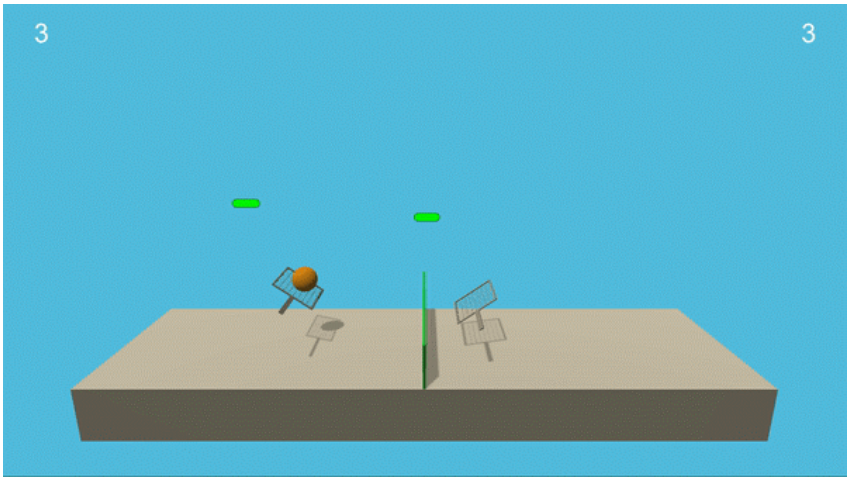
# Project 3 - Collaboration and Competition

Deep Reinforcement Learning Nanodegree

M. Lippl

March 21, 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.

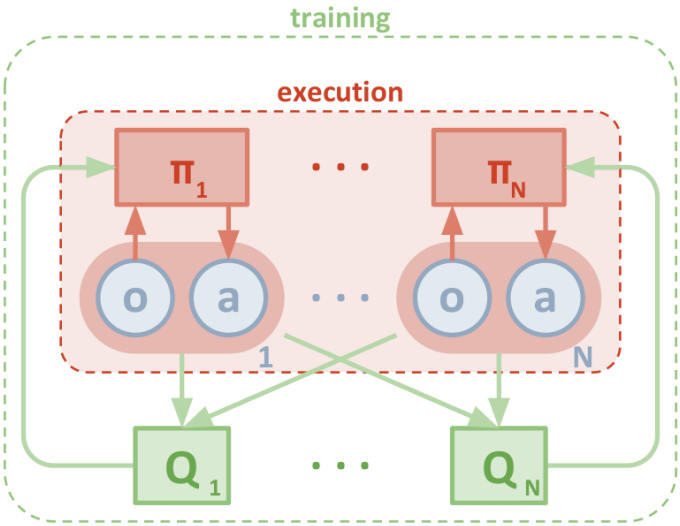



Figure 1: The multi-agent decentralized actor, centralized critic [1]

## 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. Notice that the (theoretical) gradient of the expected return is given by

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

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. The algorithm can then be sketched as follows: During each training step, we select the actions according to the parameterized policy and added stochastic noise and observe the rewards and next steps as usual, inserting them into the replay buffer. Then, for the training step, for each agent and each minibatch of $S$ samples

$(x^j, a^j, r^j, x'^j)$, we set the TD-target as

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

and define the loss function for agent $i$ as

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

The critic shall be updated accordingly. The actor is then updated according to the gradient

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

At the end of each episode, 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/
├── checkpoints_torch/
├── 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 in `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.[1]

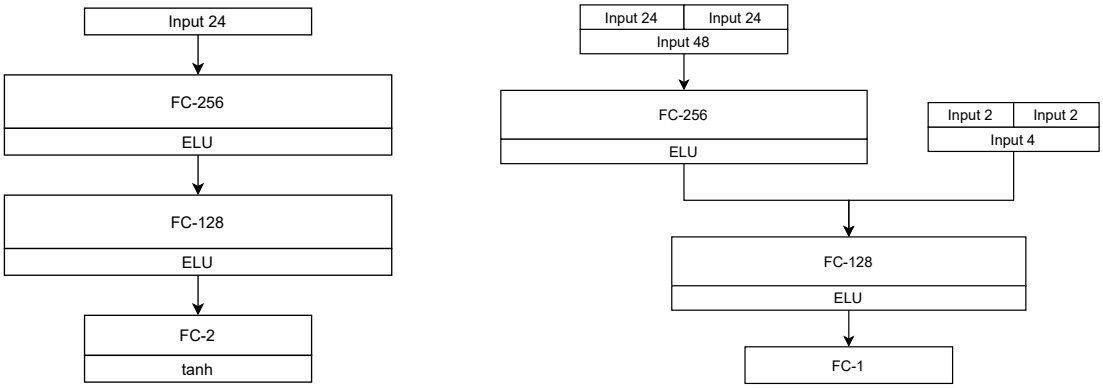
## 2.3 Implementation and Network Architecture

The architectures of the actor - and critic-networks are given in figures 2a and 2b, respectively.

