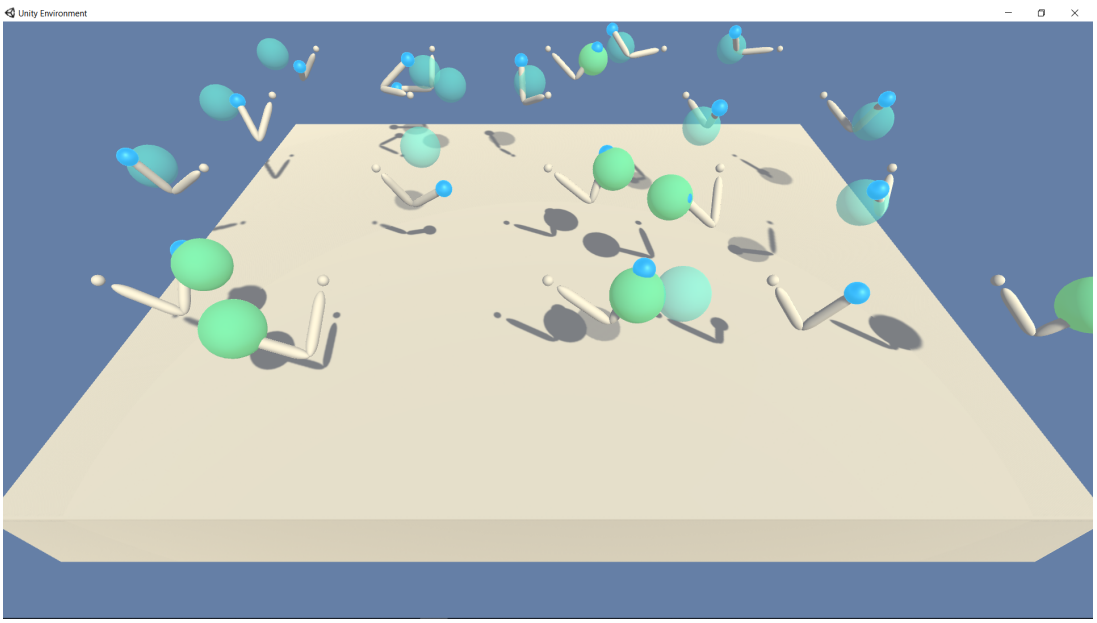
Udacity Nanodegree

Deep Reinforcement learning
Report

Project 2 : Continuous Control

Student :
Géraud MARTIN-MONTCHALIN



April 7, 2019

Contents

1 Introduction

In this project, the goal is to implement a deep reinforcement learning algorithm that can deal with continuous action space. In order to do so, I choose to use DDPG (Deep Deterministic Policy Gradient), though, I could have used PPO too, but it seems to lead to poorer results. Continuous action-spaces applies in many areas in the real world, especially in robotics which makes, I think, this project very concrete.
Our goal now, is to control the torque in the joints of a robotic arm for him to successfully reach a moving target. We will use a Unity environment made of 20 robotics arms in order to improve faster our deep neuronal networks.

1.1 Files

The *model.py* and *ddpg_agent.py* files are from the folder on ddpg-pendulum made by Udacity.

The *DDPG_continuous_control.ipynb* notebook is the part where we can train and watch a "smart" agent evolving in the Unity environment.

The *checkpoint_actor.pth* and *checkpoint_critic.pth* files are saving of the weights of the locals actor and critic. Those can be downloaded to watch a "smart" agent.

2 Description

DDPG (Deep Deterministic Policy Gradient) algorithm works as follow. First, we have the Actor which tells us what actions are probably the better to do. Second, we have the Critic which approximates the maximizer over the Q values of the next state.
Both Actor and Critic have two networks, the local and the target. The local ones are improving during each episode and are used every time the Agent learn a new batch of samples (S_t,A_t,R_t,S_{t+1}) to modify slightly the target networks towards the direction expected to have better results.

