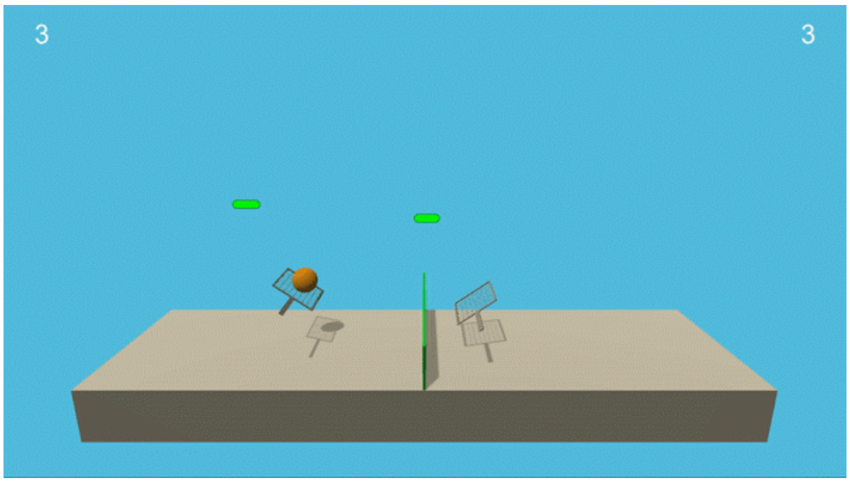
Project 3 - Collaboration and Competition

Deep Reinforcement Learning Nanodegree

M. Lippl

March 19, 2021



1 Introduction

The challenge of this project is to train two agents to play table-tennis against each other. We are given two rackets and one ball, and the goal of each match (which we consider as an episode) is to let the ball alternate between both halves of the table as long as possible. During each episode, each agent is given as score which is calculated as follows:

(i) +0.1 points if the agent hits the ball over the net,

(ii) -0.01 points if the agent either lets the ball hit the ground or hits the ball out of bounds.

An episode is considered finished if the ball either hits the table or leaves the environment to the sides. The final score of the episode is calculated as the maximum of both agents' scores. The environment is considered solved if the agents get an average score of +0.5 points over 100 consecutive episodes. In formulæ, our goal is to achieve $\text{score}_i \geq 0.5$, where

$$\text{score}_i = \frac{1}{100} \sum_{j=0}^{99} \max(a_{i-j}, b_{i-j}),$$

and a_i and b_i are the scores of each agent in episode i.

The (local) observation space for each agent consists of 8 dimensions. These are the position and speed of the ball (4 dimensions), and position and speed of the own racket (again, 4 dimensions). Notice that each unity agent returns a tuple of 24 floats for each agents, where all but four entries are set to zero. On the other hand, the action for each agent consists of two continuous dimensions, namely movement to and from the net and jumping, again only for the agent's own racket.

2 Methodology

One of the main problems with multi-agent settings is that with completely independent agents, the environment appears to be non-stationary from the viewpoint of each agent. Therefore the convergence criteria for the various common RL-algorithms (Q-Learning, etc.) do not apply.

Therefore, we follow the algorithm called Multi-Agent Deterministic Policy Gradient (MADDPG) presented in [1]. Here, each agent's actor network is trained using only it's own locally observable state and action, whereas each agent's critic is trained using the full data from all the agents.

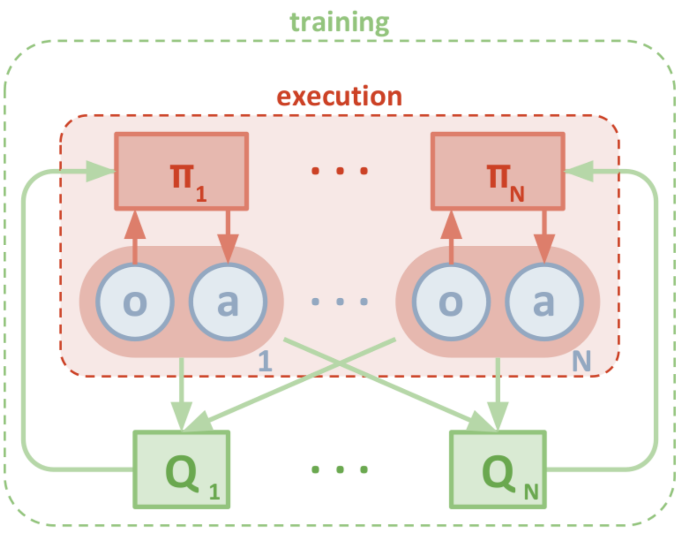


Figure 1: The multi-agent decentralized actor, centralized critic [1]

2.1 Multi-Agent Deep Deterministic Policy Gradient

The main idea is to centralize training and decentralize execution. In the case of MADDPG, this means that the agents receive full information (all actions and ideally all the agent's state information) while training but only their partial observations during execution.

