# Report

October 31, 2019

## 1 Goal

The goal of this project is to train two RL players to play tennis against each other. As in real tennis, the goal of each player is to keep the ball in play. And, when you have two equally matched opponents, you tend to see fairly long exchanges where the players hit the ball back and forth over the net.

## 2 The Environment

We'll work with the Tennis environment on the Unity ML-Agents.

In this environments, 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.

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 moves toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, your 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.

## 3 Learning Algorithm

My learning algorithm is a Multi-Agent Deep Deterministic Policy Gradient.

DDPG is an actor-critic algorithm and primarily uses two neural networks, one for the actor and other for the critic.These networks calculate action vectors for the current state and and generate a temporal-difference error signal each time step.

DDPG uses a stochastic behavioral policy for good exploration and a deterministic target policy for estimating.

The current state is the input of the actuator network and the output is a single value representing the action. The deterministic policy gradient theorem provides the update rule for the weights of the actor network.

The critic's output is simply the estimated Q-value of the current state and the action given by the actor. The critic network is updated from the gradients obtained from the TD error signal.

My implementation is based on the project 2 of the nanodegree i.e Continuous Control. The difference between both of these is that in the earlier project, I had trained one agent in a continuous environment. In this one, I've trained two agents who compete against each other.

Here is the summary of how the Actor works in code:

```
class Actor(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        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.bn1 = nn.BatchNorm1d(fc1_units)
        self.reset_parameters()

    ...

    def forward(self, state):
        if state.dim() == 1:
            state = torch.unsqueeze(state,0)
        x = F.relu(self.fc1(state))
        x = self.bn1(x)
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))
```

Following code shows the summary of the Critic:

```
class Critic(nn.Module):
    def __init__(self, state_size, action_size, seed, fcs1_units=128, fc2_units=128):
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.bn1 = nn.BatchNorm1d(fcs1_units)
        self.reset_parameters()

    ...

    def forward(self, state, action):
        if state.dim() == 1:
            state = torch.unsqueeze(state,0)
        xs = F.relu(self.fcs1(state))
        xs = self.bn1(xs)
        x = torch.cat((xs, action), dim=1)
```

```
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

I tried adding `leaky_relu` in the model instead of `relu` but the model became very slow to converge.

The set of hyperparameters chosen for the agent along with their description and the code of working of the agent is found in `ddpg_agent.py`. The hyperparamters were chosen after seeing some implementations of other students and these worked better for me.

```
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     = 2e-4             # learning rate of the actor
LR_CRITIC    = 2e-4             # learning rate of the critic
WEIGHT_DECAY = 0                # L2 weight decay
```

To help the agent explore new actions, I added something called `Ornstein-Uhlenbeck` noise to the actions which the actor returned, which is the `OUNoise` class in the `ddpg_agent` file.

Also, something called `Replay Buffer` was used which stores experience tuples in the memory so that agent can randomly sample previous experiences to learn from the memory.

I've also added gradient clipping to the critic so that the weights don't explode

```
critic_loss = F.mse_loss(Q_expected, Q_targets)
# Minimize the loss
self.critic_optimizer.zero_grad()
critic_loss.backward()
torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
self.critic_optimizer.step()
```

### 3.0.1 Some things which I tried but failed

- I tried adding `EPSILON` so as to decay the noise with time. But my model failed to converge.
- I tried adding multiple passes of learning per 5 episodes and no learning between those 5 episodes. But still model was converging slower than expected

## 4 Plot of rewards

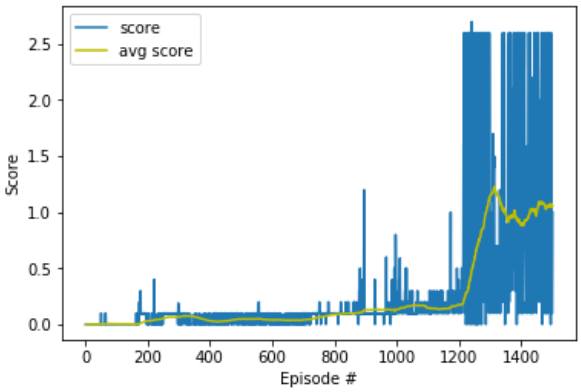
The agent achieved an average score of 0.517700007726 over 100 episodes in episode number 1243. Here is the plot of rewards achieved along with the average score over 100 episodes

Here is a screenshot of the training process. As you can see the agent achieved an average score of 1.15 in the episode range (1200-1300)

## 5 Ideas for future Work

- Change the network implementation and choose different hyperparameters
- Trying algorithms like D4PG, PPO, A3C
- Currently our replay buffer is dumb. We can use prioritised experience buffer.

```
:
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(1, len(scores)+1), scores, label='score')
plt.plot(np.arange(1, len(scores)+1), avg_scores, c='y', label='avg score')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left')
plt.show()
```




Reward Plot

```
scores, avg_scores = ddpg_tennis()
```

```
Episode 100     Score 0.00      Average Score 0.00
Episode 200     Score 0.00      Average Score 0.03
Episode 300     Score 0.09      Average Score 0.07
Episode 400     Score 0.00      Average Score 0.03
Episode 500     Score 0.10      Average Score 0.05
Episode 600     Score 0.00      Average Score 0.04
Episode 700     Score 0.00      Average Score 0.04
Episode 800     Score 0.10      Average Score 0.09
Episode 900     Score 0.09      Average Score 0.13
Episode 1000    Score 0.10      Average Score 0.13
Episode 1100    Score 0.19      Average Score 0.15
Episode 1200    Score 0.10      Average Score 0.17
Score of 0.517700007726 achieved in 1243 episodes
Episode 1300    Score 0.20      Average Score 1.15
Episode 1400    Score 0.10      Average Score 0.88
Episode 1500    Score 1.00      Average Score 1.06
```

Training Plot

- Different replay buffer for actor/critic
- Try adding dropouts in networks

In [ ]: