Tennis Multi Agent DDPG for the competition & collaboration problem

Guillermo del Valle Reboul

1 Algorithm Used

The **Deep Deterministic Policy Gradient (DDPG),** a model-free off-policy algorithm for learning continuous actions, was used for this project based in a Unity Reacher environment. It combines ideas from DPG (Deterministic Policy Gradient) and DQN (Deep Q-Network). It uses Experience Replay and slow-learning target networks from DQN, and it is based on DPG, which can operate over continuous action spaces.

The Actor Critic model is based in simple and relatively shallow neural networks.

- For the local and target ***Actor Neural Network***, 3 Linear layers fully connected are used with sizes "state_size", 256 and 128 and Relu activation functions.
- For the other neural networks of the ***Critic Neural Network***. Also 3 Linear fully connected layers. The sizes are "state_size", 256 with the action attached and 128, with no output activation function in the last layer. The use of ReLU seemed fine to achieve a good level of performance.

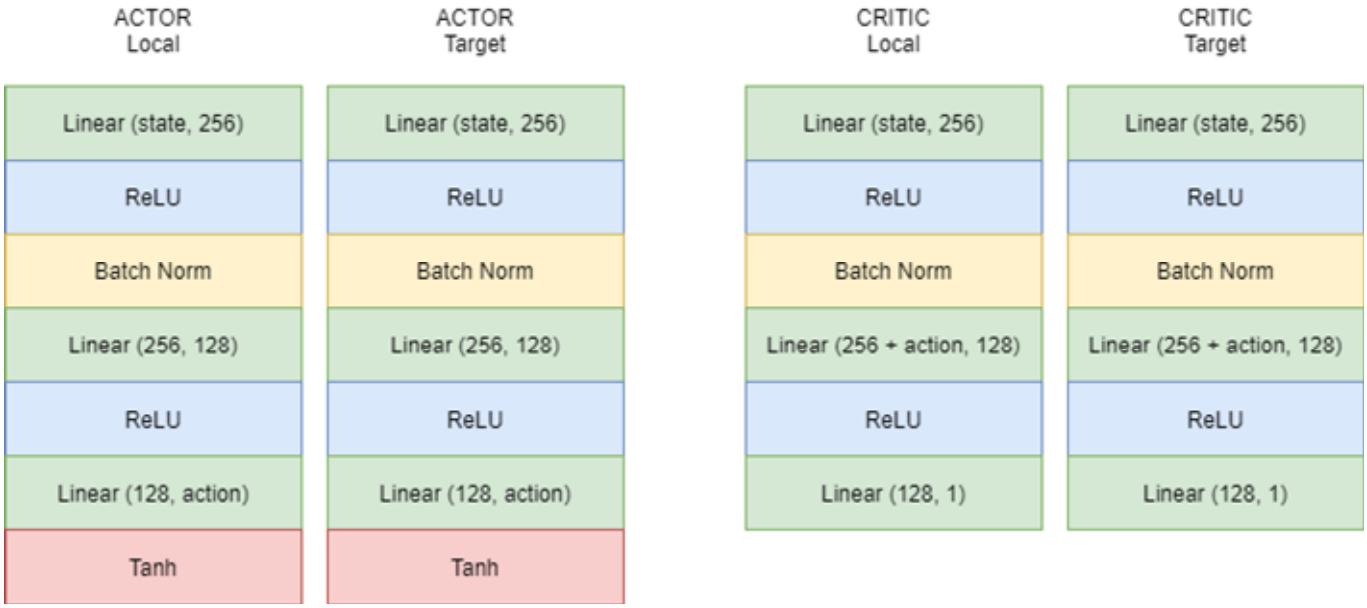

Figure 1-1. Model of the deep NN for Actor and Critic.

1.1 Collaboration and Competition Concept

The idea of two or more agents competing against one another and learning as a team at the same time seems really fascinating. How to achieve this? Very simple. An experience replay buffer is shared between the two DDPG agents. They both contribute to the buffer adding experiences and then, after several iterations, they randomly sample from its memory to learn from those experiences again. This proved to be the key element for achieving convergence and a good performance at the end of the training.

2 Hyperparameters

```
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 = 2        # Update every

LEARN = 3               # Learn several experiences
```

3 Files Used

▫ Tennis.ipynb: used in training.

▫ tennis_main.py: script that fulfils the same purpose than the notebook.

▫ maddpg.py: This file loads several DDPG Agents and interacts with them.

▫ ddpg_model.py: contains the DDPG Network model used for both target and local networks in actor-critic architecture.

▫ ddpg_actor_critic_model.py: contains the agent with the DDPG algorithm.

▫ Folder /final: contains the weights saved from training the actor/critic 0 and the actor/critic 1.

▫ README.md: for instructions.

▫ Folder /results: folder that contains results of the agent's performance.

