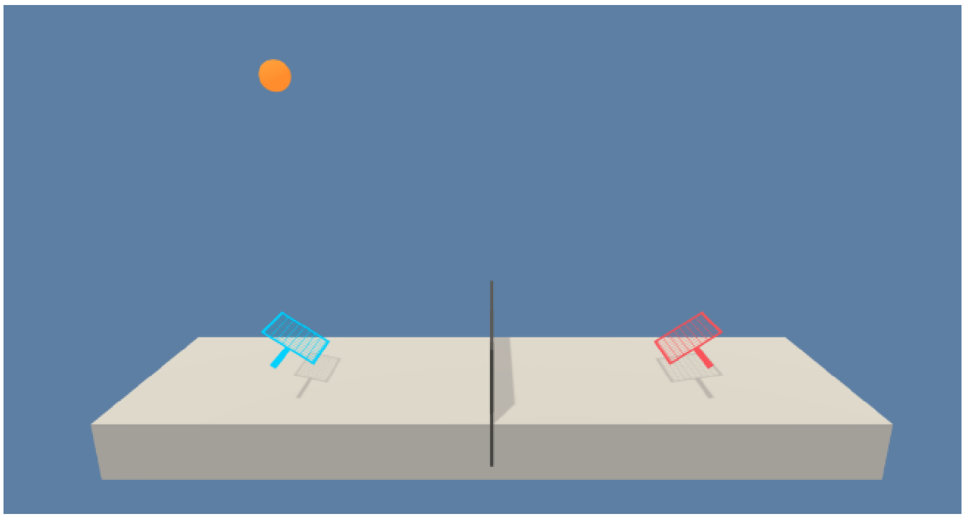
Collaboration and Competition

Fabrice R. Noreils

Introduction



For this project we are working with the Tennis Environment. In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, the agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

- After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- This yields a single **score** for each episode.

The environment is considered solved, when the average (over 100 episodes) of those **scores** is at least +0.5.

Description of the solution

I decided to create a single MADDPG multi-agent class that will handle the two DDPG agents.

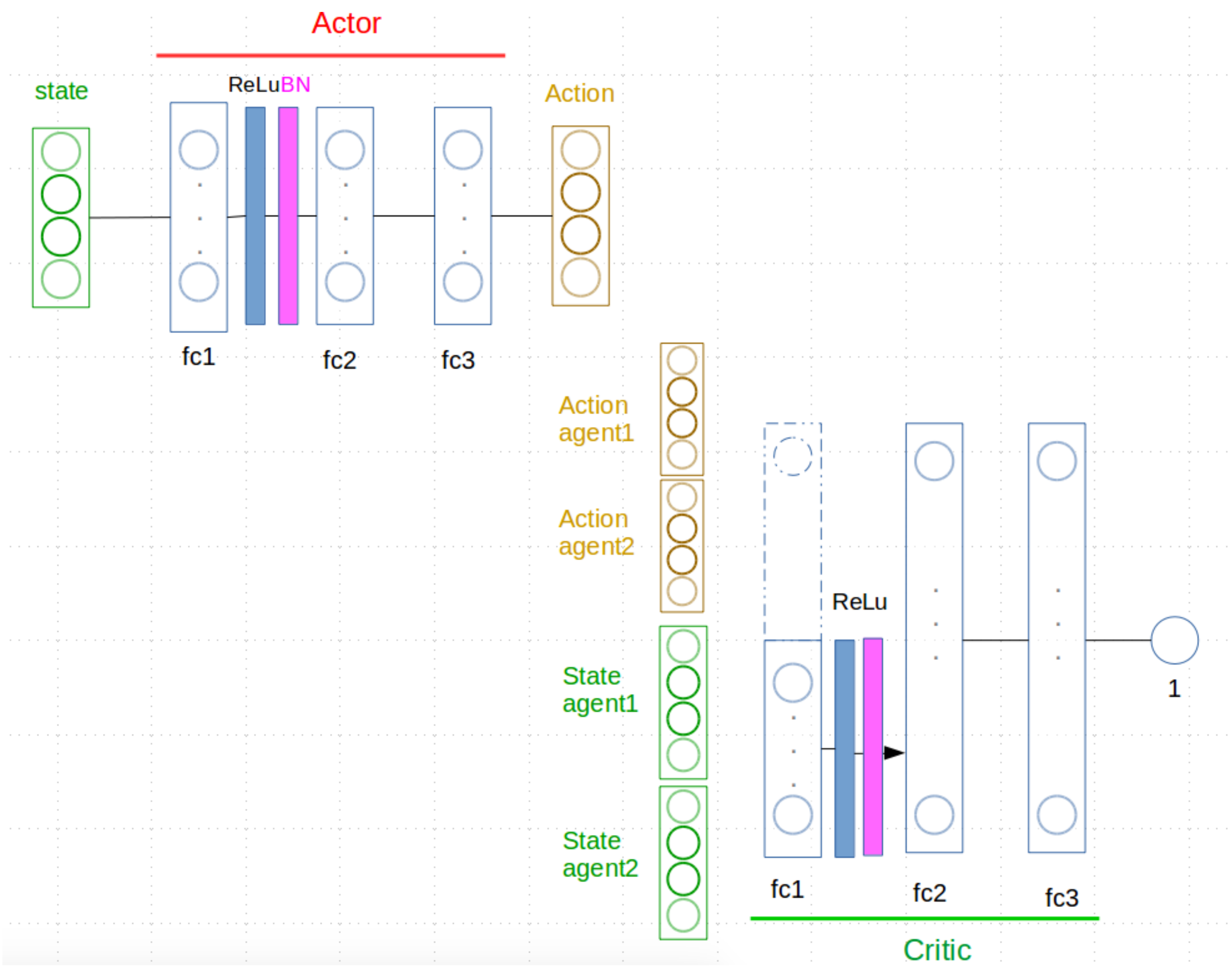
Each DDPG agent is provided with an Actor and a Critic. As we are working with several agents, the Critic takes in input all agent states and actions as it is shown on the picture below.

I tested two cases for the first FC layer of the Critic :

- First FC layer takes as input the state of both agents and then the action of both agents are concatenated with the output of the first layer to feed the second FC layer ;
- The state and action of both agents are concatenated and fed into the first FC layer directly.

I did not see any difference in terms of result so I continued with the second solution.

I kept the BN layer after the first FC layer.



Picture 1: Description of the Actor and Critic networks.

The two main equations of the MADDPG algorithm [1] are described below :

For each $Agent_j$:

Sample then :

$$\text{Set } y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k=\boldsymbol{\mu}'_k(o_k^j)}$$

Picture 2: Equation for the Critic - from [1]

This equation (Picture 2) tells us that for each next state, we need to compute the corresponding next action with the relevant target actor network. Then **all** the « next states » and the associated « next actions » are fed into the target critic network of the **current agent**. However it is important to notice that we do not add any noise here (I tested it and it lead to nowhere).

The piece of code below is computing the $a'_k = \mu'_k(\sigma_k^j)$ (picture 2) and $a_i = \mu_i(\sigma_i^j)$ (picture 3)

```
for agent_id in range(2):
    # get the torch tensor
    torch_agent_id = torch.tensor([agent_id]).to(device)
    # extract agent i's state and get action via actor network
    state = states.reshape(-1, 2, 24).index_select(1,torch_agent_id).squeeze(1)
    action = self.maddpg_agent[agent_id].actor(state)
    all_actions.append(action)
    # extract agent i's next state and get action via target actor net
    next_state = next_states.reshape(-1, 2, 24).index_select(1,
torch_agent_id).squeeze(1)
    next_action = self.maddpg_agent[agent_id].target_act(next_state)
    all_next_actions.append(next_action)
```

$$\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$$

The loss is computed with these lines of code :

```
q_target_next = agent.target_critic(next_states, all_next_actions)

#compute TD error
q_target = rewards.index_select(1, agent_id) + (self.discount_factor * q_target_next * (1
- dones.index_select(1, agent_id)))
# concatenate because the fc layer takes actions and states as input
critic_input = torch.cat((states, actions), dim=1).to(device)
q_expected = agent.critic(critic_input)
critic_loss = F.mse_loss(q_expected, q_target.detach())
```

$$\nabla_{\theta_i} J \approx \frac{1}{S} \sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i=\boldsymbol{\mu}_i(o_i^j)}$$

