Report

November 21, 2019

1. Report

The aim of this report is to describe the learning algorithm (MADDPG) method used in the Collaboration and Competition Project, provided by Udacity, along with its chosen hyperparameters.

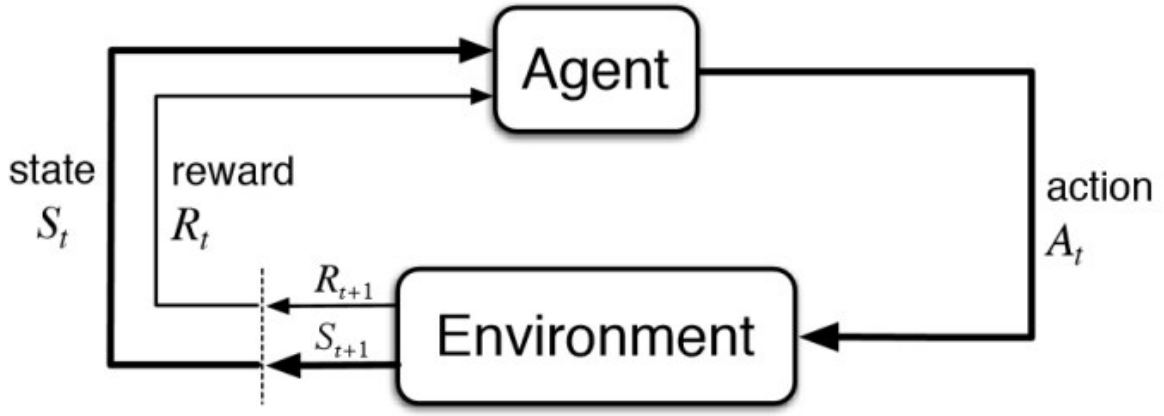
2. Background

2.1 – Markov Decision Process (MDP)

The Markov Property means that each *state* is dependent solely on its preceding state, the selected *action* taken from that state and the *reward* received immediately after that action was executed.
Mathematically, it means: $s' = s'(s,a,r)$, where s' is the future state, s is its preceding state and a and r are the action and reward. No prior knowledge of what happened before s is needed — the Markov Property assumes that s holds all the relevant information within it. A Markov Decision Process is decision process based on these assumptions.

2.2 – Deep Reinforcement Learning

Deep reinforcement learning (DRL) is a machine learning approach to artificial intelligence concerned with creating computer programs that can solve problems requiring intelligence. The peculiar property of DRL algorithms is the learning through trial and error from feedback, that is simultaneously sequential, evaluative and sampled by leveraging powerful non-linear function approximations. DRL algorithms learns what to do, how to map situations to actions, so as to maximize a numerical reward signal.



Reinforcement Learning Illustration (https://i.stack.imgur.com/eoeSq.png)

2.3 – Policy (**π**)

The policy, denoted as **π**, is a mapping from some state *s* to the probabilities of selecting each possible action given that state. An optimal policy **π*** is the solution to the Markov Decision Process and it is the best strategy to accomplish its goal.

2.4 – State action value function (Q function)

A state-action function is also called the Q function. It specifies how good it is for an agent to perform a particular action in a state with a policy π.

We can define Q function as follows:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right]$$

This specifies the expected return starting in state *s* with the action *a* according to policy π.

2.5 – Q Learning Algorithm

Q-Learning is an *off-policy Reinforcement Learning* algorithm. In its most simplified form, it uses a table, also known as Q-table, to store all Q-values of all possible *state-action* pairs. It updates its table using the *Bellman equation,* while action selection is usually made with an ***ε***-greedy policy. The optimal Q-value, denoted as Q^* can be expressed as:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \lambda \max_{a'} Q^*(s',a') | s,a\right]$$

Optimal Q-value (https://zhuanlan.zhihu.com/p/21378532?refer=intelligentunit)

	Action	
State	Left	Right
0	0	0
1	-100	65.61
2	59.049	72.9
3	65.61	81
4	72.9	90
5	81	100
6	0	0
7	100	81
8	90	72.9
9	81	0

Q-table Example

(https://itnext.io/reinforcement-learning-with-q-tables-5f11168862c8)

2.6 – Bellman Equation

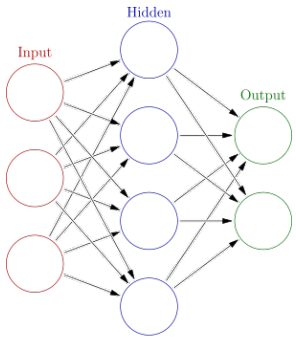
Defines the relationships between a given state (or state-action pair) to its successors. While many forms exist, the most common one usually encountered in *Reinforcement Learning* tasks is the Bellman equation for the optimal Q-value, which is given by:

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

Bellman Equation

2.7 – Artificial Neural Networks

Artificial neural networks are computing systems that are inspired by biological neural networks that constitute animal brains. Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules.



