Collaboration and Competition using Reinforcement Learning

Task Overview:

In this project, 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.

State and action space:

The observation space consists of 24 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. There are a total of 24 states and 2 actions for each agent.

Learning Algorithm:

We use the [Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments] algorithm to solve the environment. We have 2 *Actor* networks interacting with the environment to gather the experiences. There is a single *Critic* network that helps actors to make good decisions. More formally, this is the policy gradient that is used to update the actors:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim p^{\boldsymbol{\mu}}, a_i\sim \boldsymbol{\pi}_i}[\nabla_{\theta_i} \log \boldsymbol{\pi}_i(a_i|o_i) Q_i^{\boldsymbol{\pi}}(\mathbf{x}, a_1, ..., a_N)].$$

Here $Q^{\pi}_{i}(x, a_1, ..., a_N)$ is a centralized action-value (*Critic*) function that takes as input the actions of all agents, $a_1, \ldots, a_N$, in addition to some state information x, and outputs the Q-value for agent i.

The loss function of Critic network is as follows:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{\mathbf{x},a,r,\mathbf{x}'}[(Q_i^{\boldsymbol{\mu}}(\mathbf{x}, a_1, \ldots, a_N) - y)^2], \quad y = r_i + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}', a'_1, \ldots, a'_N)\big|_{a'_j=\boldsymbol{\mu}'_j(o_j)},$$

where μ' is the target parameters and μ is the local parameters.
x - current state
x'- next state
a - action
r - reward
γ - Discounting factor which can take a value between 0 and 1.

This algorithm is also called *multi-agent deep deterministic policy gradient* (MADDPG).

Network Structure:

It consists of 3 networks -> 2 Actor networks and 1 critic network:

Actor Network:

We've used a fully connected neural network with 3 layers that takes in 24 inputs (the current state of the agent) and generates 2 outputs (the action values). We have used the relu activation function after the first two layers and tanh activation after the 3rd layer.

Critic Network:

This is a slightly different network where it takes the states and actions as input and directly gives us the Q-value. It has 3 linear layers, where it takes in 48 states and 4 actions and gives a single Q-value.

Replay Buffer:

We have used a replay buffer that utilizes the concept of experience replay. We gather the experiences of the agents in a deque. The deque size is set to 10000. We randomly sample a mini-batch of size 1024 from the buffer pool and do a supervised training of the online network against the target network. Finally, a soft update is done by copying the parameters of the online network to the target network.

Ornstein-Uhlenbeck process:

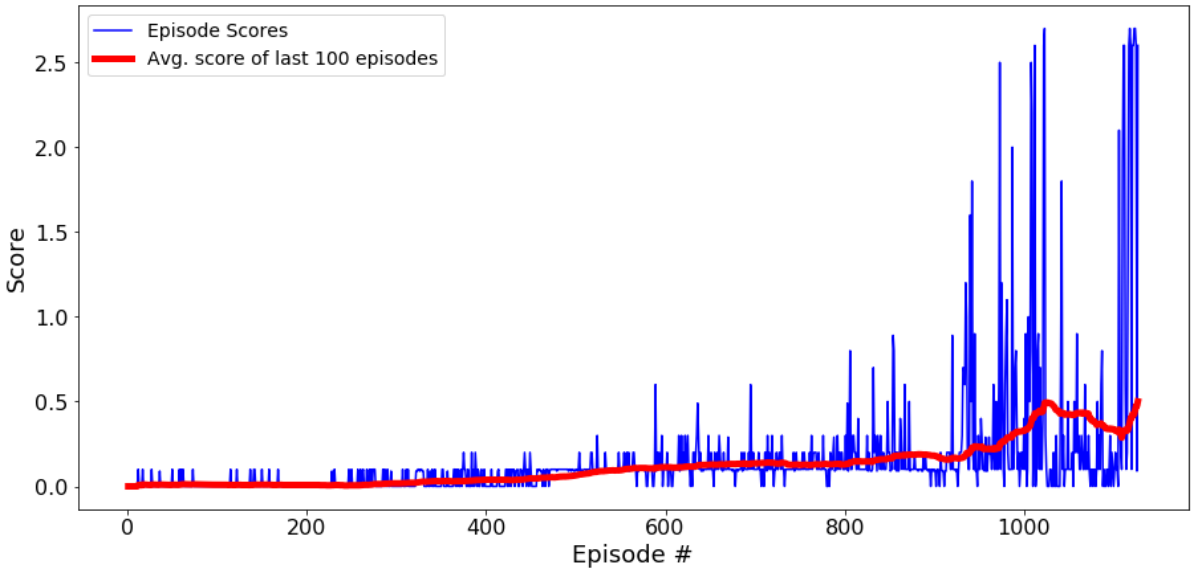
We introduce an additional noise to the actions before interacting with the environment, this encourages exploration of the agents.

Training details:

The learning rate of training the actor network was set to 1e-4 and the critic network was set to 1e-3. The discounting factor was set to 1. 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 average reward is calculated for the last 100 episodes of the training and monitored.

Results:

The agents received an average score of 0.501 in the 1127^{th} episode and the environment has been solved



Ideas for future work:

Prioritized Experience Replay could be implemented to further enhance the performance of the agents, so as to replay important transitions more frequently, and therefore learn more efficiently. We can assign priority weights for each transition step, which will be directly proportional to the TD error of the actor. Instead of randomly sampling from the buffer pool, we could sample based on the priority weights.