Udacity Deep Reinforcement Learning
Project Collaboration and Competition

Rohil Pal (rohilpal9763@gmail.com)

Abstract

In this project, we have to make two agents learn to play the game of Tennis. The environment is provided by **Unity-ML agents** [1]. The aim of this project is to demonstrate how two or more agents can collaborate with each other to learn to achieve the goal efficiently (in this case to learn to play tennis).

Environment

In this environment, two agents (players) 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.

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.

Solving the environment

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

- After each episode, we add up the rewards that each agent received (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.

The environment is considered solved, when the average (over 100 episodes) of those scores is at least $+0.5$.

Algorithm

Similar to Project 2 on Continuous Control, the environment can be solved with the *DDPG* algorithm after some obvious modifications. Although to achieve reproducible and better results (i.e solve the environment in lesser number of episodes), I think we need to employ the multi-agents adaptation of *DDPG* also known as ***MADDPG (Multi-Agent DDPG)***[2].

What is DDPG algorithm?

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

DDPG interleaves learning an approximator to $Q^*(s, a)$ **(Critic)** with learning an approximator to $a^*(s)$**(Actor)**, and it does so in way which is specifically adapted for environments with continuous action spaces.

Solution

The *DDPG* implementation for the 2nd with some modifications to hyperparameters worked very well in this project too.

Following are the main differences between the implementations of Project #2 and #3.

- The ***Ornstein-Uhlenbeck noise*** process in this project had to add noise to the actions for both agents. So, the dimensions of the Tensor storing the noise values is $(num\_agents, num\_actions)$.
- The score for each episode was the ***maximum*** of the scores of the two agents for that episode. So the corresponding code was added in the training loop.

Strategies employed to improve learning

As suggested in the *Benchmark implementation* ***(Attempt #4)*** of Project #2, the agents learnt from the experience tuples every 20 time-steps and at very update step, the agents learnt 10 times. Also, gradient clipping as suggested in ***Attempt #3*** helped improved the training. Also, to add a bit of exploration while choosing actions, as suggested in the *DDPG* paper, *Ornstein-Uhlenbeck process* was used to add noise to the chosen actions.

Actor and Critic Network Architecture

Both the *Actor* and the *Critic* networks have 2 hidden layers each. In both networks, the 1st layer has 400 neurons while the 2nd has 300 neurons. All hidden layers are followed by Batch Normalization layers and then *ReLU* activation. In Actor network, final layer contains number of neurons (= $\#actions$), followed by *tanh* activation. In the Critic network, final layer contains only one neuron. Since, the critic network estimates Q-values for all state-action pairs, we have to somehow insert the corresponding actions at some layer. We do this in the second hidden layer of the Critic network.

Weight Initialization

All layers but the final layer in both the networks are initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer. The final layer weights in both networks were initialized from a uniform distribution $[-3\times10^{-3}, 3\times10^{3}]$.

Hyper-parameters

- Learning rate (Actor) : $1e^{-3}$
- Learning rate (Critic) : $1e^{-3}$
- Batch size : 512
- Replay Buffer size : $1e^{5}$
- Discount factor (γ) : 0.99
- Soft Update parameter (τ) : 0.001

Results

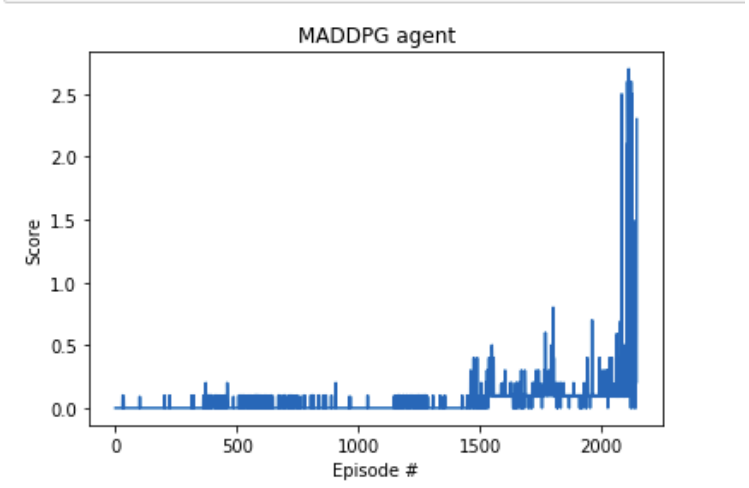

Figure 1: MADDPG agent (scores (maximum of the scores of 2 agents))

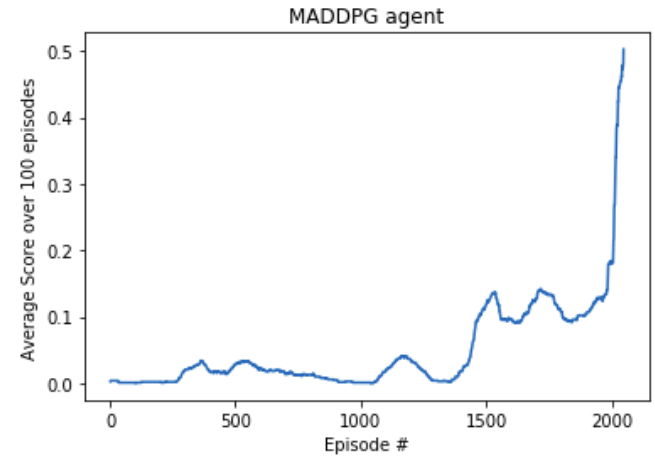

Figure 2: MADDPG agent (scores (averaged over all 100 episodes))

The algorithm was able to solve the environment in approximately 2050 episodes.

Note: I was able to achieve this after performing training for multiple times. First time, I was able to solve the environment in about 3200 episodes while in some runs, the score kept oscillating between 2 values. In the 9^{th} run, I was able to achieve the aforementioned claim. So, in my view, my implementation is quiet unstable and would be implementing better techniques as mentioned in the **Future Work** section.

Future work

- Implementing **multiple actors** and **shared critic** approach as mentioned in this MADDPG[2] paper because it involves information sharing between the agents which in turn has said to be efficient in the module on Multi-Agents Reinforcement learning.
- Also suggested in the same paper[2], is the use of **ensemble of policies**, which I believe is generally applicable to any multi-agent algorithm.

References

[1] `https://github.com/Unity-Technologies/ml-agents`

[2] `https://arxiv.org/abs/1706.02275`