Collaboration and Competition Project

Introduction:

The project is about solving multi-agents environment. In Figure 1, two agents play tennis table, and control their rackets to bounce a ball over the net. Every agent hits the ball over the net receives a reward of +0.1, and a reward -0.01 if the agent lets the ball hits the ground or hits the ball out of the bounds. The goal of every agent is to maintain the ball in play.

The observation space consists of 8 variables corresponding to position and velocity of the ball and the racket. Each agent receives its own local observation of the environment, and has two actions available corresponding to movement toward (or away from) the net and jumping. The actions are continuous and vary in the interval [-1, +1]

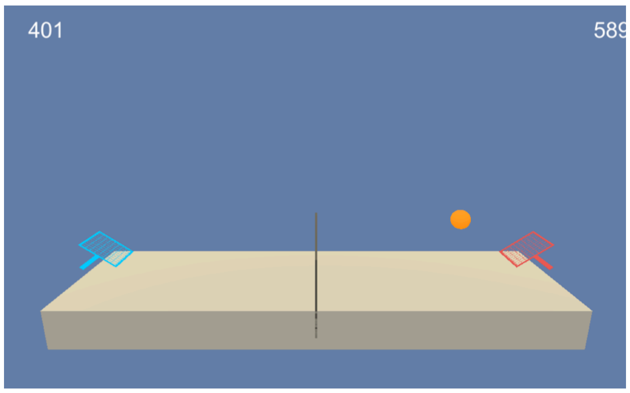

Figure 1. Multi-agents tennis environment.

In this project, we will consider the task episodic, and will adapt the DDPG algorithm to simultaneously train two agents to play tennis using the same actor-critic networks. The environment is considered solved if each agent achieves an average score of at least +0.5 over 100 consecutive episodes after taking the maximum score over both agents.

Learning algorithm:

The DDPG algorithm is a variation of the actor-critic. The learning algorithm assumes that both agents learn using the same actor-critic networks, and add their experiences in a shared replay buffer. That means both agents use the same actor network to select their actions.

Implementations:

The main functions used to train the agent are:

1. MADDPG.py: the implementation of multi-agent ddpg.
2. model.py: Actor and Critic neural networks

The module MADDPG.py includes Agents class that defines the functions required to interact and learn from the environment, a ReplayBuffer class to add and sample experiences for both agents from shared memory, and the OUNoise class for the exploration part of the DDPG. I adapted the OUNoise class to include a decayed sigma value to allow adaptive exploration. In my experiments, the noise parameters were set to: $\mu = 0$; $\theta = 0.15$; sigma= 0.5; sigma_decay=0.99; and sigma_min = 0.05. I also adapted the step and act functions in the Agents class dealing with a single agent to multi-agents. I found that trying with adaptive noise helped a lot the improving the performance of training.

The module model.py defines the neural network architectures, which are built in two separate class objects: Actor and Critic. The number of hidden layers and the number of units in each hidden layer are the same in both architectures. We have used architectures with two hidden layers of size: fc1_units = 256 and fc2_units = 128 neurons, respectively. The activation function of all hidden layers is the rectified non-linear function (ReLU). The activation of the last layer of the Actor is the tanh function, while the last layer of the Critic is linear. I adapted a batch normalization within the hidden layers to stabilize the training.

In my experiments, the following hyper-parameters were changed until getting the final result in this report: number of episodes = 3000, internal iterations = 500, seed = 100, memory size = 10^5, batch size = 64, discount factor (γ) is 0.99, the soft update target factor is 0.001, the learning rates were 1^{-4} and 1^{-3} for Actor and Critic, respectively, the L2 weight decay was left to zero. I added an update every hyper-parameter to control when to update the target network parameters.

Results:

Using the hyper-parameters values indicated in the implementation section, the agents were able to solve the environment after 1000 episodes. Figure 2. Shows the average scores over 100 consecutive episodes. The maximum average score accomplished after 100 consecutive episodes was +1.10, which is higher than the average score recorded by the benchmark implementation (+0.9148). Training the agents during the first 1500 episodes achieved better performance and stability than the benchmark solution in Figure 3.

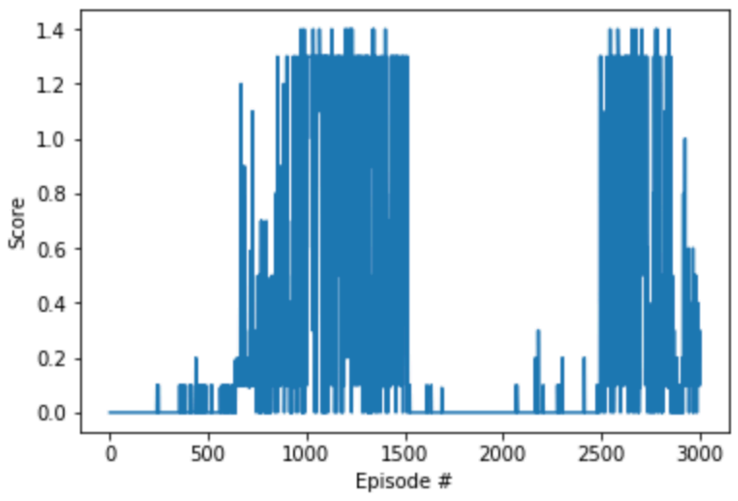

Figure 2. Score Plot of the Tennis environment.

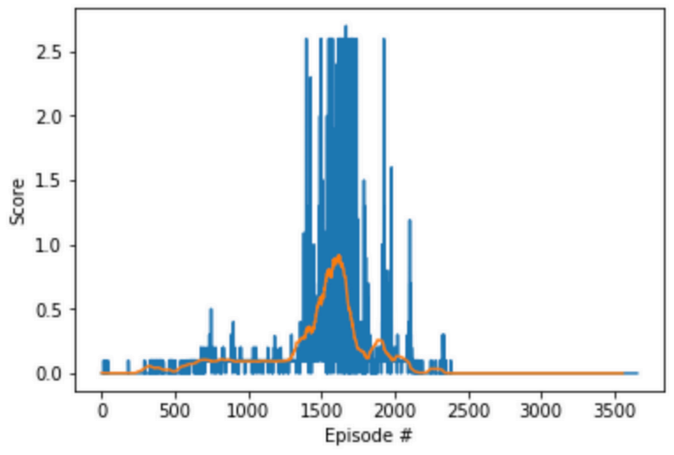

Figure 3. Benchmark solution.

Conclusion:

In this experiment, I started by the DDPG algorithm and trained the agents in a self-play model. There are still some interesting areas to explore for solving this environment. For example, implementing the MADDPG algorithm for two agents train in parallel using two separate actor-critic networks. Adapt the Replay buffer to the prioritized replay buffer to enhance performance and train faster. Implement the Alphazero algorithm to solve this enviornment.