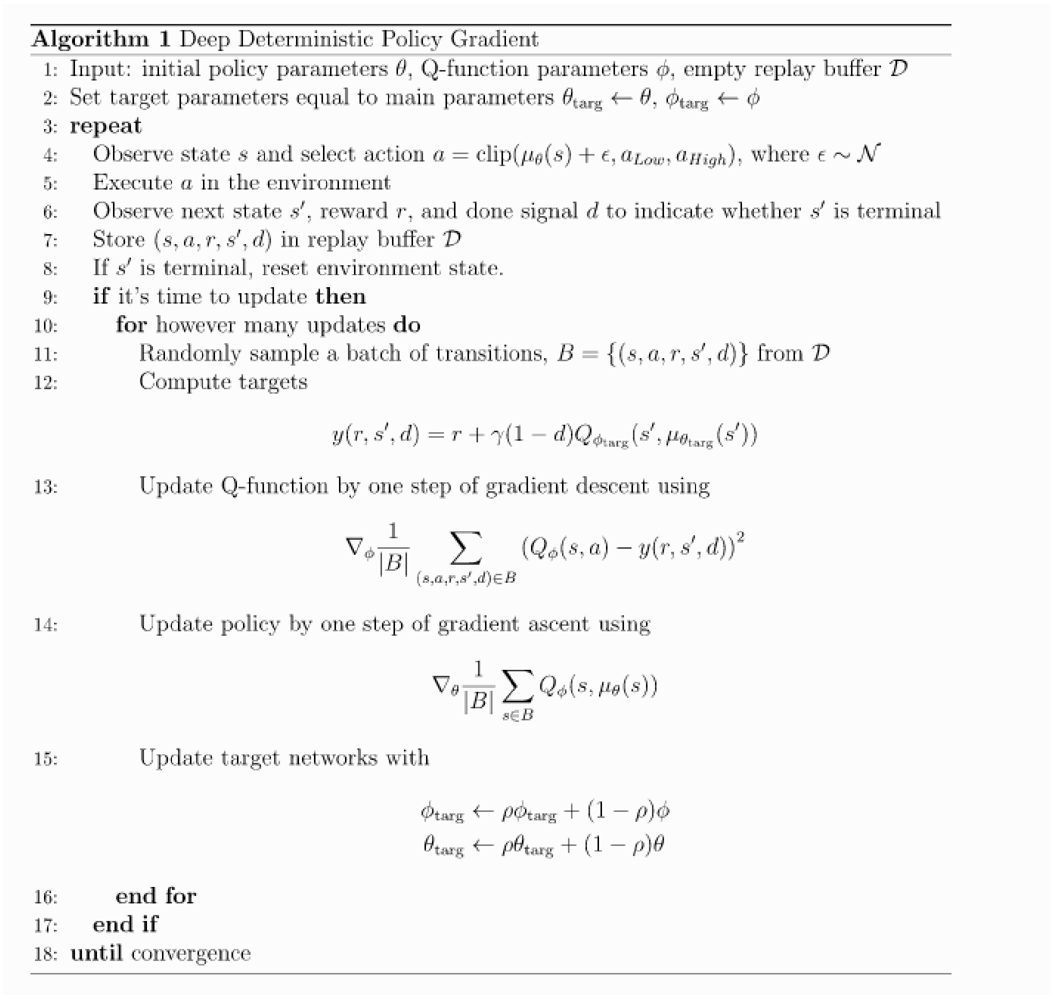
In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

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

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
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:     Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:     Execute $a$ in the environment
6:     Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:     Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:     If $s'$ is terminal, reset environment state.
9:     if it's time to update then
10:        for however many updates do
11:            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:            Compute targets
                   $y(r, s', d) = r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:            Update Q-function by one step of gradient descent using
                   $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s,a) - y(r,s',d)\right)^2$
14:            Update policy by one step of gradient ascent using
                   $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:            Update target networks with
                   $\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1-\rho)\phi$
                   $\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$
16:        end for
17:    end if
18: until convergence
```

7. Parameter in training

During training, I used hidden layers with 300 and 150 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. Finally the average score reached 30 at **728 episode.**

For the other parameters, I used learning rate for Actor as 1e-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. I used relu activation function for each layer.

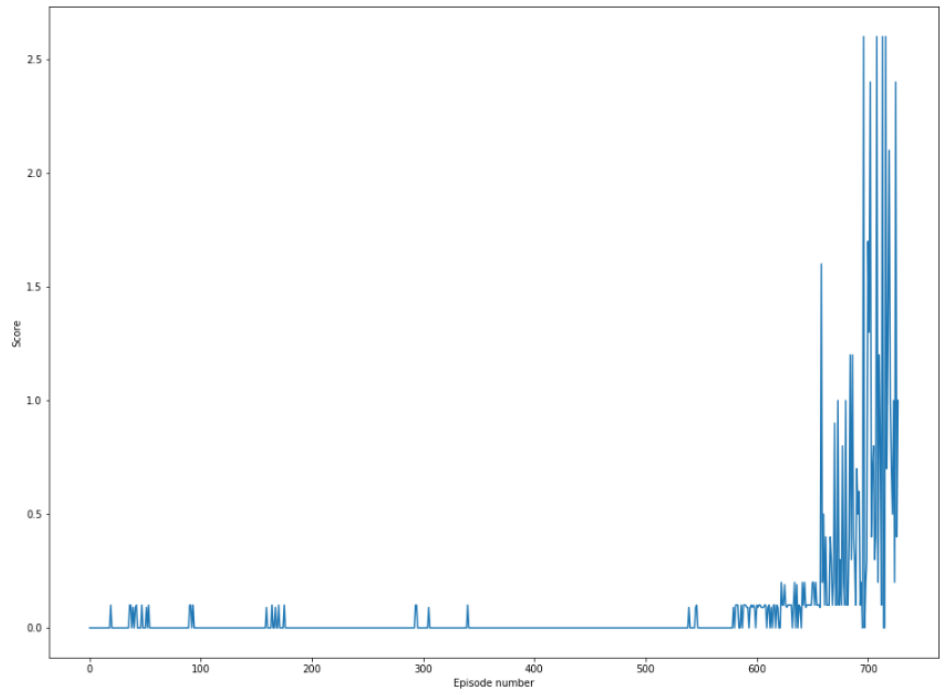
Neural Network Architecture

Actor

State—BatchNorm—300—relu—150—relu—BatchNorm—action—tanh

Critic

State—BatchNorm—300—relu—150—relu—action



```
CPU times: user 252 ms, sys: 104 ms, total: 356 ms
Wall time: 208 ms
```

Future Work

1. From the scores over episodes plot we can see the episode score remained 0 for a long time and the average score is close to 0 until 600 episodes. This is not like a normal training process. Also I tried to continue the training process and see the average score went back to very low number after 900 episodes. So in the future work I want to modify the ddpg agent and model parameters to see if I can implement a steady increasing average score model which is more reasonable.
2. Modifying the hyperparameters of the model to achieve less episodes and faster training time.
3. Since I used AWS ec2 instance to train the model, I didn't visualize how my agent work in real environment. I will copy the weights to install it on my computer and see if there is anything I can improve from the performance of real agent.
4. I will try to complete soccer training after I feel comfortable with the current model in tennis game.
5. Compare low level model without experience replay and actor-critic with my current model to see if how these complicated algorithm affect the performance and the training speed.