Picture 3: Equation for the Actor – from [1]

This equation (Picture 3) tells us that for each state, we need to compute the corresponding action with the relevant actor network. Then **all** the « states » and the associated « actions » are fed into the critic network of the **current agent**. However it is important to notice that we do not add any noise here !

Regarding the Action, the loss is computed with these lines of code :

```
# detach actions of the other agent to avoid issues in the computation graph
q_input = [ actions if i == agent_number else actions.detach() for i, actions in
enumerate(all_actions) ]

q_input_ = torch.cat(q_input, dim=1)
q_input2 = torch.cat((states, q_input_), dim=1)
actor_loss = -agent.critic(q_input2).mean()
```

Choose carefully the loss function ! I started with SmoothL1Loss() and I lost lot of time (well not really because it forced me to dive in the algorithm) because even after 2000 episodes I was not able to cross the 0.2 score level.

So I was thinking that, may be, I experienced an exploding or vanishing gradient. To check it I stored the actor and critic gradients for further investigation (see Result Section) but in fact my problem lied in the wrong loss function.

I switched to `F.mse_loss` and miracle, it worked straightaway and well because I solved the environment in 800 episodes.

Regarding the parameters, all FC layers are set to 256.

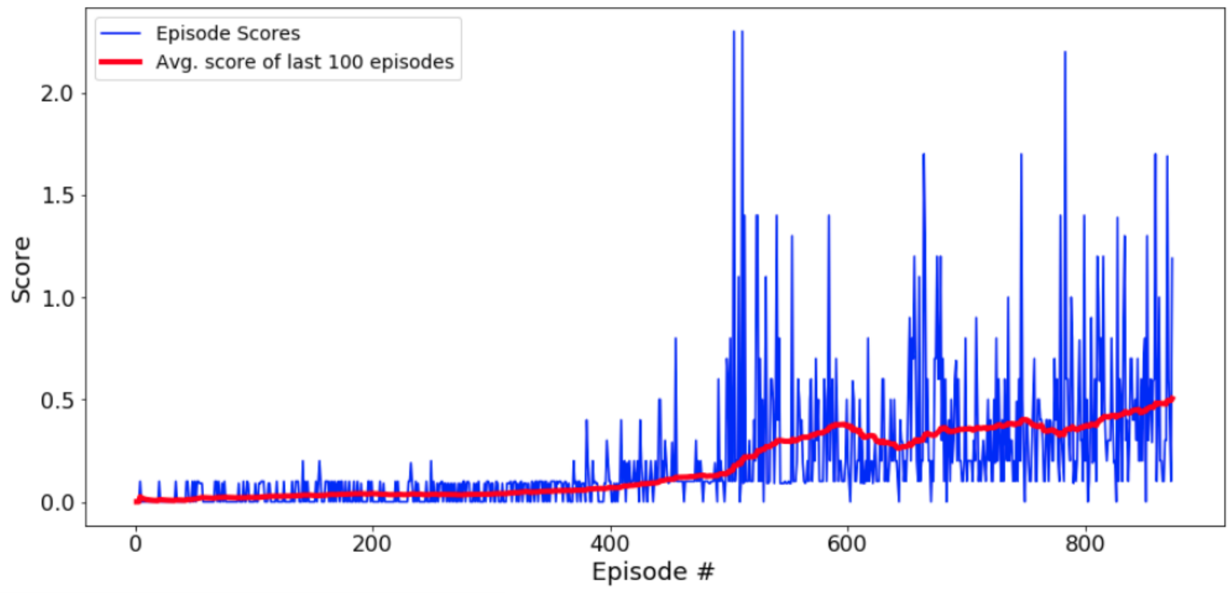
The batch size is set to 396 because it provides a better result than 256 or 512.