Notice that there were a few tweaks in the algorithm necessary which we have learned from the Udacity-community. Among them are that we let the agents execute a number of random matches (defined by the hyperparameter `EPISODES_BEFORE_TRAINING`) before we let the agents learn. This helps supplying the replay buffer with crucial experiences first. In addition, when learning has started, for each episode we let the agents train a multiple number of times (parametrized by `LEARN_PER_CYCLE`) with different samples.

Also, it shall be noted that although we have implemented Ornstein-Uhlenbeck-Noise, a simple Gaussian process seemed to be more efficient.

[1]Note that, as with projects 1 and 2, the suffix `_torch` stems from the fact that there have been additional efforts to implement the algorithms using TensorFlow 2.x instead of PyTorch.




(a) Actor (b) Critic

Figure 2: Network architectures used in the implementation

| Hyperparameter | Value | Description/Comment |
|---|---|---|
| BUFFER_SIZE | 10,000 | Total size of replay buffer |
| BATCH_SIZE | 256 | Size of training batchs ("minibatches") |
| GAMMA | 0.98 | Discounting factor |
| TAU | 0.001 | Soft updates |
| learn_rate_actor | $10^{-4}$ | Learning rate of the Actor's Adam-optimizer |
| learn_rate_critic | $3 \cdot 10^{-4}$ | Learning rate of the Critic's Adam-optimizer |
| seed | 5 | Manual seed for neural networks |
| epsilon | 0.05 | Final value during training |
| LEARN_PER_CYCLE | 3 | Number of times learning is triggered per episode |
| EPISODES_BEFORE_TRAINING | 300 | Number of episodes before learing is permitted |

Table 1: Hyperparameters chosen for our implementation of MADDPG

## 2.4 Hyperparameters

The hyperparameters were chosen as shown in table 1.

## 2.5 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 figure 3 or in the correspongin output of the training loop in the Jupyter-notebook, the game was solved after 1965 episodes. Training time for each episode varied from a few seconds to almost a minute, depending on how long each episode ran.

As the students were warned in the briefing for the project, the training is apparently instable as can be observed in the recessions of the scores at about 1500, 2000, and especially at 2300 episodes. A 'global' crash of the agents' performance to zero has not been observed, but cannot be ruled out if the training would have continued beyond 2500 episodes. It shall also be noted that some training sessions with the exact same configuration were notably slower until the trailing average crossed +0.5 points.

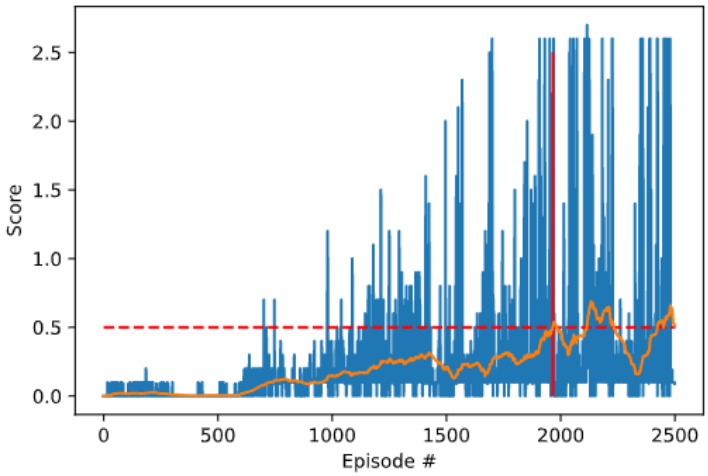



Figure 3: Resulting graph of a successful run of the implemented agent

### 2.6 Further Research

First of all, although the challenge has been solved, we believe that the current implementation still has potential for more experimentation and variation of parameters. At least a further tuning of the parameters to achieve a higher maximum of the trailing average score should be possible, as Udacity's benchmark solution has reached a maximum of +0.9148, for example. Furthermore, there might be some room for improvement by investigating the different stochastic processes to explore the agent's state-action-space. In terms of understanding further alternatives, a recent paper [2] by Google Brain which has just recently been suggested by a member of the Udacity learning community, might prove to be helpful.

As with the previous two projects, we will continue our work on the implementation using TensorFlow 2.x, in particular exploiting the `tf.Gradienttape`-feature.

## 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.

[2] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem, "What matters in on-policy reinforcement learning? a large-scale empirical study," 2020.