Project 2. Continuous Control

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 `p2.py`. We have taken one of the two DDPG codes provided in the course as a basis (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 following:

- 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 four units corresponding to the four 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 = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-3         # learning rate of the actor
```

```
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
UPDATE_STEPS = 20       # how often to run the learning algorithm
NUMBER_UPDATES = 10     # runs of the learning algorithm each time
```

We have performed the two modifications of the algorithm suggested in the course (both implemented in `ddpg_agent.py`):

- In line 111 a clipping of the norms of the gradients is carried out when training the critic network, using the `torch.nn.utils.clip_grad_norm_` function. This prevents the gradients from exploding, and therefore also prevents big changes in the network weights when updating them.
- The learning step is repeated 10 times in a row every 20 time steps. This is implemented in lines 66-69.

Since the 20 agents behave equally and are independent, we only use one agent in the training process, which shares all their experiences. We have set the size of the replay buffer to one million of tuples, and the size of a minibatch to 256, since we have to deal with the experiences of 20 agents. The Adam algorithm is used in the optimization process for the actor and the critic.

Finally, the file `p2.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 +30 is reached over 100 consecutive episodes and over all agents) in 82 episodes, and the text output is the following:

```
Number of agents: 20
Episode 100  Score: 23.54  Average Score: 8.02
Episode 182  Score: 37.19  Average Score: 30.20
Environment solved in 82 episodes!  Average Score: 30.20
```

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 37.42.

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.

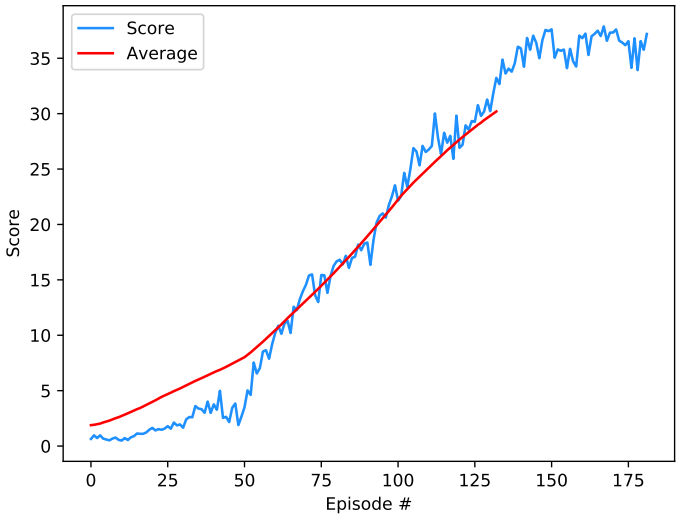

Figure 1: Score and moving average at each episode

- Use prioritized experience replay.
- Test other algorithms such as TNPG or TRPO. According to [4], these algorithms achieve very good performance.
- Use algorithms that allow multiple independent agents to be trained in parallel, such as PPO [5], A3C [6] or D4PG [7].

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] J. Schulman et al. Proximal policy optimization algorithms. `arXiv:1707.06347v2 [cs.LG]`, 2017.

[6] V. Mnih et al. Asynchronous methods for deep reinforcement learning. `arXiv:1602.01783v2 [cs.LG]`, 2016.

[7] G. Barth-Maron et al. Distributed distributional deterministic policy gradients. `arXiv:1804.08617v1 [cs.LG]`, 2018.