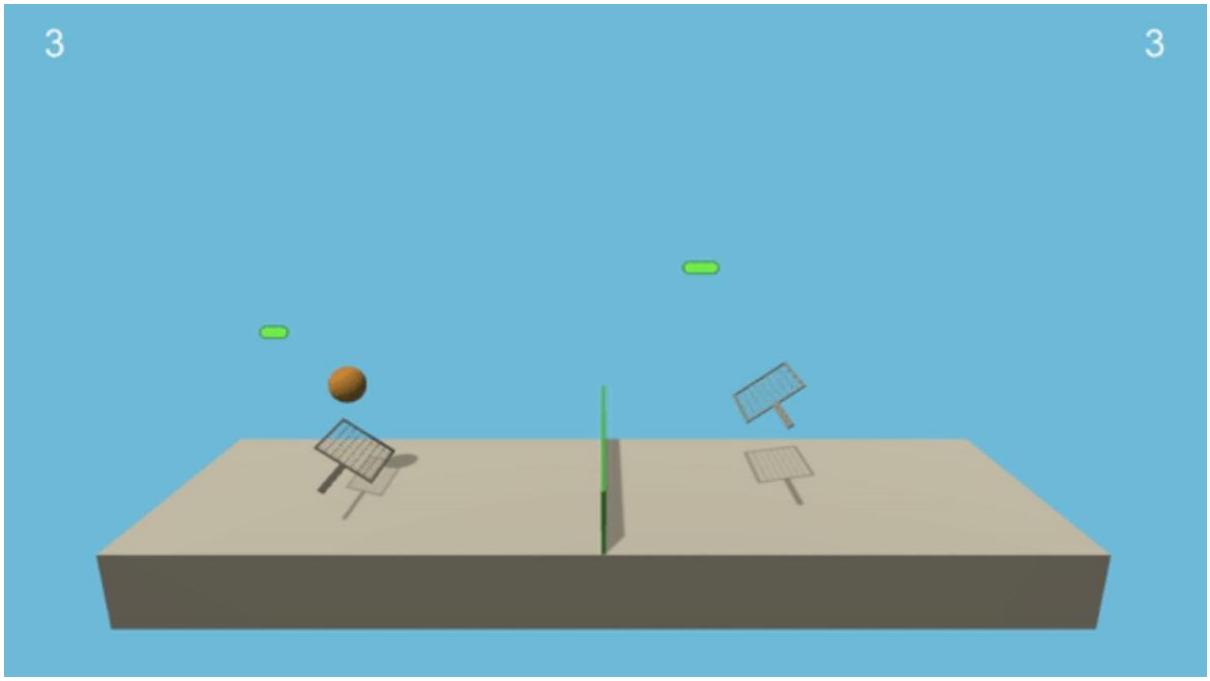
MADDPG – Collaboration & Competition

1. Introduction

In this project, we will work with the Unity ML-Agents 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.



Example of Unity ML-Agent Tennis environment

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

2. Implementation

2.1. Learning Algorithm

The learning algorithm in this project is based on Multi-Agent Deep Deterministic Policy Gradients (MADDPG) and it was implemented using PyTorch. MADDPG is the multi-agent counterpart of the Deep Deterministic Policy Gradients algorithm (DDPG), proposed by Lowe

et al. (2017). The image below illustrates the main components of DDPG. Just like DDPG, MADDPG uses two target networks to add stability to training by learning from estimated targets and updating target networks slowly where all agents share the experience replay memory.

