

# Deep Reinforcement Learning Nanodegree

## Report

### Project 3: Collaboration and Competition

#### Introduction

In this project an Multi-Agent Actor-Critic Method of Reinforcement Learning is utilized for training a agent in a 3D simulated environment. The agent is in charge of two table tennis rackets and their main goal is to act collaboratively and maintain the ball in play for as long as possible, until a certain score is reached.

The project is composed by five files:

Tennis\_Train.ipynb and Tennis\_Test - a Jupyter Notebook file containing the main code to initialize dependencies, environment and Agents;

maddpg\_agent.py - a code that contains the characteristics of the multi-agent and how it behaves through this task

model.py - containing the deep neural networks (DNN) architectures used by the agent;

checkpoint\_actor.pth and checkpoint\_critic.pth files with saved DNN's weights, that solved the environment.

The problem trying to be solved is modeled as a Markov Decision Process, involving mappings from states to actions, called policy, in such way that these actions will maximize the total cumulative reward signal of the agent. States are any information that the environment provides the agent, excluding the reward signal. Actions are ways that an agent can interact with the environment, and rewards are signals that the agent receive after interacting with the environment, shaping its learning.

The solution to the problem, on this project, is obtained by utilizing a Policy-Based Method called Multi-Agent Deep Deterministic Policy Gradient (MADDPG). Using DNN's as non-linear function approximators, the algorithm can approximate an optimal policy. The model take as input a given state and outputs the best action to be taken, in that state, to its respective agent.

#### Implementation

##### Preparation

The goal is to train an agent able to get an average score of +0.5 over 100 consecutive episodes. The scores are distributed like follows: +0.1 each step an racket can hit the ball over an net and -0.1 if it hit the table or gets thrown out of its bounds. As the agents interact with the environment, the reward signal guides them towards maintaining the ball in play, characterizing a collaboration scenario.

At first, in the notebook file, the dependencies are installed, libraries are imported, the simulation environment is initialized.

The next step is to explore the State and Action Spaces. 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.

To learn how the Python API controls the agent and receives the feedbacks from the environment, a code cell is provided with a random action agent.

## **Learning Algorithm**

The maddpg algorithm is an approximate Actor-Critic Method, but also resembles the DQN approach of Reinforcement Learning. The agent is composed of two Neural Networks (NNs) the Actor and the Critic, both with target and local networks totaling 4 NNs, these are used to encode the policy function.

The learning pipeline takes first a state as input in the Actor network, outputting the best possible action in that state, this procedure makes possible for ddpq to tackle continuous action spaces, in contrast to the regular DQN approach. This action is used in the Critic network, alongside with the state, where it outputs an action value function ( $q$ ), this  $q$  is used as a baseline for updating both Actor and Critic networks, reducing the variance and instability of classic RL algorithms. The optimization is done with a gradient ascent between both Actor's and Critic's target and local networks parameters.

The behaviour of the agent can be explored in the maddpg\_agent.py file. Important libraries and components are imported and local parameters are initialized: BUFFER\_SIZE, defines the replay buffer size, shared by the agents, this is an object that contains tuples called experiences composed by state,actions,rewards,next states and dones, these are necessary informations for learning; BATCH\_SIZE, when the number of experiences in the replay buffer exceeds the batch size, the learning method is called; TAU, this hyperparameter controls the model soft updates, a method used for slowly changing the target networks parameters, improving stability; LR\_ACTOR and LR\_CRITIC, the optimizer learning rates, these control the gradient ascent step; WEIGHT\_DECAY, the l2 regularization parameter of the optimizer.

The main implementation begins on fourth step: additional libraries and components are imported, an agent is created and initialized with proper parameters: state\_size and action\_size. The maddpg function is created, taking as parameters the number of episodes (n\_episodes) and the maximum length of each episode (max\_t).

In each episode the environment is reseted and the agents receive initial states. While the number of timesteps is less than max\_t, the following procedures are done:

The agent use it's act method with the current state as input, the method takes the input and passes it through the actor network, returning an action for the state. A environment step is taken, using the previous obtained action, and it's returns: next state, rewards and dones (if the episode is terminated or not). These are stored in the env\_info,variable, that passes them individually for each of these information's new

variables. The agent uses its step method, the method first adds the experience tuple for the shared replay buffer and, depending on the size, calls the learn method. The rewards are added to the scores variable and the state receives the next state, to give continuation to the environment, if any of the components of the done variable indicates that the episode is over, the loop of `max_t` breaks, and a new episode is initialized.

If the average score of the last 100 episodes is bigger than 0.5, the network's weights are saved and the loop of `n_episodes` breaks and the `maddpg` function returns a list with each episode's score. This list is plotted with the episodes number, showing the Agent's learning during the algorithm's execution.

## Neural Network Architecture

**Actor** Composed of three hidden layers with 100, 75, 75 nodes respectively. The first one with batch normalization and each hidden layer is followed by a ReLU activation function. The output layer is followed by a Tanh function, making possible tackling continuous action spaces. The network takes as input the state and outputs the best calculated action for this.

**Critic** Composed of two hidden layers with 100+(action space size), 100 nodes respectively. The first one with batch normalization and each hidden layer is followed by a ReLU activation function. The network takes as input the state and outputs the action value function for the best action outputted by the Actor network.

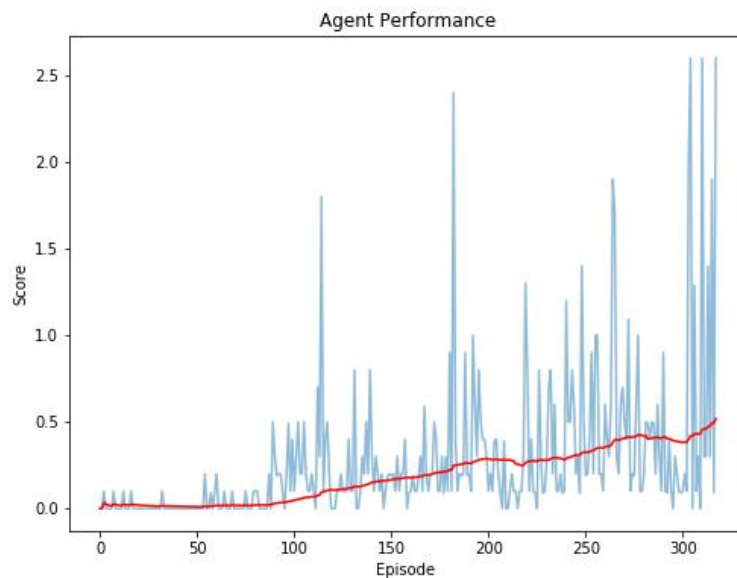
## Hyperparameters

Hyperparameters used in the experiment are given below,

Hyperparameter	Value
Buffer Size	1e-6
Batch Size	512
Gamma	0.99
TAU	5e-2
LR_Actor	5e-4
LR_Critic	5e-4
Weight Decay	0.0
Epsilon	1.0

## Plot of Rewards

Episode 100      Average Score: 0.05  
Episode 200      Average Score: 0.29  
Episode 300      Average Score: 0.38  
Episode 318      Average Score: 0.52  
Environment solved in 218 episodes!      Average Score: 0.52  
CPU times: user 13min 41s, sys: 42.1 s, total: 14min 23s  
Wall time: 15min 5s



## Ideas for future work

- It would be interesting to train the agent with PPO2 as the next step for modifications and improvement.