# Multi-Agent Collaboration and Competition

## Project Background:

Artificial Intelligence is now the key component of all software we build. And it is replacing all the human task of IT infrastructure day by day. Software like Dynatrace, Riverbed Aternity and so on are using Artificial Intelligence and Data Science to perform human task. Specially in Artificial Intelligence, Multi Agent Reinforcement Learning is the key of the success. Towards this goal of having effective AI-agent and human interaction, reinforcement learning algorithms that train agents in both collaborative and competitive are showing great promise. The environment has 2 agents that control rackets to bounce balls over the net. The reward structure for states is as follows:

| States | Reward |
|---|---|
| Hit Over the Net | +0.1 |
| Ball Hits the Ground | -0.01 |

Thus, the reward is constructed to teach the agents to keep the ball in play. The task is episodic, and the environment is solved when the agents get an average score of +0.50

## Goal:

My goal is to train two reinforcement learning agent to play tennis and they will keep the ball in play for as long as possible.

## Implement a learning algorithm:

I used policy-based algorithm since we have a continuous action space and value-based methods won't scale. Here are the advantages of policy-based methods: Here are some advantages of policy-based methods:

- **Continuous action spaces** — Policy-based methods are well-suited for continuous action spaces.
- **Stochastic policies** — Policy-based methods can learn true stochastic policies.
- **Simplicity** — Policy-based methods directly learn the optimal policy, no need to maintain a separate value function estimate.

This project used a Multi Agent Deep Deterministic Policy Gradient (MADDPG) method detailed in this paper (https://arxiv.org/pdf/1509.02971.pdf)

## Hyperparameters:

The following hyperparameters were used:

- 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

## Model Architecture:

The algorithm uses two deep neural networks (actor-critic) with the following structure:

Actor

- 2 fully connected layers with 200 and 150 units each

```
class Actor(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=200, fc2_units=150):
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))
```

Critic

- 2 fully connected layers with 200 and 150 units each

```
class Critic(nn.Module):

    def __init__(self, state_size, action_size, seed, fcs1_units=200, fc2_units=150):
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear((state_size+action_size) * num_agents, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        xs = torch.cat((state, action), dim=1)
        x = F.relu(self.fcs1(xs))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Replay Buffer

- A fixed size buffer of size 1e5

## Plot of Rewards:

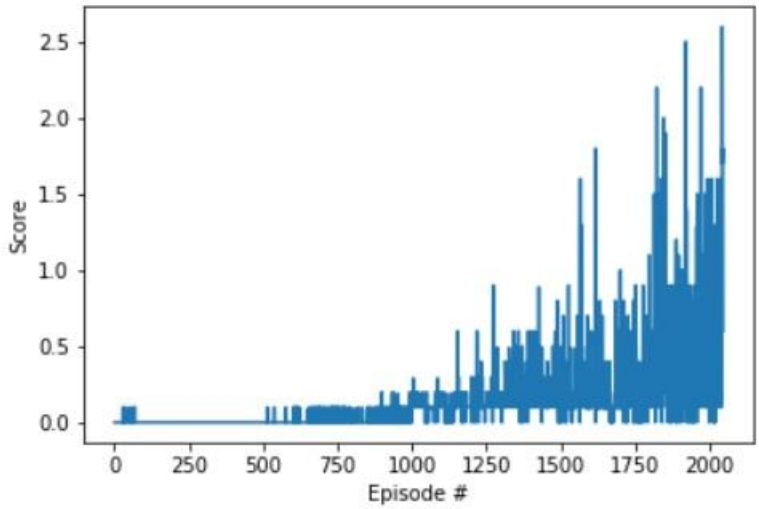
The Agent was initially very slow but picked up rewards fast after around 1000 episodes.

```
scores , average_scores_list = train_mddpg()
```

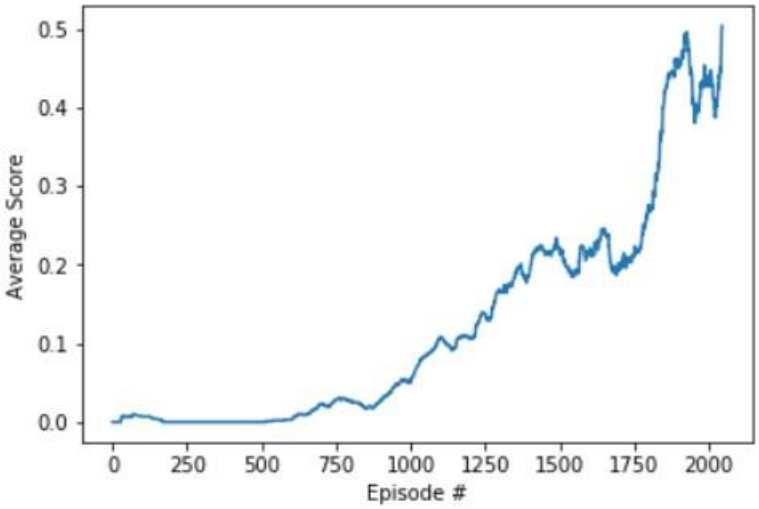
```
Episode 100     Average score: 0.007
Episode 200     Average score: 0.000
Episode 300     Average score: 0.000
Episode 400     Average score: 0.000
Episode 500     Average score: 0.000
Episode 600     Average score: 0.003
Episode 700     Average score: 0.023
Episode 800     Average score: 0.026
Episode 900     Average score: 0.029
Episode 1000    Average score: 0.052
Episode 1100    Average score: 0.107
Episode 1200    Average score: 0.109
Episode 1300    Average score: 0.167
Episode 1400    Average score: 0.194
Episode 1500    Average score: 0.213
Episode 1600    Average score: 0.219
Episode 1700    Average score: 0.199
Episode 1800    Average score: 0.267
Episode 1900    Average score: 0.456
Episode 2000    Average score: 0.429
Solved in episode: 2044         Average score: 0.503
```

The Environment was solved in 2044 episodes!

The graph below shows the final training results.




The plot of the average scores as training progresses is shown below:




- The agent1_checkpoint_actor.pth file represents the first agent actor
- The agent1_checkpoint_critic.pth file represents the first agent critic
- The agent2_checkpoint_actor.pth file represents the second agent actor
- The agent2_checkpoint_critic.pth file represents the second agent critic

## Improvements:

**1. Batch Normalization** — Batch normalization was not used on this project. The Google Deep Mind paper (https://arxiv.org/pdf/1509.02971.pdf) talks about the benefits of using this approach.

**2. Proximal Policy Optimization** — This technique modifies the parameters of the network in such a way that the new set of parameters is looked for in the immediate neighbourhood of the parameters in the previous iteration of the training. This is shown also to be an efficient way of training the network, so the search space is more optimal. (https://arxiv.org/abs/1707.06347)