Report

Project 3: Collaboration and Competition

Benji Alwis

Introduction

Being able to scale up multi-agent reinforcement learning algorithms is an important step towards achieving human-robot interaction at scale. However traditional RL algorithms such as Q-learning and policy gradient methods are ill-suited for multi-agent learning. One key challenge is that when the policies of the agents are changed during training, the environment becomes non-stationary from the viewpoint of a single agent which makes the learning process unstable. This prevents the use of replay buffers that is important requirement in making Q-learning stable. On the other hand, policy gradient suffer from high variance in multi-agent environments.

The Multi-Agent Deep Deterministic Policy Gradient (MADDPG) method, adopts centralised learning and decentralised execution by using extra information on other agents to ease training but do not use them at execution time. This is illustrated in figure 1. As shown in green, during training, each agent has access to the policies of the other agent. However, during execution (shown in red), each agent acts on its own.

It is hard to use a similar approach in Q-learning since it generally does not allow different information at training and test phases. Simple extension of actor critic policy gradient networks used in MADDPG allows this. It achieves this by augmenting the critic with extra information about policies of other agents. After training, only the local actors are used in a decentralised manner.

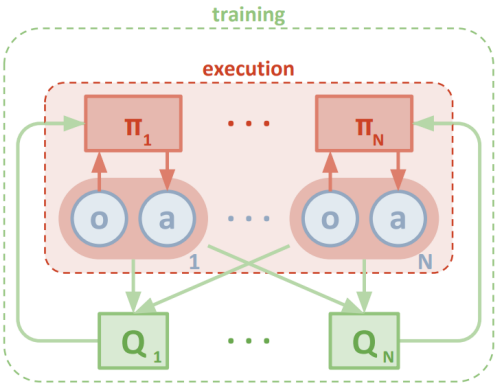


Figure 1 – Overview of the decentralized actor, centralized critic approach used in MADDPG (taken from the paper).

Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm below.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)\big|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

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

    end for
    Update target network parameters for each agent i:

$$\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$$

  end for
end for

Figure 2 – MADDPG algorithm (taken from the original paper).

Implementation

In this project, we utilise a training agent within a simulated 3D 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 code base used in this project was derived from the "DDPG pidedal" tutorial from the [Deep Reinforcement Learning Nanodegree] (https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893). It has been written in Python 3.6 with the use of PyTorch 0.4.0 framework.

The project consist of following files.

`model.py` : Implements the Actor and the Critic classes. Each of these classes implement a target and local network.

__init__() - this is the initialisation function where a fully connected neural network of one input layer, one hidden layer and one output layer is implemented.

forward(state) - this function maps states to actions.

‘ddpg_agent.py`: Basic implementation of an agent. It does not contain step or learn functions since they are implemented in maddpg class.

`maddpg_agent.py`: Implements the MADDPG alorithm. It instanciates a set of DDPG agents and provide the lerarning capability.

__init__() - - It instanciates a set of agents.

act() - it returns action to perform for each agents based on policy,

step() - it saves experience in replay memory, and takes a random sample from buffer to learn,

maddpg_learn() - this function slightly differs from the DDPG learn() method. The actors have only access to agent own information, whereas the critics have access to all agents information. This method updates the policy and value parameters using given batch of experience tuples based on the following logic.

*Q_targets = r + γ * critic_target(next_state, actor_target(next_state))*

where:

actor_target(states) -> action

critic_target(all_states, all_actions) -> Q-value

- `TennisProject.ipynb` : This Jupyter notebooks allows to instantiate and train both agent. More in details it allows to :

- Prepare the Unity environment and Import the necessary packages
- Check the Unity environment
- Define a helper function to instanciate and train a MADDPG agent
- Train an agent using MADDPG
- Plot the score results

solved_checkpoint_actor.pth and solved_checkpoint_critic.pth – these files contain the saved DNN’s weights.

Experiments

Experimental Goal

In this experiment, the task assigned for the two agents is to control rackets to keep a ball bouncing 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. To reach the goal, the agents should achieve an average score of +0.5 over 100 consecutive episodes. The score for each agent is calculated by accumulating the maximum reward received by an agent for each episode.

