Project 3. Collaboration and Competition

1 DDPG Algorithm

The DDPG (Deep Deterministic Policy Gradient) algorithm [1] is used in this project. DDPG is a deterministic off-policy gradient method restricted to continuous action-spaces. DDPG is similar to the DQN algorithm [2]. It uses many of the same techniques employed in DQN, such as an experience replay buffer to consider past experiences, and separate target networks to stabilize the learning process. The main difference between DDPG and DQN is that DQN obtains the next (discrete) action by getting the maximum of the outputs of the Q network (thus obtaining the greedy action), while in DDPG the next (continuous) action is directly obtained by the actor Q network. DDPG uses the actor-critic approach, where the agent learns both a policy (using the actor network) and a value function (using the critic network). Also, in the learning process, some noise is added to the next obtained action to favour the exploration of the environment. The main steps of the DDPG algorithm are shown in Algorithm 1.

In DQN the weights of the main network are directly copied into the target network every certain number of time steps. However, in DDPG we perform a soft update of the target network weights (for both the actor and the critic networks) at every time step. This soft update is controlled by a τ parameter. Thus the target network weights slowly and continuously follow the main network weights. Note that this soft update can also be used in DQN.

2 Python Code

The code is divided in three files: `model.py`, `ddpg_agent.py` and `p3.py`. We have reused the files employed for the second project, which in turn are an extension of the DDPG code provided in the directory `ddpg-pendulum` of the Udacity repository.

The file `model.py` contains the *Actor* and *Critic* classes. They define the neural networks for the actor (μ and $\hat{\mu}$) and the critic (Q and $\hat{Q}$). Their architectures are the same than those of used in the second project:

- The actor networks have two hidden layers of 256 units and a ReLU activation function in each one. The input of the actor is a state and the output consists of two units corresponding to the two elements of an action vector, with a tanh activation function.
- The critic networks also have two hidden layers of 256 units and a ReLU activation function in each one. The input of the critic is a state and an

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a \mid \theta^Q)$ and actor $\mu(s \mid \theta^\mu)$ with weights θ^Q and θ^μ
Initialize target networks $\hat{Q}$ and $\hat{\mu}$ with weights $\theta^{\hat{Q}} \leftarrow \theta^Q$ and $\theta^{\hat{\mu}} \leftarrow \theta^\mu$
Initialize replay memory $\mathcal{D}$
for episode $= 1 \dots M$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Observe initial state s_1
  for $t = 1 \dots T$ **do**
    Select action $a_t = \mu(s_t \mid \theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store tuple (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of S tuples (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set target $y_j = r_j + \gamma \hat{Q}(s_{j+1}, \hat{\mu}(s_{j+1} \mid \theta^{\hat{\mu}}) \mid \theta^{\hat{Q}})$
    Update critic by minimizing the loss: $L = \frac{1}{S} \sum_j \left(y_j - Q\left(s_j, a_j \mid \theta^Q\right)\right)^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{S} \sum_j \nabla_a Q(s, a \mid \theta^Q)|_{s=s_j, a=\mu(s_j)} \nabla_{\theta^\mu} \mu(s \mid \theta^\mu)|_{s_j}$$

    Update the target networks:

$$\theta^{\hat{Q}} \leftarrow \tau\theta^Q + (1-\tau)\theta^{\hat{Q}}$$
$$\theta^{\hat{\mu}} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\hat{\mu}}$$

  end for
end for

action vector, with the particularity that the state is connected with the first hidden layer, while the action vector is connected directly with the second hidden layer. The output of the critic networks is a single unit.

The file `ddpg_agent.py` contains the hyperparameters, the *Agent* class, the *OUNoise* class and the *ReplayBuffer* class. In the *Agent* and *ReplayBuffer* classes, a variable *self.device* has been added to specify the architecture in which the calculations are to be carried out (CPU or GPU). The *OUNoise* class implements the Ornstein-Uhlenbeck noise [3], which is applied in the training process to the next obtained action to favour the exploration of the environment. The hyperparameters of the Ornstein-Uhlenbeck process are the default ones provided in the code: $\mu = 0$, $\theta = 0.15$ and $\sigma = 0.2$. Obviously, this noise is not applied in the testing process.

