Project 3 – Report

In this project was used the Deep Deterministic Policy Gradient (DDPG) algorithm to solve the environment. The starting point was the implementation provided by Udacity to solve BipedalWalker environment (ddpg-bipedal).

Learning algorithm description

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. DDPG can be seen as a Deep Q-Learning algorithm for continuous action domains, since the algorithm combines ideas from DPG (Deterministic Policy Gradient) and DQN (Deep Q-Network), using Experience Replay and slow-learning target networks from DQN, and it is based on DPG, which can operate over continuous action spaces.
Just like Actor-Critic methods, we have two networks (our target networks):

- Actor - It proposes an action given a state
- Critic - It predicts if the action is good (positive value) or bad (negative value) given a state and an action.

The full algorithm is described below:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

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

  end for
end for

Architectures

The architectures used in the actor and critic can be seen below:

Actor

```python
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.bn1 = nn.BatchNorm1d(fc1_units)

        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.bn1(self.fc1(state)))
        x = F.relu(self.fc2(x))

        return torch.tanh(self.fc3(x))
```

Critic

```python
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=400, fc2_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.bn1 = nn.BatchNorm1d(fcs1_units)

        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        x = F.relu(self.bn1(self.fcs1(state)))
        x = torch.cat((x, action), dim=1)
        x = F.relu(self.fc2(x))

        return self.fc3(x)
```

These architectures were found through experimentation. Among the modifications made to the original architecture used in the base code, the modification that had the greatest impact was the inclusion of a batch normalization after the first hidden layer.

Hyperparameters

The hyperparameters used can be seen below:

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 1024       # minibatch size
GAMMA = 0.99            # discount factor
TAU = 3e-3              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 1e-4        # learning rate of the critic
WEIGHT_DECAY = 0.0001   # L2 weight decay

LEARN_EVERY = 10        # Learning interval
LEARN_TIMES = 5         # Number of times to call learning function

EPS = 7
EPS_DECAY = 0.0001
EPS_MIN = 0
```

Theta and sigma from Ornstein-Uhlenbeck process was setted to 0.15 and 0.10 respectively.

```
def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.10)
```

The noise produced by the Ornstein-Uhlenbeck process is reduced over time through the EPS variable, to encourage the agent to explore the environment in early episodes, and gradually reduce exploration and to rely more on the actor's actions. EPS is reduced by EPS_DECAY, every time learn functions is called, until EPS reaches its minimum (EPS_MIN)

LEARN EVERY is used to define the number of time steps required before updating the network weights, once LEARN_EVERY timesteps in the environment have passed, the function to update the network weights (learn) is executed LEARN_TIMES, each time with a different set of experience.

Other Changes

The noise added to the state by Ornstein-Uhlenbeck process was changed, to use random values from a standard normal distribution.

```
dx = self.theta * (self.mu - x) + self.sigma * np.random.standard_normal(self.size)
```

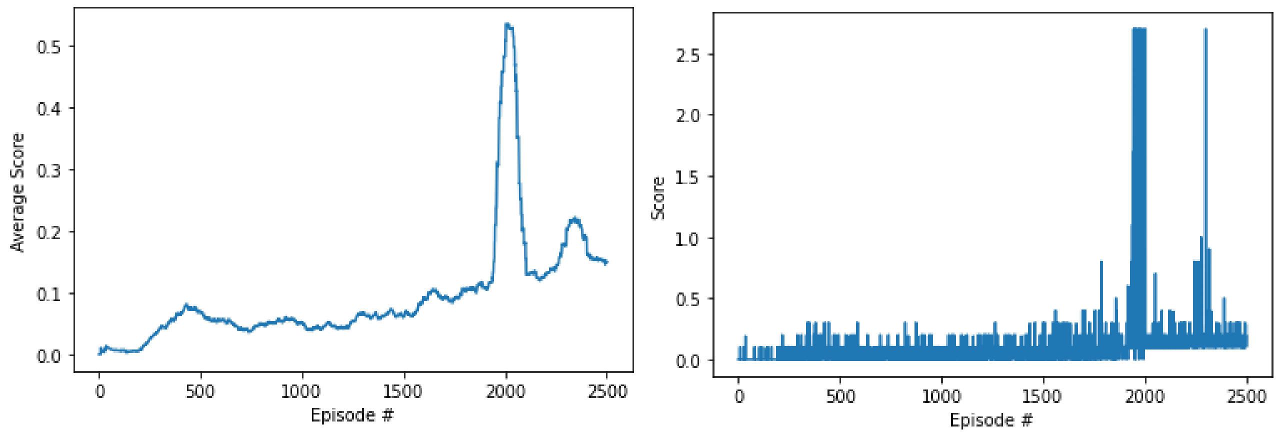
Results

The agent achieved an average score of 0.5 in 2002 episodes, after that, it continued training for more 500 episodes to see if it can still improve the average score. The best average score was achieved at episode 2005, after that, the agent's performance dropped considerably to an average of approximately 0.150 in subsequent episodes.

```
Episode 100     Score : 0.000   Average Score: 0.006
Episode 200     Score : 0.000   Average Score: 0.007
Episode 300     Score : 0.000   Average Score: 0.041
Episode 400     Score : 0.200   Average Score: 0.066
Episode 500     Score : 0.000   Average Score: 0.068
Episode 600     Score : 0.000   Average Score: 0.056
Episode 700     Score : 0.200   Average Score: 0.043
Episode 800     Score : 0.100   Average Score: 0.049
Episode 900     Score : 0.000   Average Score: 0.053
Episode 1000    Score : 0.090   Average Score: 0.051
Episode 1100    Score : 0.200   Average Score: 0.043
Episode 1200    Score : 0.100   Average Score: 0.044
Episode 1300    Score : 0.090   Average Score: 0.067
Episode 1400    Score : 0.100   Average Score: 0.062
Episode 1500    Score : 0.000   Average Score: 0.060
Episode 1600    Score : 0.100   Average Score: 0.092
Episode 1700    Score : 0.100   Average Score: 0.092
Episode 1800    Score : 0.000   Average Score: 0.103
Episode 1900    Score : 0.100   Average Score: 0.109
Episode 2000    Score : 0.090   Average Score: 0.483
Episode 2002    Score : 2.700   Average Score: 0.509

Enviroment solved in 2002 episodes!
Continuing training for more 500 episodes (until episode 2502) to see if we can
still improve average score!

Episode 2100    Score : 0.090   Average Score: 0.180
Episode 2200    Score : 0.100   Average Score: 0.129
Episode 2300    Score : 0.100   Average Score: 0.177
Episode 2400    Score : 0.100   Average Score: 0.189
Episode 2500    Score : 0.100   Average Score: 0.151
Episode 2502    Score : 0.190   Average Score: 0.150Training done!
Best average score: 0.536
Episode with best average score: 2005
```



Ideas for Future Work

During the development of the project it became clear that the algorithm is very sensitive to the hyperparameters used, so a good starting point for improving the agent would be an optimization of the hyperparameters, which would probably help the agent to solve the environment more quickly (in less episodes). Another improvement would be the implementation of prioritized experience replay, since like DQN, this algorithm used experience replay to learn agents. I also realized that training multiple times the algorithm with different seeds leads to a huge difference in performance, So the implementation of a training with multiple seeds, can help to find the solution of the environment more quickly. Finally, implementing MADDPG can also help the agent to have a more stable and fast learning.

Note

The model weights of the actor and critic used to solve the environment is saved respectively with the names best_actor.pth and best_critic.pth in the project root folder.