Environment

The environment provided by Udacity is based on [Unity ML-agents](https://github.com/Unity-Technologies/ml-agents). It is a two-player game in which agents control rackets to bounce ball over a net. We use the Tennis agent in this environment. Each agent (connected to a single brain called Tennis Brain) has its own, local observation space that consists of 8 variables. They correspond to the position and velocity of the ball and racket. In the Udacity provided environment, 3 observations are stacked (8 *3 = 24 variables). The two continuous actions correspond to movement towards the net, and the jump. Each agent independently takes an action and receives a reward of +0.1 if the ball was hit over net. It receives a reward of -0.1 if it lets ball hit their ground, or hit it out of bounds.

Hyper Parameters

SEED = 10 # Random seed

GAMMA = 0.995 # discount factor

TAU = 1e-3 # for soft update of target parameters

LR_ACTOR = 1e-4 # learning rate of the actor

LR_CRITIC = 1e-3 # learning rate of the critic

WEIGHT_DECAY = 0. # L2 weight decay

NB_EPISODES = 10000 # Max nb of episodes

NB_STEPS = 1000 # Max nb of steps per episodes

UPDATE_EVERY_NB_EPISODE = 4 # Nb of episodes between learning process

MULTIPLE_LEARN_PER_UPDATE = 3 # Nb of multiple learning process performed in a row

BUFFER_SIZE = int(1e5) # replay buffer size

BATCH_SIZE = 200 # minibatch size

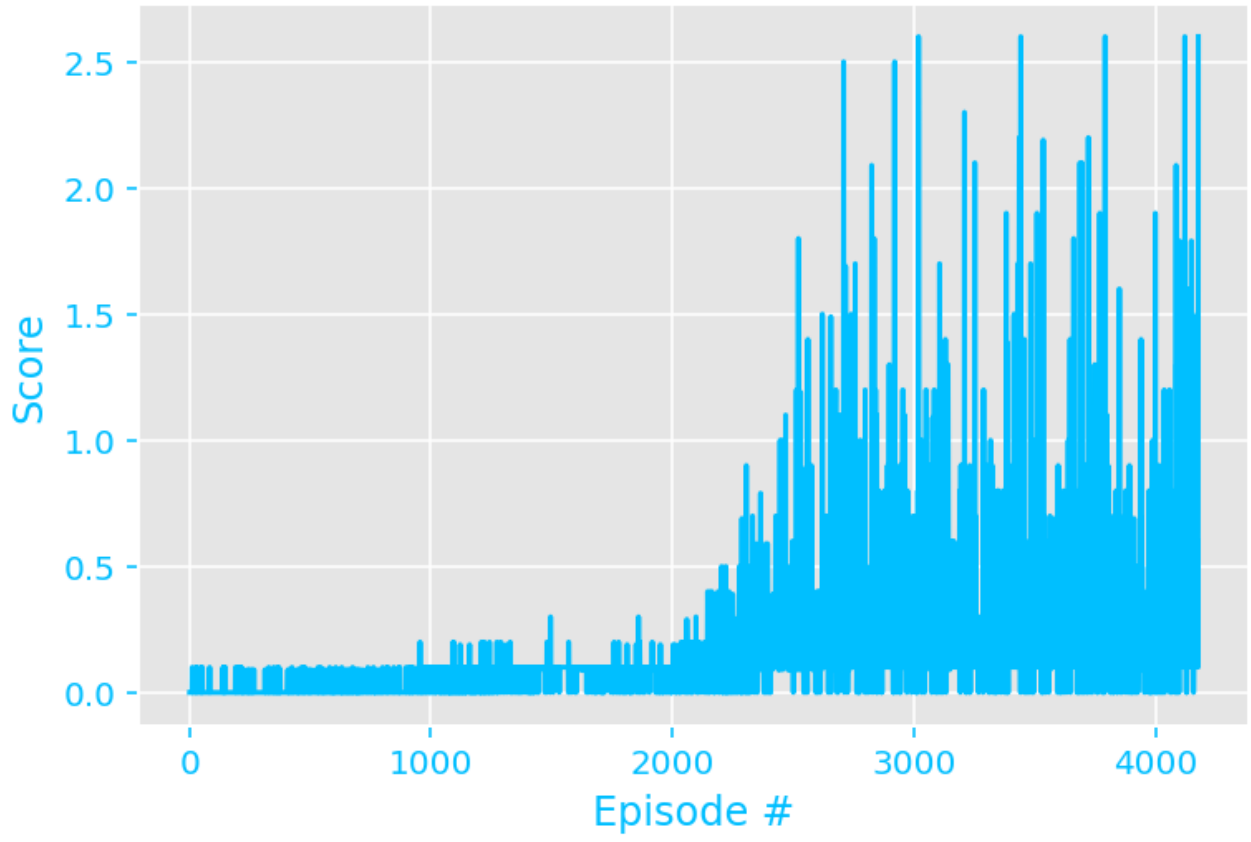
ACTOR_FC1_UNITS = 400 #256 # Number of units for the layer 1 in the actor model

