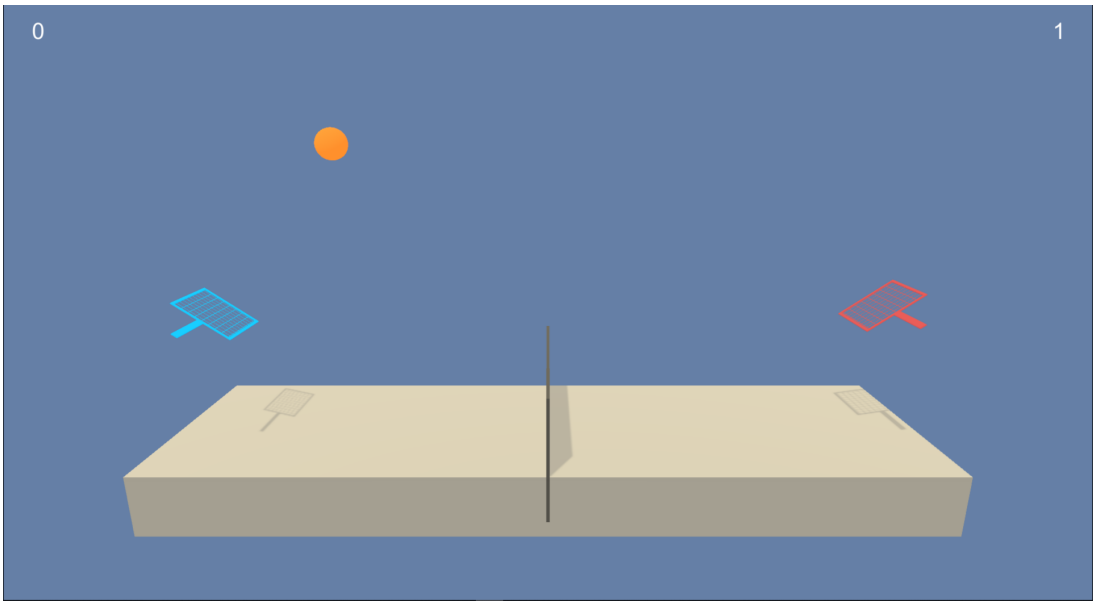
Udacity Nanodegree

Deep Reinforcement learning
Report

Project 3 : Colab-Compet

Student :
Géraud MARTIN-MONTCHALIN



May 5, 2019

Contents

1 Introduction

In this project, the goal is to implement a deep reinforcement learning algorithm that can deal with continuous action space and multiple agent working together. In order to do so, I choose to keep using DDPG (Deep Deterministic Policy Gradient).
our goal is to teach two Tennis rackets to pass each other a ball over a fillet.
The two rackets will use here the same actors and critics network. They will also use the same buffer to keep their past experiences.

1.1 Files

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

The *TENIS.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.

2.2 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 add the function that was suggested in the Udacity benchmark of the second project to clip the gradient during the training of the critic network:
"$torch.nn.utils.clip\_grad\_norm\_(self.critic\_local.parameters(), 1)$"

2.3 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.

Listing 1: 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))])
```

2.4 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 the results we had while training the agent. The parameters used for the training of the Agent are the following :

Listing 2: parameter display

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # 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 = 5000          # Maximum of step by episode
```

With these updates, the environment was solved between 1500 and 2000 episodes depending on the simulation. In the following one, the environment has been solved in about 1620 episodes.

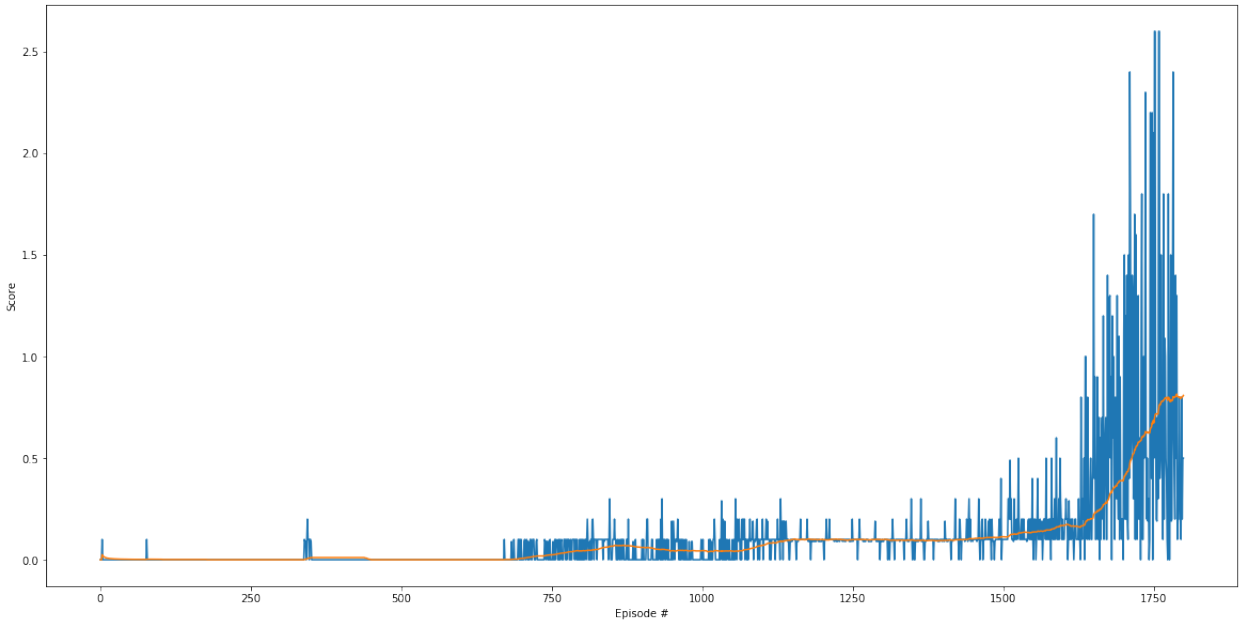


Figure 1: Sucessful Training

In this figure, the blue curve is the score at each episode and the orange one 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 multiple agent system. 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 it on the Socccer environment.