Consider N agents with policies parameterized by $\theta = \{\theta_1, \dots, \theta_N\}$ and $\pi = \{\pi_1, \dots, \pi_N\}$ be the set of all policies. Then the gradient of the expected return

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim, a_i\sim}\left[\nabla_{\theta_i} \log \pi(a_i|o_i) Q_i^{\pi}(x, a_1, \dots, a_N)\right]$$

Here, Q_i is the centralized action-value function for the agent i which takes some state information x and the actions of *all* agents.

Setting the target

$$y^j = r_i^j + \gamma Q_i^{\mu'}(x'^j, a'_1, \dots, a'_N)\Big|_{a'_k = \mu'_k(o_K^k)},$$

we may define the loss function for agent i as

$$\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \Big(y^j - Q_i^\mu(x^j, a_1^j, \dots, a_N^j)\Big)^2.$$

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i} Q_i^\mu(x^j, a_1^j, \dots, a_i, \dots, a_N^j)\Big|_{a_i=\mu_i(o_i^j)}$$

At the end, there is again a soft update of the target net:

$$\theta^- \longleftarrow \tau \cdot \theta_i + (1-\tau)\cdot \theta_i^-$$

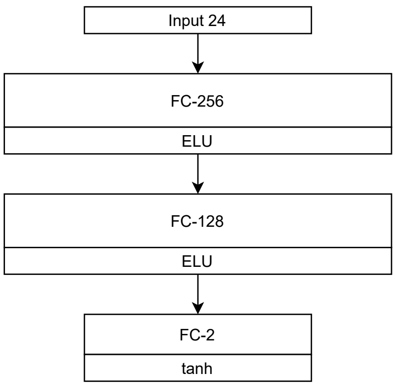
2.2 Structure of the Project

As the previous projects before, our project has the simple structure:

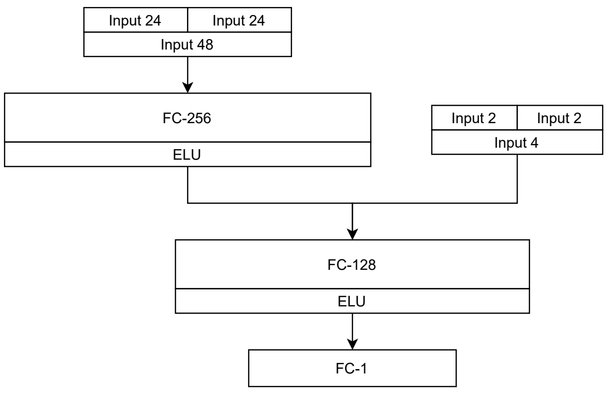
```
udacity-rl-p3/
├── Tennis.ipynb
├── agent_torch.py
├── networks_torch.py
├── train.py
└── play.py
```

The Jupyter-notebook is self-contained and contains the results of the last experiment. However, the different modules of the program are also contained in separate python-files. In particular, `agent_torch.py` contains the definition of the MultiAgent- and Agent-classes, whereas is `networks_torch.py` we define the actor- and critic-networks. Additionally, `train.py` is the script which contains main training loop. Finally, `play.py` contains a short loop which enables one to replay the agent without learning.

2.3 Architecture



(a) Actor



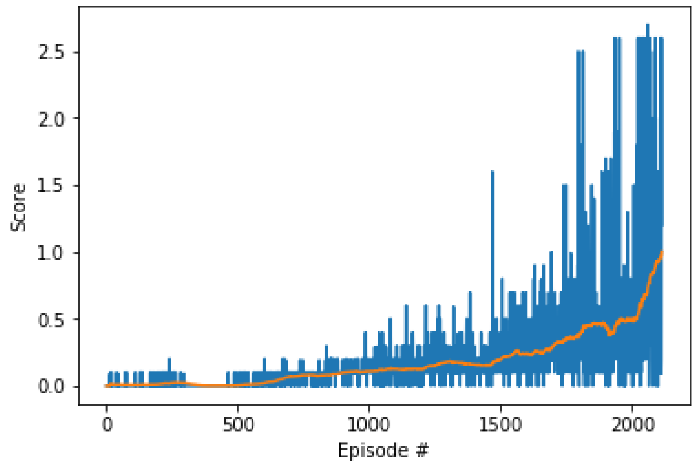
(b) Critic

Figure 2: Network architectures used in the implementation

2.4 Results

The code and results of our experiments are shown in the Jupyter-Notebook `https://github.com/asiopueo/udacity-rl-p3/Tennis.ipynb`.

As one can see in the following graph, the game was solved after about 2000 episodes. Training time for each episode was lower than a second which meant that training was manageable.



2.5 Further Research

Technical implementation using TensorFlow instead of PyTorch.

References

[1] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," 2020.