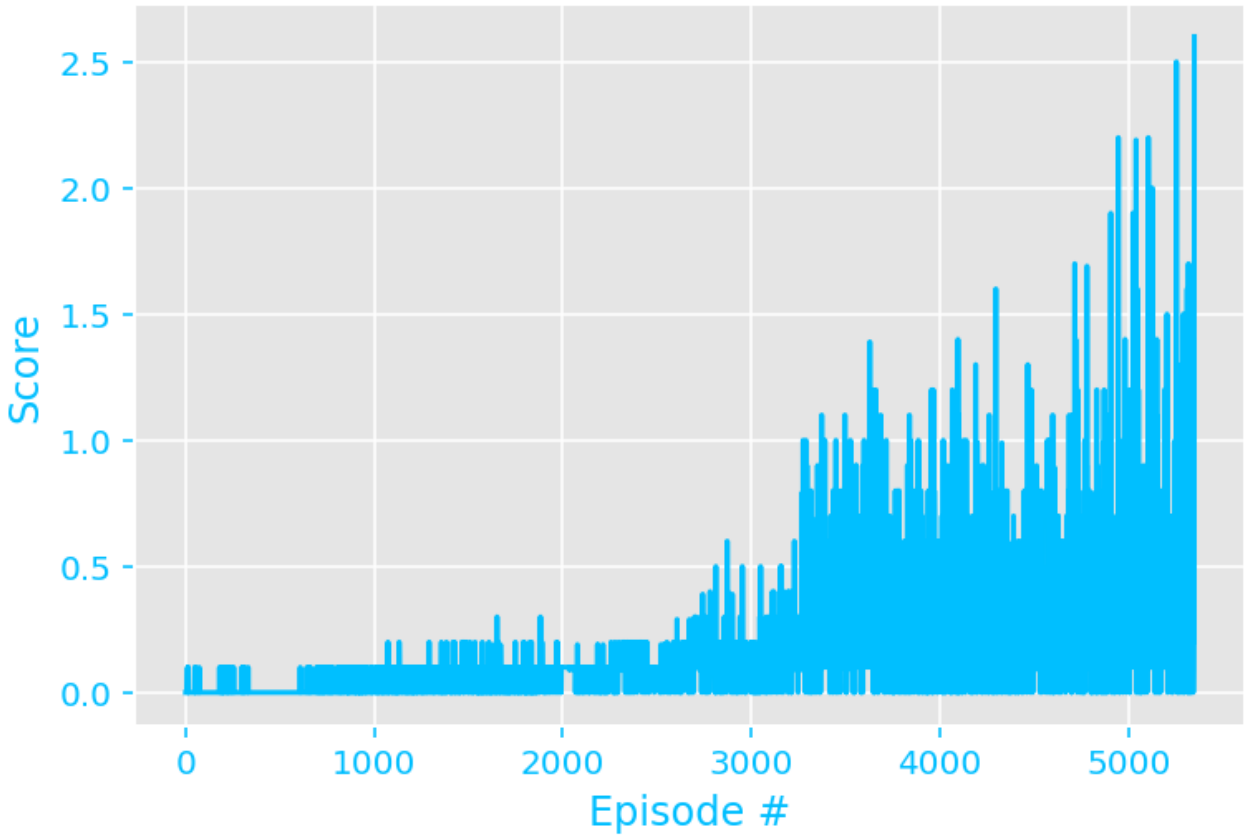
ACTOR_FC2_UNITS = 300 #128 # Number of units for the layer 2 in the actor model