I did not touch to the other parameters.

As we will see in the Result Section, my feeling is that the performance of the algorithm is strongly related to the sampling or how many time you sample and run the agent and update the net weights.

Result

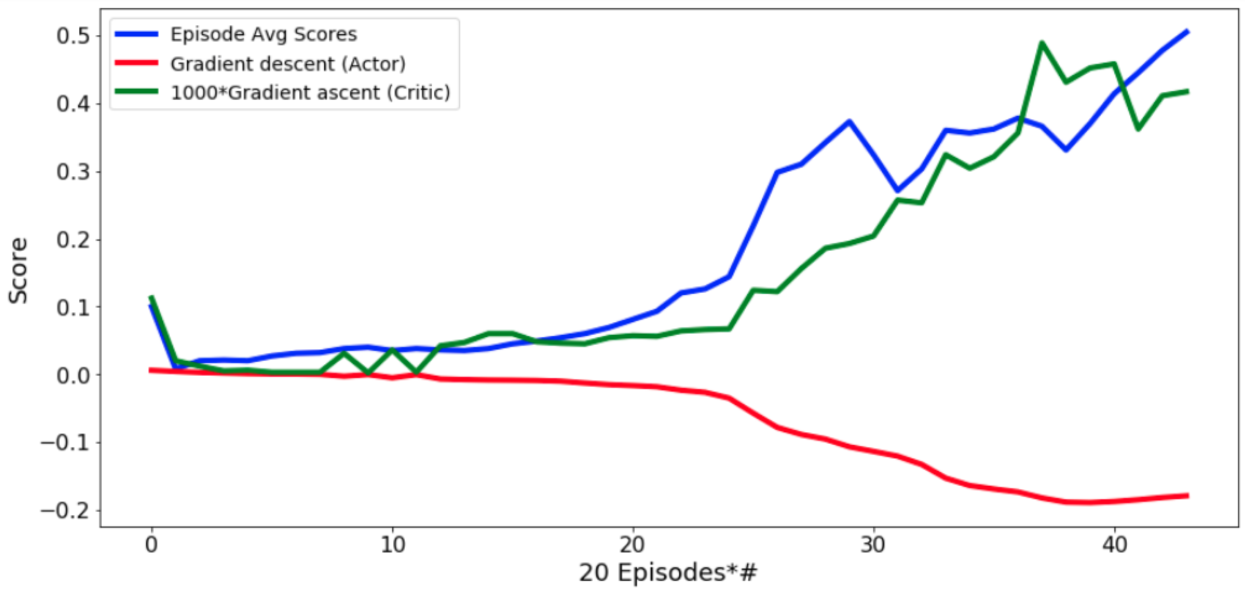
As it is shown below, the environment has been solved in 870 episodes.



Picture 4: Display of Episode scores and Avg score for the last 100 Episodes

As I stored the actor and Critic gradients, I printed them (see picture 5 below) and I can make the following observations :

- Although the average score was stalled below 0.4 for around 200 episodes, I can see that the DDPG agents were still learning as the Actor gradient went more negative and the Critic gradient went up ;
- The slope of the Actor gradient started to become positive when the average score was near 0.5 and it could be interesting to check the behavior of the gradients if I let the algorithm running few hundred more episodes. Unfortunately, as I am running on CPU, it takes a lot of times. The algorithm is performing very badly if I turn on GPU and I cannot figure out why… If you find the bug please tell me !



Picture 5: Display of Actor and Critic gradients

My feeling is that the key to improve the performance of the algorithm is **when to do sampling.** Indeed when the algorithm is called many times during a sequence of episodes, the average score improves really a lot. But if we sample too much at the beginning, it is really bad – the score can quickly goes to 0.002/0.004 and stays there for a while. So the question is to find when to increase the sampling (call MADDPG several times) during the training and I did not find out so far… But this is what I will do as a next step.

[1] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., & Mordatch, I. (2017). Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *NIPS*.