2.1 Deep Neuronal Networks

We have two separate kind of Networks, one for the Actor and one for the Critic. Both are composed of the input layer taking the state as value, two hidden layers and one output layer. The output layer of the Actor releases the action and the one of the Critic release the Q-value of the $(state, action)$ pair. The Critic can do so by implementing the action in its first hidden layer.
Then I also implemented batch normalization on the first layer of both networks in order to accelerate the training process.

2.2 Parameters

The parameters used for the training of the Agent are the following :

Listing 1: parameter display

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 3e-4         # learning rate of the actor
LR_CRITIC = 1e-4        # learning rate of the critic
WEIGHT_DECAY = 0.0001   # L2 weight decay
UPDATE_EVERY = 20       # Train the Agent every UPDATE_EVERY step
NUM_UPDATE = 10         # How many times should we update the agent
SIGMA = 0.2             # sigma of the Ornstein-Uhlenbeck Noise function
MAXSTEP = 1000          # Maximum of step by episode
```

2.3 Agent

The Agent Class is the core of the algorithm as it's where all the improvement process happens. In the Class, we had to add a loop function in the *step* function to add every agent sample in the memory. I also implemented the possibility to train a chosen number of batches every time the learning process is launched. I didn't loop over the 20 agents in the *action* function because I needed a batch of sample instead of single samples for the batch normalization in the networks.
Finally, I add the function that was suggested in the Udacity benchmark to clip the gradient during the training of the critic network:
"$torch.nn.utils.clip\_grad\_norm\_(self.critic\_local.parameters(), 1)$"

2.4 Ornstein Uhlenbeck Noise

This noise fits perfectly for continuous action-state environment because instead of gaussian noise, it is temporally correlated and so it gives smother transitions which can be rather useful for controlling the torque of a joint or the acceleration of a car for example.
In this Class, I modified the sample function by implementing a normal distribution of the noise instead of a uniform one. This implementation seemed to be a major update as it gave me directly better results on my simulations.

Listing 2: Implementation of normal distribution in the sample function

```
#np.random.random() became np.random.randn()
dx = self.theta * (self.mu - x) + self.sigma * np.array([np.random.randn() for i
    in range(len(x))])
```

2.5 Replay Buffer

This Class is where all the samples which were created during each episodes are stored in order to be re-used later during the learning process. We limit it in size because too many samples would be useless and would only slowing down the computer.

3 Training

In this part we will see different approaches which can leads to bad and good results.

3.1 First implementations

At first, I was unable to reach higher score that 1 because I was clearly overfitting my Agent. Then I found that the $WEIGHT\_DECAY$ parameter was blocking me too to reach higher score, so I disabled it.
Finally, before implementing the normal distribution on the sample function of the OUNoise Class, my algorithm was unable to go over a score of 15 whatever my $BATCH\_SIZE$ was in between 64 to 1024 as we can see on the following figures.

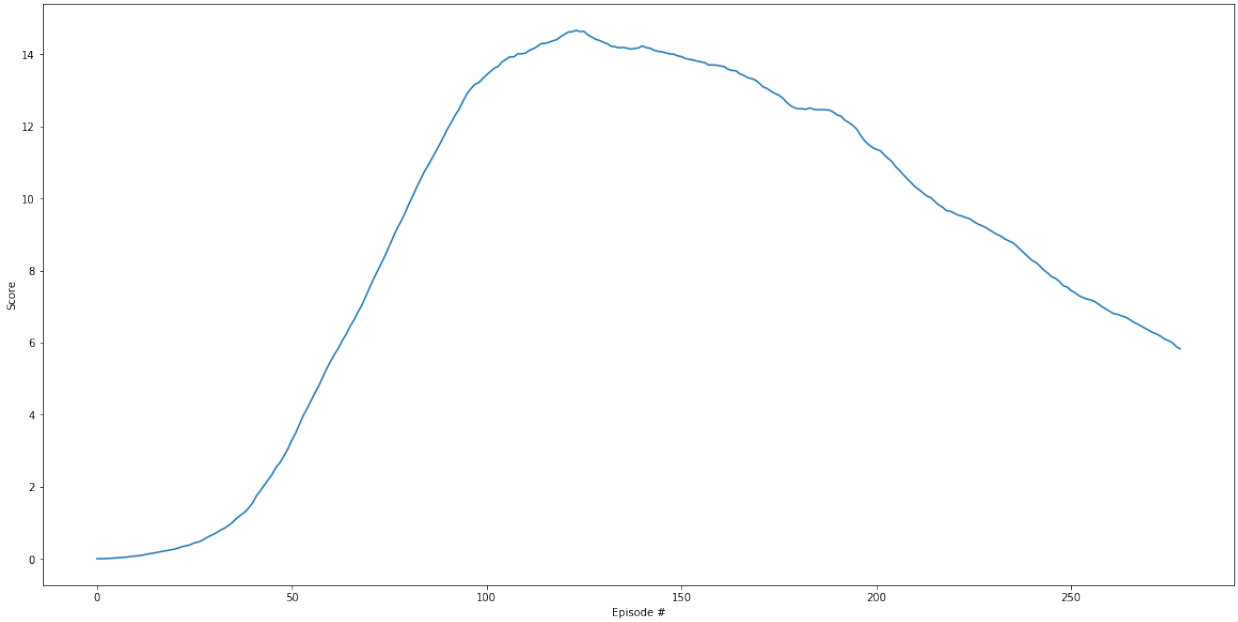


Figure 1: Results for $BATCH\_SIZE = 256$, $LR\_ACTOR = 3e-4$,$LR\_CRITIC = 1e-4$

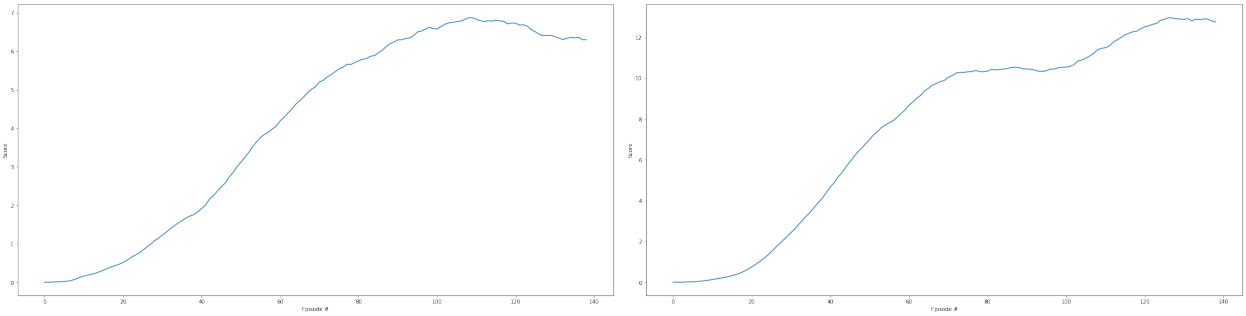


Figure 2: Results for $BATCH\_SIZE = 128$ and $BATCH\_SIZE = 1024$

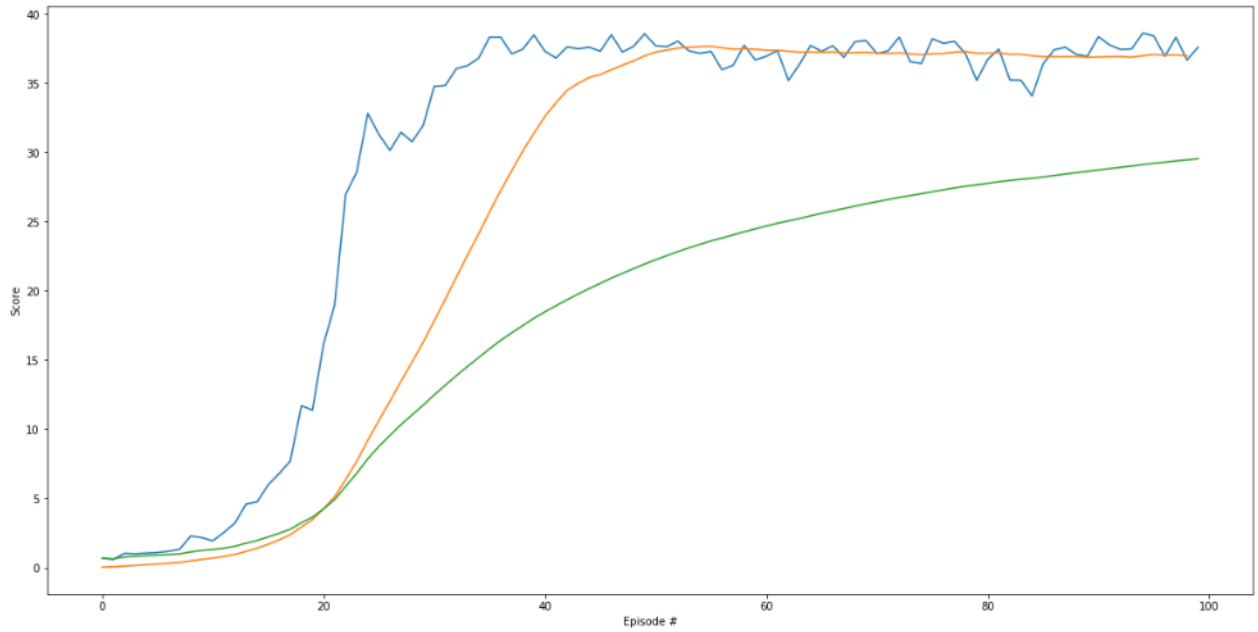
3.2 Succesful implementations

Then, I took the time to read carefully the papers and search online for solutions which made me implementing Batch Normalization and Ornstein-Uhlenbeck Noise Normalization. I did many simulations to see the effects of the parameters and I found that I had pretty good results with the following parameters:

Listing 3: parameter display

```
BUFFER_SIZE = int(1e5)  # 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-4         # learning rate of the actor
LR_CRITIC = 1e-4        # learning rate of the critic
WEIGHT_DECAY = 0.       # L2 weight decay
UPDATE_EVERY = 1        # Train the Agent every UPDATE_EVERY step
NUM_UPDATE = 1          # How many times should we update the agent
SIGMA = 0.2             # sigma of the Ornstein-Uhlenbeck Noise function
MAXSTEP = 1000          # Maximum of step by episode (at least 1000)
```

With these updates, the environment was solved before the 10th episode. In the following simulation, the environment has even been solved in the second episode.




```
Episode 101 Average Score deque: 29.90  score: 36.54 time: 4342.916607618332
Episode 102 Average Score deque: 30.27  score: 37.89 time: 4386.265207529068
envionment solved
```

Figure 3: Sucessful Training

In this figure, the blue curve is the score at each episode, the orange one is the smoothed score with a window of 20 (it's the average of the twenty last episodes' scores) and the green curve is the mean score over the last 100 episodes(score_deque).

4 Conclusion

All in all, this project has given me the ability to enhance a lot my understanding of DDPG algorithm and even more of continuous action-states. I learned also a lot about how the tuning of each parameters could influence the whole training task.
I will surely try now to implement DDPG on the Crawler. It would be interesting two to use D4PG or PPO to solve this environment. Prioritized experience replay should be useful as well.

Listing 4: Papers which were usefull for this project:

BATCH NORMALIZATION:
https://arxiv.org/abs/1502.03167

CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING:
https://arxiv.org/abs/1509.02971

PARAMETER SPACE NOISE FOR EXPLORATION:
https://arxiv.org/pdf/1706.01905.pdf