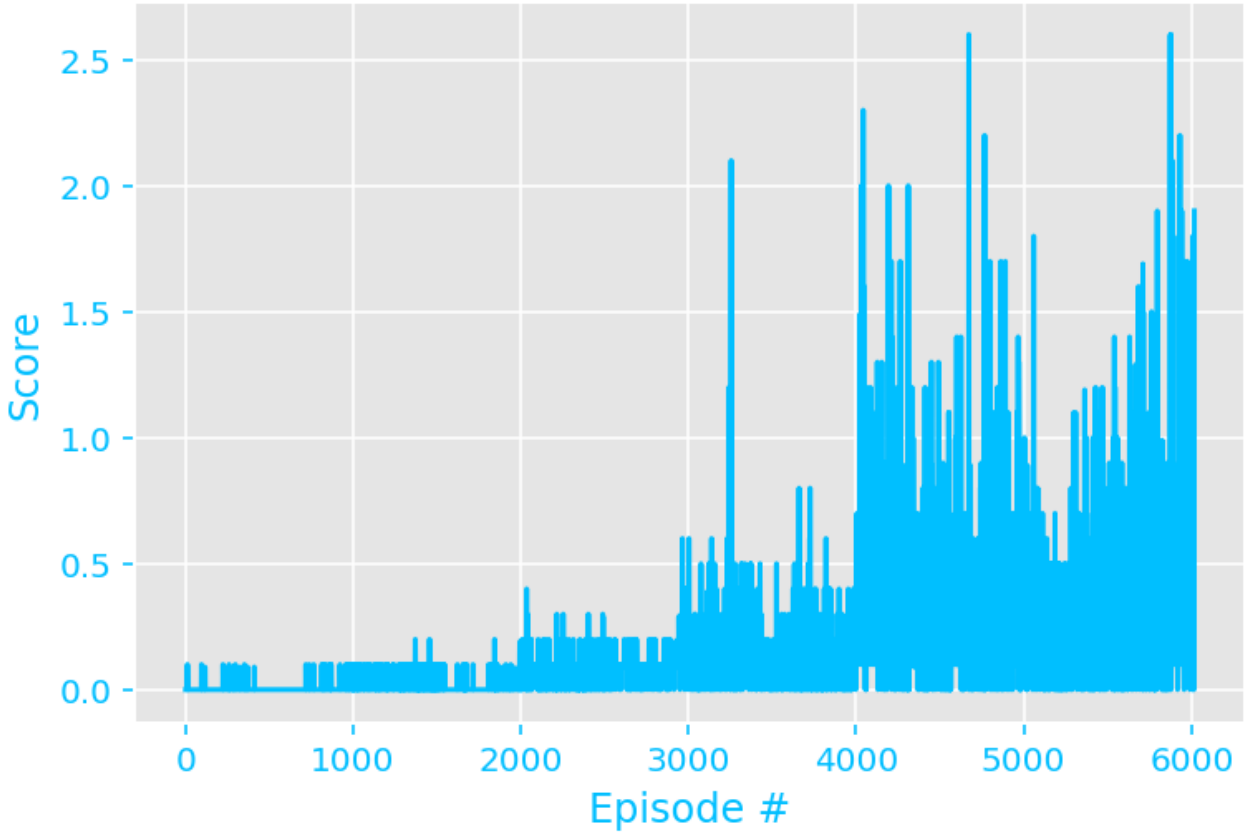
```
CRITIC_FCS1_UNITS = 400  #256      # Number of units for the layer 1 in the critic model
CRITIC_FC2_UNITS = 300  #128       # Number of units for the layer 2 in the critic model
NON_LIN = F.relu                   # Non linearity operator used in the model
CLIP_CRITIC_GRADIENT = False    # Clip gradient during Critic optimization
ADD_OU_NOISE = True                # Add Ornstein-Uhlenbeck noise
MU = 0.                            # Ornstein-Uhlenbeck noise parameter
THETA = 0.15                      # Ornstein-Uhlenbeck noise parameter
SIGMA = 0.2                        # Ornstein-Uhlenbeck noise parameter
NOISE = 1.0                       # Initial Noise Amplitude
NOISE_REDUCTION = 1.0          # Noise amplitude decay ratio
```

