

Project 3 (Collaboration and Competition) for Udacity Deep Reinforcement Learning Nanodegree

Goal

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 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,

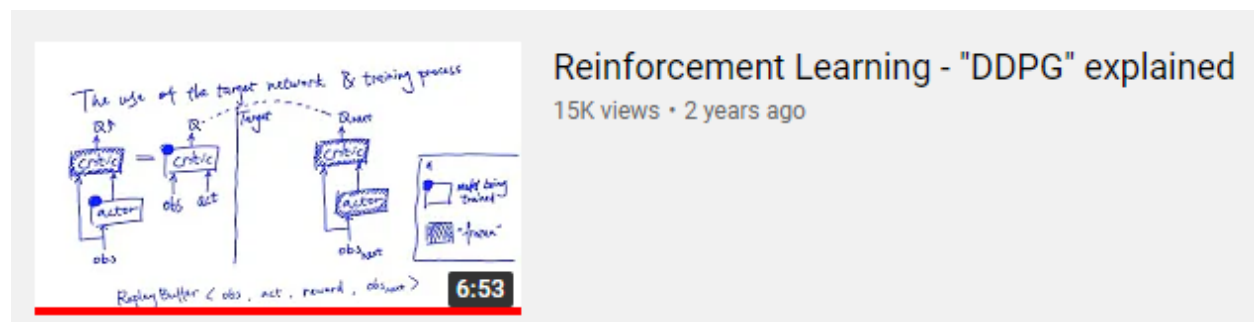
After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.

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.

DDPG in detail

This Youtube video explained DDPG in a very clean way, and it is highly recommend to watch through the video and get some basic understanding of DDPG:

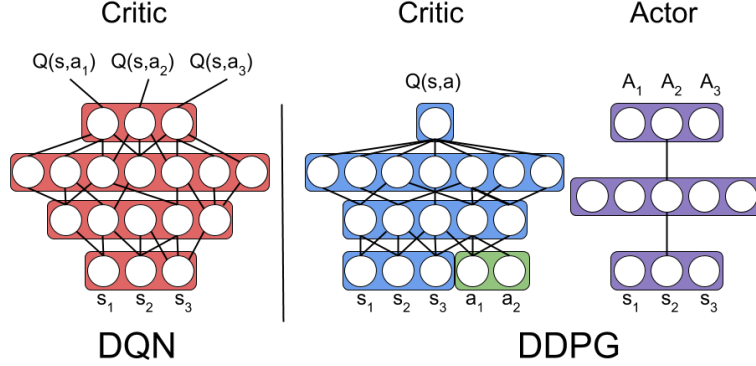


Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. A high-level DDPG structure looks the following, and you can see it has some DQN features like the replay buffer, critic network and so on. As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a target policy network to compute an action which approximately maximizes Q_{target} . The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_{\phi}(s,a) - (r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))) \right)^2 \right],$$

The below image shows the compasiron between DDPG and DQN.



MADDPG

MADDPG, or Multi-agent DDPG, extends DDPG into a multi-agent policy gradient algorithm where decentralized agents learn a centralized critic based on the observations and actions of all agents.