The hyperparameters for the DDPG algorithm and their values are defined at the top of `ddpg_agent.py`:

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 5e-2              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor
```

```
LR_CRITIC = 2e-4        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
UPDATE_STEPS = 1        # how often to run the learning algorithm
NOISE_DECAY = 1e-3      # noise reduction factor after the learning step
```

We have preserved the clipping of the gradient norms performed for the second project (in line 111 of `ddpg_agent.py`) in order to prevent the gradients from exploding, and therefore to prevent big changes in the network weights when updating them. We have also maintained the values for the replay buffer size (one million of tuples) and the size of a minibatch (256). The learning step is executed at every time step, and the Adam algorithm is used in the optimization process for the actor and the critic.

For this project, however, it has been necessary to add a noise decaying factor that decrements the noise after each learning step. Specifically, an epsilon parameter is defined and initialized to 1 in the *Agent* class (in line 52), and it is reduced by the `NOISE_DECAY` factor until it reaches a minimum of 0.01 (line 128). When applying noise to the next obtained action, the noise is multiplied by this epsilon parameter (line 79).

Note that, despite having two interactive agents, we have been able to solve the problem by using the DDPG algorithm with only one agent that shares the experiences of the two agents. The reason may be that both agents are identical and the goal is just to keep the ball in play as long as possible (not defeating the adversary), thus making unnecessary the use of two agents in the training process.

Finally, the file `p3.py` is the main file which contains the *main* function. It also has two more functions: *train_ddpg*, which is called in the learning process, and *test*, called in the testing process.

3 Results

The code has been executed using Ubuntu 18.04 and Python 3.6.9. The environment is solved (i.e. an average score of +0.5 is reached over 100 consecutive episodes) in 214 episodes, and the text output is the following:

```
Number of agents: 2
Episode 100  Score: 0.000  Average Score: 0.008
Episode 200  Score: 0.100  Average Score: 0.014
Episode 300  Score: 0.200  Average Score: 0.329
Episode 314  Score: 2.600  Average Score: 0.501
Environment solved in 214 episodes!  Average Score: 0.501
```

Figure 1 shows the score reached at each episode, where the red line represents the moving average of 100 episodes.

By testing the obtained agent with the first 100 episodes of the environment we get an average score of 1.906.

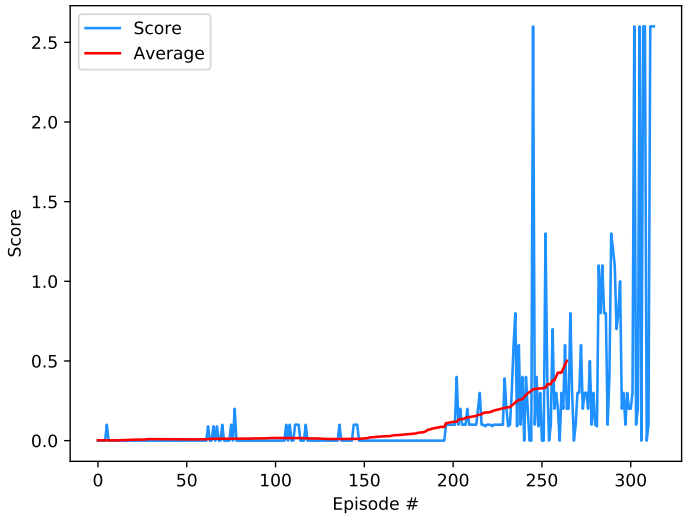

Figure 1: Score and moving average at each episode

4 Future Work

In order to get a better agent, we can try several things:

- Perform a longer training and reach a higher average score.
- Use prioritized experience replay.
- Test other algorithms such as TNPG or TRPO. According to [4], these algorithms achieve very good performance.
- Use an algorithm specifically designed for problems with multiple interactive agents, such as the MA-DDPG [5].

References

[1] T. P. Lillicrap et al. Continuous control with deep reinforcement learning. `arXiv:1509.02971v6 [cs.LG]`, 2015.

[2] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518, 529–533, 2015.

[3] G. E. Uhlenbeck, L. S. Ornstein. On the theory of the brownian motion. *Physical review*, 36, 823–841, 1930.

[4] Y. Duan et al. Benchmarking deep reinforcement learning for continuous control. `arXiv:1604.06778v3 [cs.LG]`, 2016.

[5] R. Lowe et al. Multi-agent actor-critic for mixed cooperative-competitive environments. `arXiv:1706.02275v4 [cs.LG]`, 2017.