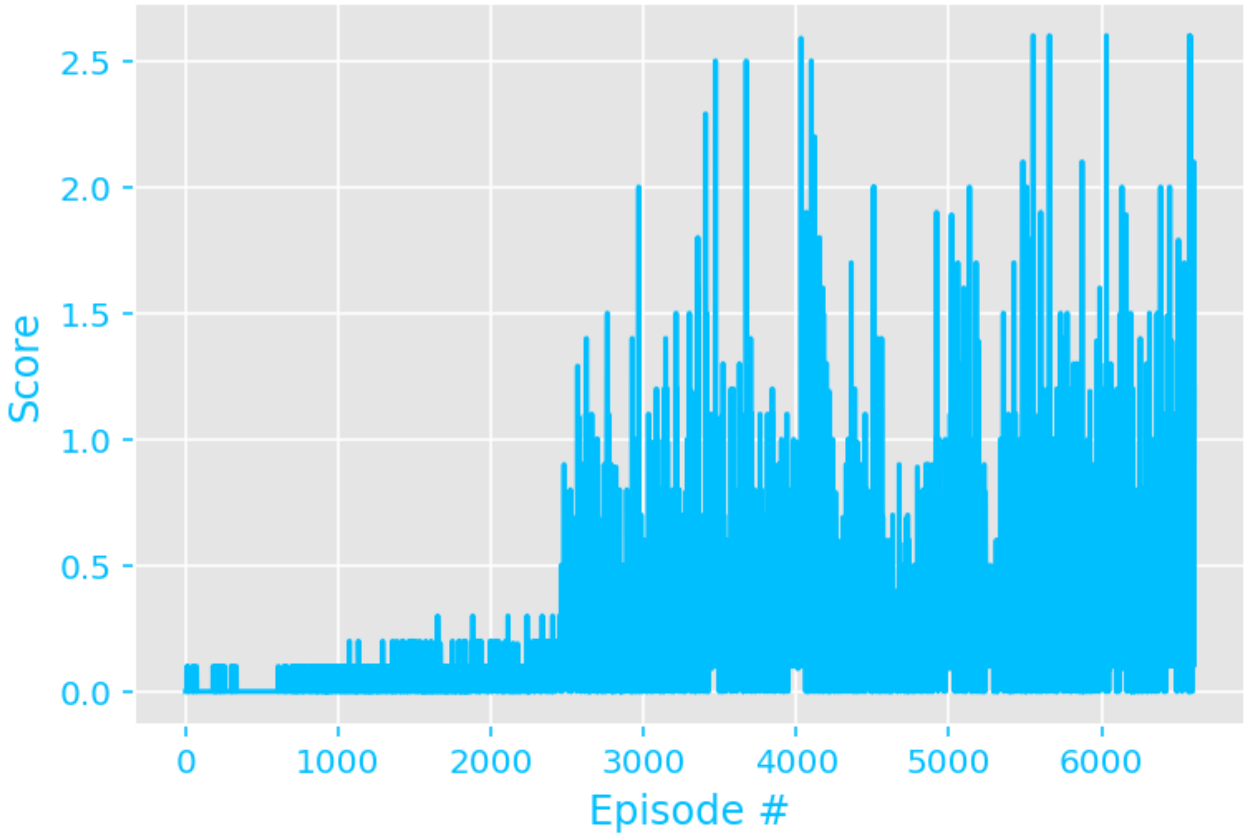
Results

I observed that re-production of the same results under the same hyper-parameters were very difficult. The following figures show different results generated by the same experimental setup in different runs. The number of episodes taken to achieve the goal varied between 4175 and 6607.




Score
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
5000
Episode #


Score
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
5000
6000
Episode #



Next Steps

I observed that the MADDPG algorithm can be over sensitive and brittle against hyper parameters. Another issue is that DDPG is that it tends to overestimate Q-values. Twin Delayed DDPG (TD3) aims to overcome this limitation by using Clipped Double-Q Learning, “Delayed” Policy Updates and Target Policy Smoothing. I aim to try this extension.