4 Plot of Rewards

4.1 Multi Agent DDPG

MADDPG Solved the environment in 1031 episodes (Average score = 0.5166 > 0.5).

```
Episode 1031    Average Score: 0.5166
```

```
In [6]: fig = plt.figure()
        ax = fig.add_subplot(111)
        plt.plot(np.arange(1, len(scores)+1), scores)
        plt.ylabel('Score')
        plt.xlabel('Episode #')
        plt.show()
```

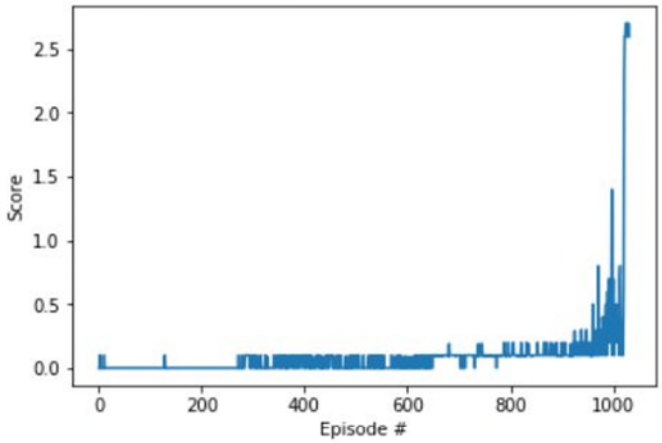

Figure 4-1. Summary of the agents' performance.

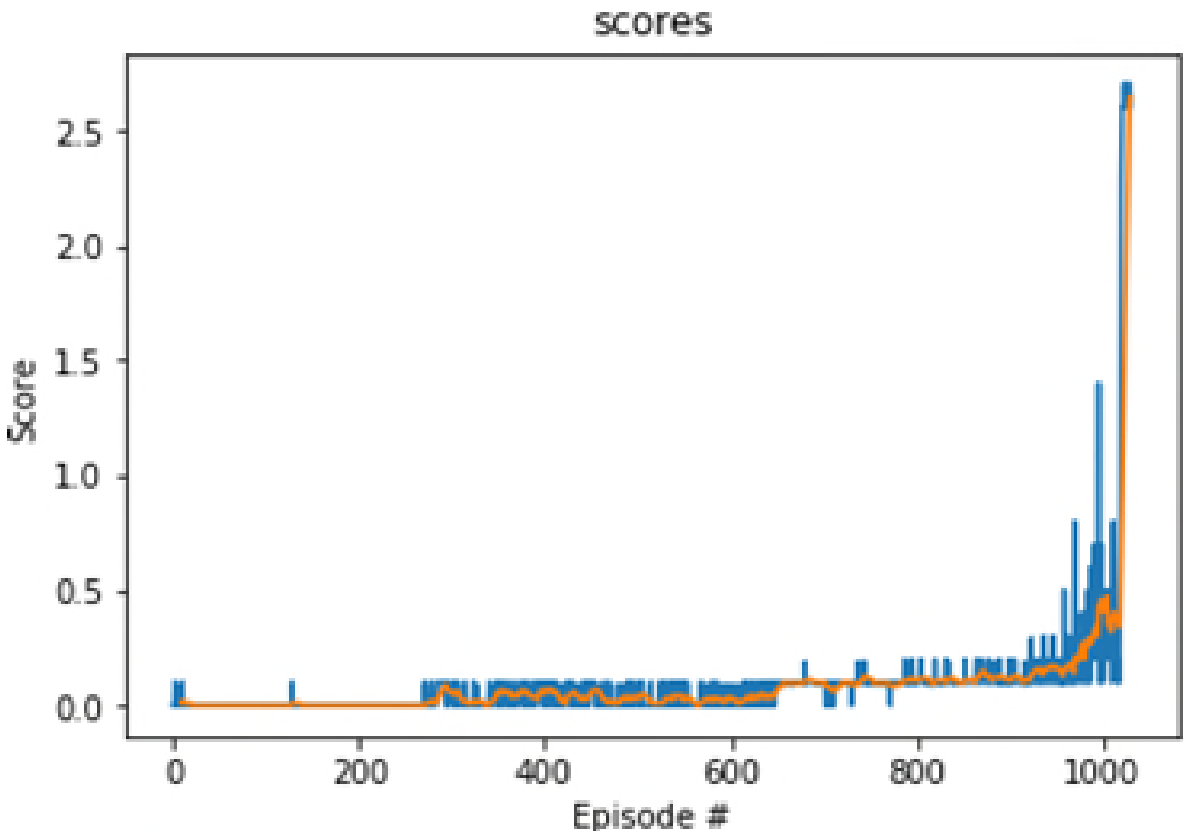

Figure 4-2. Plot of the MADDPG agents' performance over the 1031 episodes.

5 Lessons learnt

- The Actor and Critic used both a **3-layer neural network** with a relatively low learning rate (1e-4) which proved to be the *key element* that made the agent learn faster and achieve high rewards. Deeper networks did not seem to improve performance.
- The correct implementation of the act, step and learn methods for the **MADDPG Agents** was extremely important for **correct behaviour of the agents**, as well as the correct interface with the environment, the state variables, and the continuous action space.
- The learning took approximately 800 episodes to increase at a good rate. Choosing **random actions** for the first couple of hundred episodes did not seem to increase learning much.
- The introduction of **Batch Normalization** proved to be key for the consistent learning of the agents.
- The correct use of the shared **Replay buffer** ended up being essential for achieving convergence.
- The addition of Ornstein-Uhlenbeck Noise proved to be a little bit problematic for me in the past, and for that reason it was replaced by a standard random noise. This addition of noise to the action was important for the learning process. Also, I applied noise decay.
- The use of an update every 2 steps in the agent's neural networks proved to be important for achieving convergence. Every 2 steps, the actor and critic networks would pick data from the memory buffer to learn and update their weights (3 samples of experience).
- The use of a 128 Batch size seems enough for achieving converge.

6 Ideas for Future implementations

- Fine tuning the hyperparameters as well as trying shallower or deeper neural networks so that the learning speed can be improved.
- The use of another algorithm like D4PG can be explored as well as PPO, A2C, A3C.