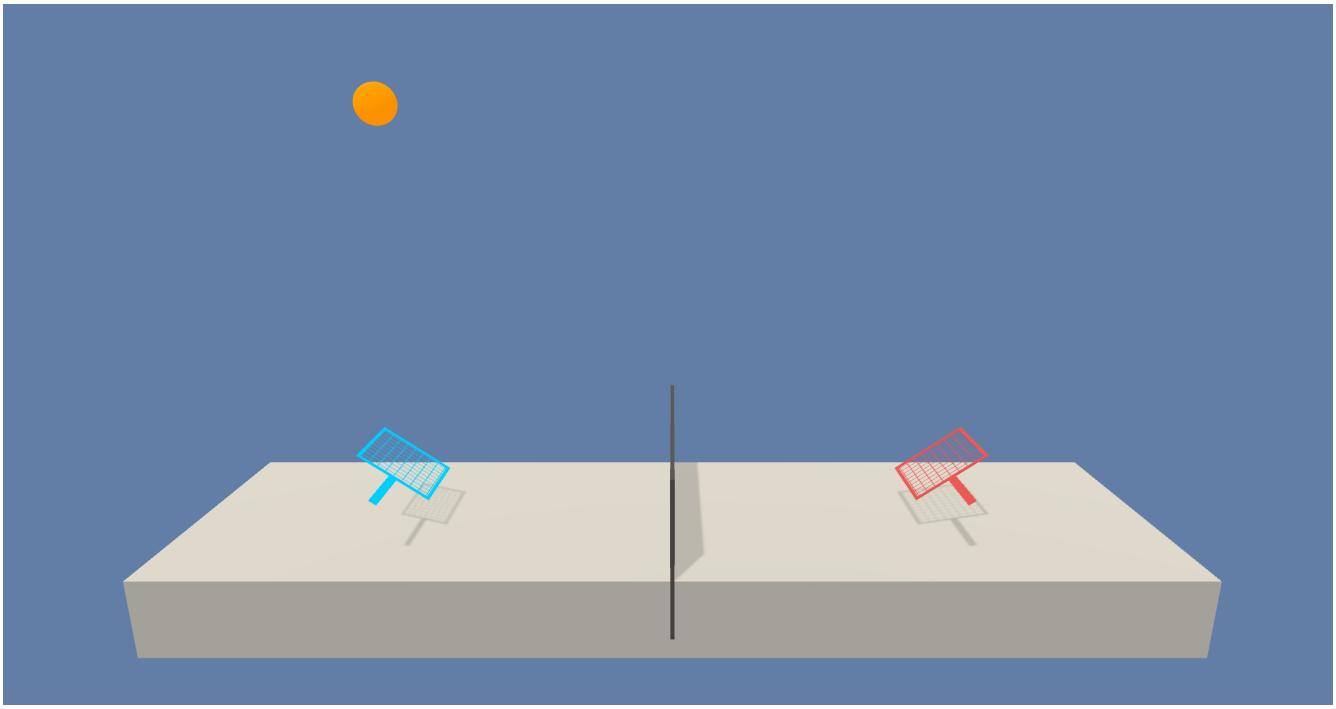
Collaboration & Competition

Goal

The main idea of the project is to solve a multi-agent collaborative environment using Deep Reinforcement Learning where the main idea for both of the agents is to **keep the ball in play**.



Environment

The virtual environment is provided by Udacity and is based on Unity ML agents.

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 **state space** consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation.

The **action space** contains 2 variables, corresponding to movement toward (or away from) the net, and jumping.

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).

Learning Algorithm

The algorithm used to solve this problem is **Multi Agent Deep Deterministic Policy Gradient** (**MADDPG**). As the name suggests it is basically a modified version of the DDPG algorithm.

It involves 2 networks. One actor and the other critic. Each of these 2 networks hold a local and a target copy of the network.

During training, the critic uses extra information like states observed and actions taken by all other agents.

For the actor, the information is agent specific. Each actor has access to only it's agent's observations and actions.

During execution time, only the actors are present, and hence it's own observations and actions are used.

Learning the critic for each agent allows us to use a different reward structure for each agent potentially. As a result, it can be used in a cooperative, competitive and mixed scenarios.

Neural Network Architecture

Actor

The actor network consists of 3 fully connected layers. The initial input is passed through a batch normalization procedure, which really improved the performance of the agent.

- Layer 1: Input size = 24 (state size), Output size = 128
- Layer 2: Input size = 128, Output size = 128
- Layer 3: Input size = 128, Output size = 2 (action size)
- [State] → Layer 1 → Leaky ReLu → Layer 2 → Leaky ReLu → Layer 3 → Tanh → [Action]

```
Actor network...
 Actor(
  (bn1): BatchNorm1d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=24, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=2, bias=True)
)
```

Critic

The critic network consists of 3 fully connected layers. The initial input is batch normalized.