ANN architecture

3 – Multi-Agent Deep Deterministic Policy Gradients Algorithm (MADDPG)

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:

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

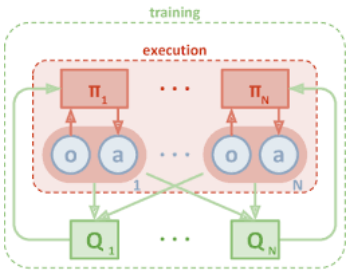
  end for
end for

The Multi-Agent Deep Deterministic Policy Gradient Algorithm leads to learned policies that only use local information at execution time (their own observations), does not assume a differentiable model of the environment dynamics or any particular structure on the communication method between agents and is applicable not only to cooperative interaction but to competitive or mixed interaction involving both physical and communicative behavior.

3.1 – Multi-Agent Actor Critic

We accomplish our goal by adopting the framework of centralized training with decentralized execution. Thus, we allow the policies to use extra information to ease training, so long as this information is not at test time. It is unnatural to do this with Q-learning, as the Q function generally cannot contain different information at training and test time. Thus, we propose a simple extension of actor-critic policy gradient methods where the critic is augmented with extra information about the policies of other agents.

In other words, during training, the Critics networks have access to the states and actions information of both agents, while the Actor networks have only access to the information corresponding to their local agent



4 – Environment

In this project, we'll train two agents control rackets to bounce a ball over a net. The goal of each agent is to keep the ball in play.

Details

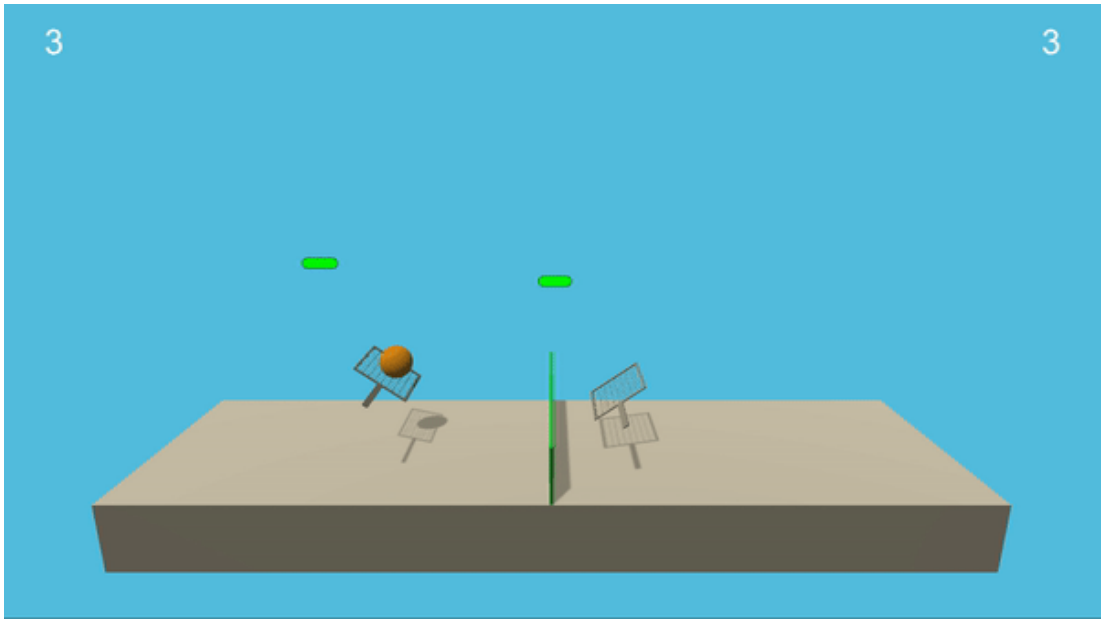
This environment was developed on Unity Real-Time Development Platform and it has some peculiarities:

- 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.