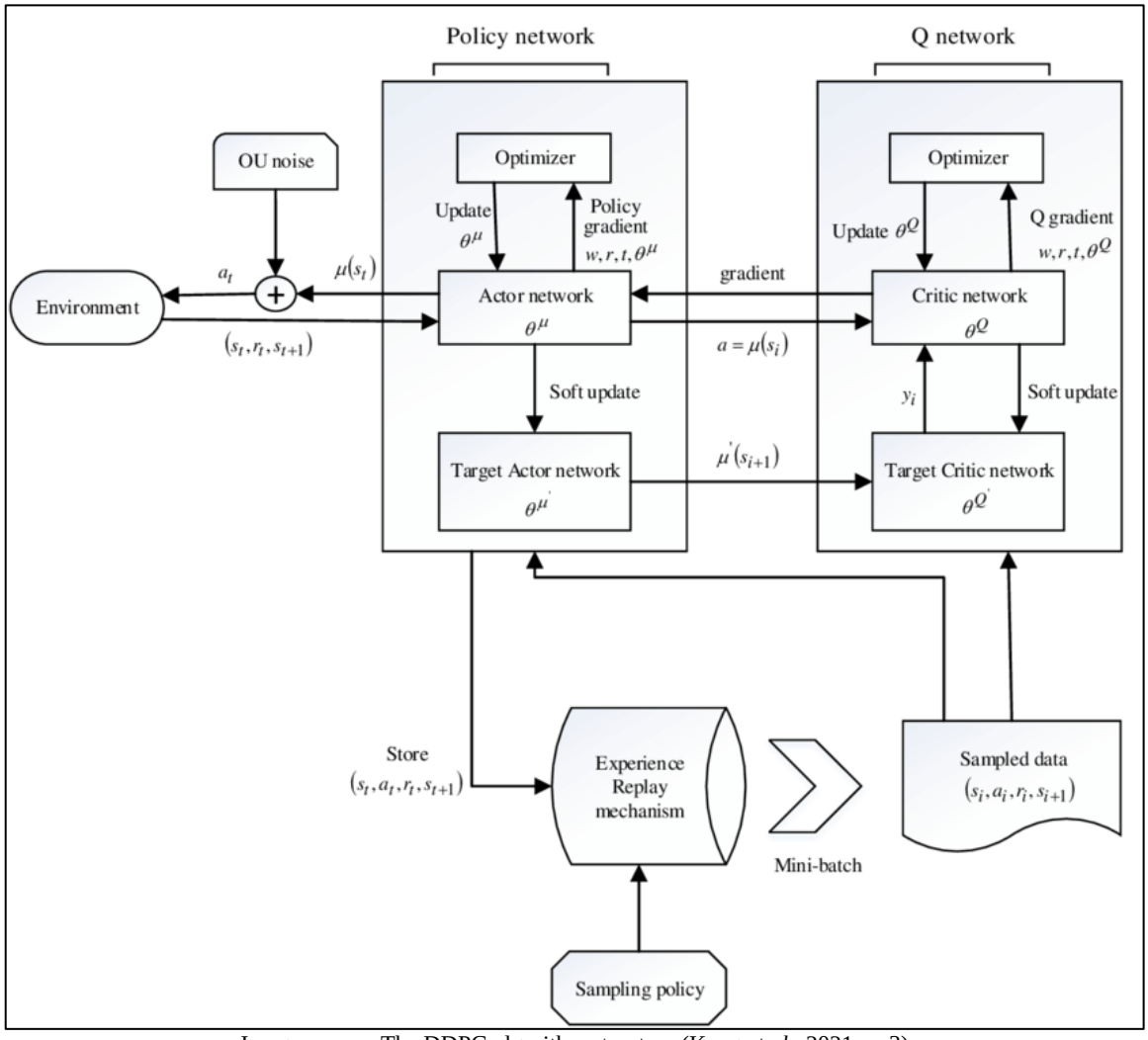


Image source: The DDPG algorithm structure (Kang *et al.*, 2021, p. 3)

2.2. MADDPG Code Break Down

Six helper classes are used to implement the MADDPG algorithm:

Class “Actor” implements the Neural Network which is composed by the following layers:

- 33 input neurons
- 400 Fist hidden layer neurons
- 300 Second hidden layer neurons
- 4 Output neurons

The “forward” method builds a network that maps state to action values using RELU and TAHN as the activation functions.

Class “Critic” implements the Neural Network which is composed by the following layers:

- 33 input neurons
- 400 Fist hidden layer neurons
- 300 Second hidden layer neurons
- 4 Output neurons

The “forward” method builds a network that maps state to action values using RELU as the activation function.

Class “ReplayBuffer” implements the experiences store and contains two main methods, “add” and “sample”, which add new experiences to the memory and randomly sample a batch of experiences respectively, and a helper “_len_” method that returns the current size of the internal memory to be used in the “Agent” class when determining if there are enough experiences collected compared to the set BATCH_SIZE.

Class “OUNoise” implements the Ornstein-Uhlenbeck noise process to construct an exploration policy as seen in Lillicrap *et al*. (2015) implementation. Methods of this class are “reset” that resets the internal state (= noise) to mean (mu), and “sample” where it updates the internal state and returns it as a noise sample.

Class “Agent” interacts and learns from the environment. The class initialises the Actor and Critic local and target networks and sets Adam as optimiser. Methods of this class are “Step” which saves experiences in the replay buffer and uses random sample from buffer to learn, “Act” which returns actions for given states as per current policy, “Reset” which resets and updates the internal state, and returns a noise sample, “Learn” which updates policy and value parameters using given batch of experience tuples, and “soft_update” which updates the model parameters.

Class “MADDPG” initiates *N* number of agents using class “Agent” and initialises the shared experiences replay using class “ReplayBuffer”. Methods of this class are “Step”, “Act”, “Reset”, and “Learn” which are interacting with the equivalent “Agent” class methods for each initiated agent.

The pseudocode below illustrates the MADDPG algorithm for *N* agents:

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

Pseudocode source: The MADDPG process (Lowe *et al.*, 2017, p. 13)

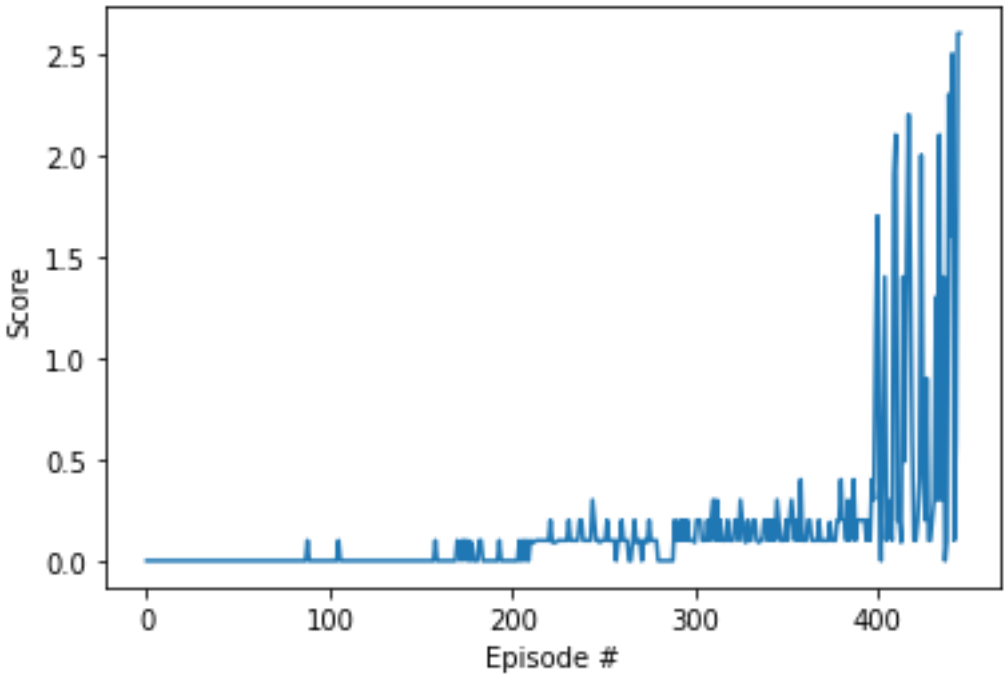
2.3. Hyperparameters

The table below shows the chosen hyperparameters, decided by a combination of trial and error and of continuous control baselines in bibliography.

Name	Value	Description
BUFFER_SIZE	1e5	Replay buffer size
BATCH_SIZE	256	Minibatch size
GAMMA	0.99	Discount factor
TAU	1e-2	Soft update of target parameters
LR	2e-4	Learning rate for actor and critic
WEIGHT_DECAY	0.0	L2 weight decay

2.3 Results

The plot below illustrates the agents' rewards per episode. The agents were able to meet that target of +0.5 average reward over 100 episodes after 446 episodes.



3 Future Work

Using a hyperparameters tuning tool might increase the agent's performance. For example, Tune, a Ray library, can run experiments in parallel and once it has converged in the best performing hyperparameters it can continue the training loop using them.

The agents start learning slowly and we can see a significant increase in the rewards after episode 400. Using one more hidden layer in both actor and critic networks in addition of implementing a minibatch normalisation in the hidden layers should probably stabilise and accelerate the training process.

Additional, even though each agent is achieving the goal of keeping the ball in play, they perform small rapid movements when they are waiting for the ball. It might be beneficial adding a small negative reward when an agent action isn't 0.0 (stays still). That might result in smoother movements from the agents.

4 References

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P. & Mordatch, I. (2017). "*Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*". Available at: https://doi.org/10.48550/arXiv.1706.02275.

Kang, C., Rong, C., Ren, W., Huo, F. & Liu, P. (2021). "*Deep Deterministic Policy Gradient Based on Double Network Prioritized Experience Replay*". IEEE Access. PP. 1-1. Available at: https://doi.org/10.1109/ACCESS.2021.3074535.