The 2nd layer concatenates the actions size for both the agents, i.e. 2 * 2 = 4

- Layer 1: Input size = 24(state size), Output size = 128

- Layer 2: Input size = 132 (128 + (number of agents x action size)), Output size = 128
- Layer 3: Input size = 128, Output size = 1 (action value)
- [State] → Layer 1 → Leaky ReLu → Layer 2 → Leaky ReLu → Layer 3 → [Action Value]

```
Critic network...
 Critic(
  (bn1): BatchNorm1d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fcs1): Linear(in_features=24, out_features=128, bias=True)
  (fc2): Linear(in_features=132, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```

Note:

- There is no activation function leading to the final output layer for Critic. This is because the action values can be negative as well, but having something like a ReLu activation will only output positive values.
- The output is given by a Tanh function for the Actor network. This is because the action vector has continuous values between -1 and 1.

Hyperparameters

The following hyper-parameters has been used.

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128  # minibatch size
GAMMA = 0.99  # discount factor
TAU = 1e-3  # for soft update of target parameters
LR_ACTOR = 1e-3  # learning rate of the actor
LR_CRITIC = 1e-3  # learning rate of the critic
WEIGHT_DECAY = 0  # L2 weight decay
```

In addition to these, there is a **noise decay factor** of **0.999**. A linear decay of the **noise weight** helped speed up the learning process.

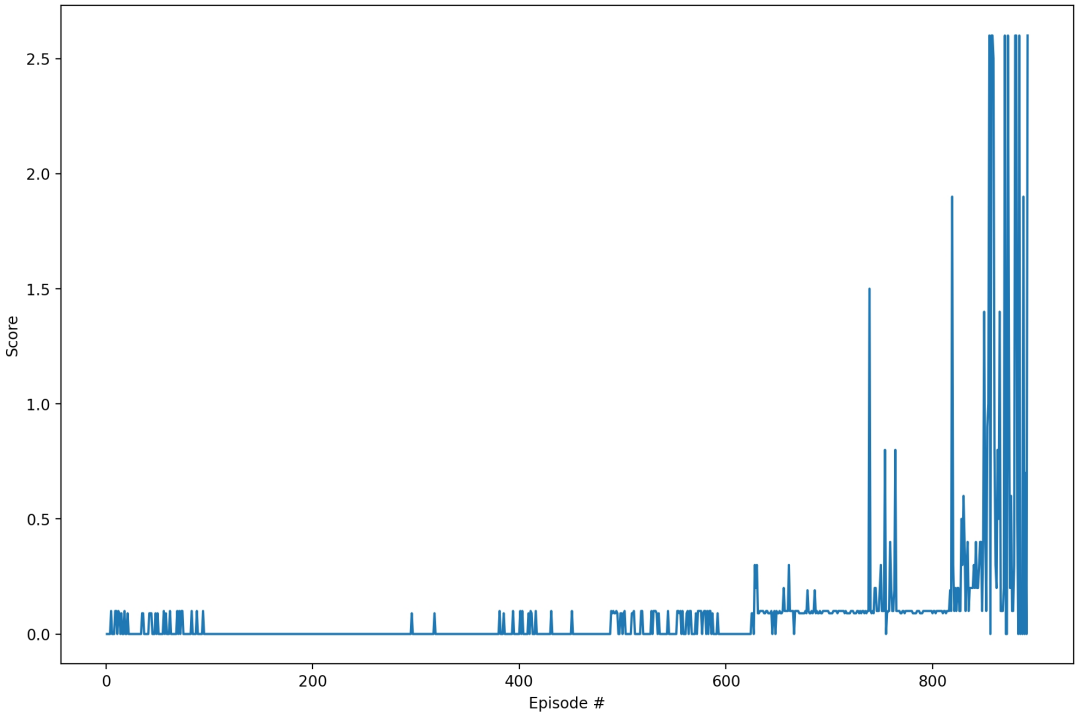
Plot of Rewards

It took 892 episodes to successfully train the agent.

```
Starting to train the agent....

Episode 892     Episode Score: 2.600    Noise Factor: 0.45      Max Score: 2.600        Average Score: 0.52
Environment solved in 892 Episodes      Average Score: 0.52
```

The plot is shown below.



Ideas for future work

For increased stability of the model and reduced training time, a few enhancements could have been done:

- Add dropout layers, which theoretically makes the network more balanced
- Reducing the frequency of the local network to target layer update could make the rewards plot smoother
- A Monte-Carlo Tree Search (MCTS) based technique, as seen in the AlphaZero algorithm could potentially give better results.
- Prioritized Experience Replays will definitely help improve the performance, by learning more from some experiences than others.