The observation space consists of **8** variables corresponding to 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.

The task is episodic and in order to solve the environment, your agents **must** get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents).

- After each episode, the rewards that each agent received are summed (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- This yields a single score for each episode



Udacity's Environment

5 – Solution

5.1– Hyperparameters

In order to gain optimal performance, hyperparameters tuning is very important. The following hyperparameters were set in the solution:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 250         # minibatch size
GAMMA = 0.99            # 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 2539
WEIGHT_DECAY = 0        # L2 weight decay
```

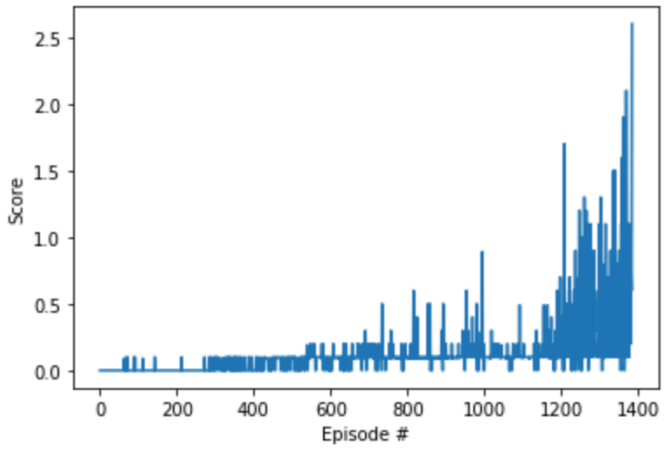
```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.size = size
        self.reset()
```

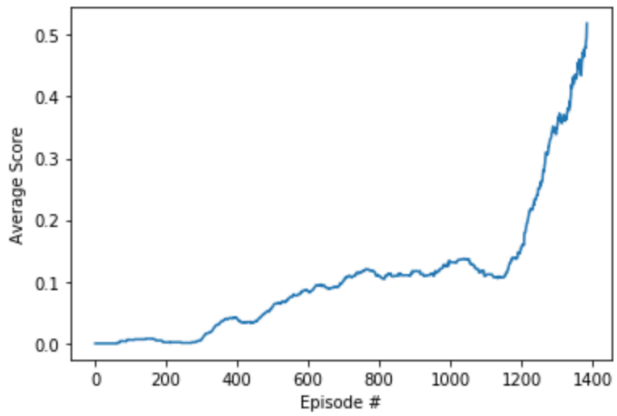
```
class Actor(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=200, fc2_units=150):
```

We used Adam for learning the neural network parameters with a learning rate of 1e-4 for the actor and 1e-3 for the critic. A discount factor was set to 0.99. For the soft target updates, we used TAU = 1e-3. The neural networks used the rectified non-linearity for all hidden layers. The final output layer of actor was a tanh layer to bound the actions. The low-dimensional networks had 2 hidden layers with 200 and 150 nodes respectively. Actions were not included until the second hidden layer of Q. The other layers were initialized from the uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer.

5.2 – Plot of Rewards

The environment considered as being solved if an agent would get an average of +0.5 reward over 100 consecutive episodes. The result is an average +0.518 reward in 1384 episodes, implemented by MADDPG. The agent achieved a good performance. The following image is the agent's learning scores:





6 – Ideas for Future Work

For further performance improvement, we could implement Twin Delayed DDPG (TD3) algorithm which demonstrated good performances for complex environments introducing three critical tricks:

Clipped Double-Q Learning. TD3 learns two Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Delayed" Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

Target Policy Smoothing. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

7 – References

Zychlinski, Shaked. "The Complete Reinforcement Learning Dictionary"
https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e

Google DeepMind, "Distributed Distributional Deterministic Policy Gradients"
https://arxiv.org/pdf/1804.08617.pdf

Yoon, Chris. "Deep Deterministic Policy Gradients Explained"
https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

Google DeepMind, "Continuous Control with Deep Reinforcement Learning"
https://arxiv.org/pdf/1509.02971.pdf

Google DeepMind, "Proximal Policy Optimization Algorithms"
https://arxiv.org/pdf/1707.06347.pdf

OpenAI, "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments"
https://arxiv.org/pdf/1706.02275.pdf