It leads to learned policies that only use local information (i.e. their own observations) at execution time, does not assume a differentiable model of the environment dynamics or any particular structure on the communication method between agents, and is applicable not only to cooperative interaction but to competitive or mixed interaction involving both physical and communicative behavior. The critic is augmented with extra information about the policies of other agents, while the actor only has access to local information. After training is completed, only the local actors are used at execution phase, acting in a decentralized manner.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

```

for episode = 1 to  $M$  do
  Initialize a random process  $\mathcal{N}$  for action exploration
  Receive initial state  $\mathbf{x}$ 
  for  $t = 1$  to max-episode-length do
    for each agent  $i$ , select action  $a_i = \mu_{\theta_i}(o_i) + \mathcal{N}_t$  w.r.t. the current policy and exploration
    Execute actions  $a = (a_1, \dots, a_N)$  and observe reward  $r$  and new state  $\mathbf{x}'$ 
    Store  $(\mathbf{x}, a, r, \mathbf{x}')$  in replay buffer  $\mathcal{D}$ 
     $\mathbf{x} \leftarrow \mathbf{x}'$ 
    for agent  $i = 1$  to  $N$  do
      Sample a random minibatch of  $S$  samples  $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$  from  $\mathcal{D}$ 
      Set  $y^j = r_i^j + \gamma Q_i^{\mu}(\mathbf{x}'^j, a_1^j, \dots, a_N^j)|_{a_i = \mu_i(o_i^j)}$ 
      Update critic by minimizing the loss  $\mathcal{L}(\theta_i) = \frac{1}{S} \sum_j (y^j - Q_i^{\mu}(\mathbf{x}^j, a_1^j, \dots, a_N^j))^2$ 
      Update actor using the sampled policy gradient:
      
$$\nabla_{\theta_i} J \approx \frac{1}{S} \sum_j \nabla_{\theta_i} \mu_i(o_i^j) \nabla_{a_i} Q_i^{\mu}(\mathbf{x}^j, a_1^j, \dots, a_i, \dots, a_N^j)|_{a_i = \mu_i(o_i^j)}$$

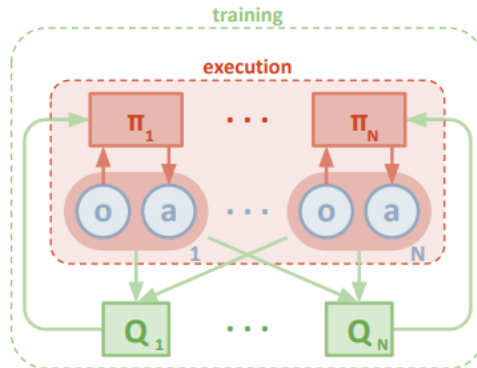
    end for
    Update target network parameters for each agent  $i$ :
    
$$\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$$

  end for
end for

```

Approach

The high level structure shows as the following, and the code under maddpg_agent.py follows the diagram:



1. The state and action space of this environment

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.

```
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents
num_agents = len(env_info.agents)
print('Number of agents:', num_agents)

# size of each action
action_size = brain.vector_action_space_size
print('Size of each action:', action_size)

# examine the state space
states = env_info.vector_observations
state_size = states.shape[1]
print('There are {} agents. Each observes a state with length: {}'.format(states.shape[0], state_size))
print('The state for the first agent looks like:', states[0])
```

2. Explore the environment by taking random actions

We then take some random actions based on the environment we created just now, and see how the agents perform (apparently it will be bad without learning)

```
for i in range(1, 6):
    env_info = env.reset(train_mode=False)[brain_name]
    states = env_info.vector_observations
    scores = np.zeros(num_agents)
    while True:
        actions = np.random.randn(num_agents, action_size)
        actions = np.clip(actions, -1, 1)
        env_info = env.step(actions)[brain_name]
        next_states = env_info.vector_observations

        rewards = env_info.rewards
        dones = env_info.local_done
        scores += env_info.rewards
        states = next_states
        if np.any(dones):
            break
    print('Score (max over agents) from episode {}: {}'.format(i, np.max(scores)))

# play game for 5 episodes
# reset the environment
# get the current state (for each agent)
# initialize the score (for each agent)
# select an action (for each agent)
# all actions between -1 and 1
# send all actions to the environment
# get next state (for each agent)
# get reward (for each agent)
# see if episode finished
# update the score (for each agent)
# roll over states to next time step
# exit loop if episode finished
```

3. Implement the MADDPG algo to train the agent

The last step is to implement the MADDPG algorithm to train the agents. The code can be found in `maddpg_agent.py`. However, I would like to mention several techniques to improve the speed and convergence:

Adjust the OU noise by adding decreasing factors, and related discussions can be found in this repo: [Udacity discuss channel](#)

Change different discount factor GAMMA to see the performance. The agent does not need to see too far to predict its next movement. So slightly reduce the GAMMA value to focus more on the current states.

Results:

The average rewards along with the training process show as following:

