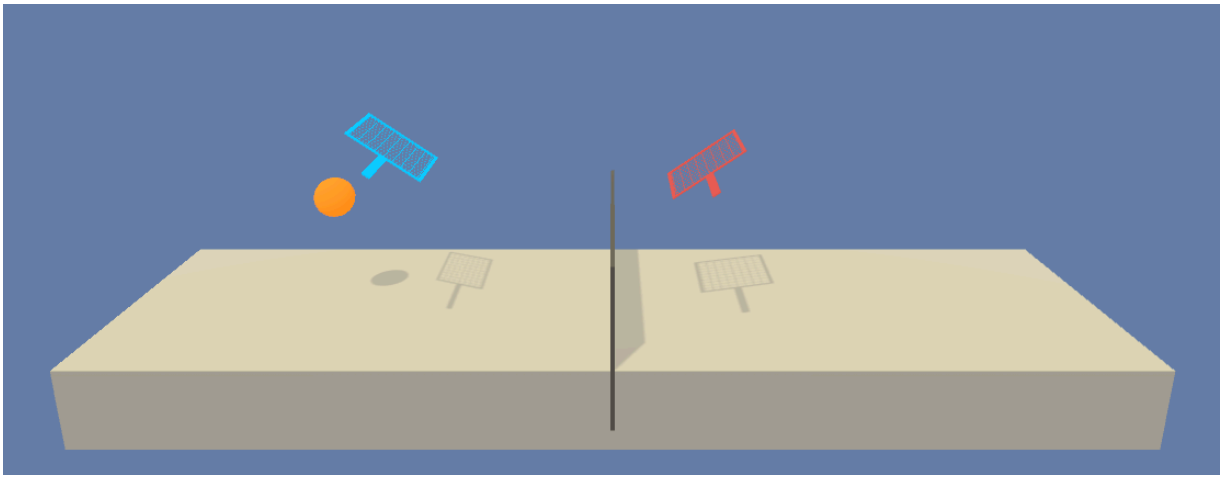
Name: Mark Abrenio

Project: Udacity Project 3 (Tennis Game)

For this project we were required to utilize an algorithm to solve a simple video game that has two tennis rackets playing tennis. The goal is to get the two tennis rackets to successfully hit the ball back and forth between them without it hitting the net etc.



To solve this game I ended up using a slightly modified version of DDPG called MADDPG. I first thought I could simply solve the game using the DDPG algorithm from my project 2 code. However, I quickly realized this was not going to work. Since a lot of these RL algorithms work by bootstrapping off actions that give rewards, DDPG would not solve this game for me. The reason is, I was using the same actor and critic model for both the left and right racket. So they both always did the exact same thing!

Therefore, even after hundreds of episodes they weren't really working in coordination to get a reward — therefore bootstrapping was not happening. Therefore, after looking around on the internet I realized that I needed a different algorithm. MADDPG looked like it could handle this game.

There are not many straight forward tutorials or blogs that explain the difference between DDPG and MADDPG in depth. In fact, I want this

document to serve as a possible guide for others as a simple way to modify DDPG to turn it into MADDPG for multiple agents that need to coordinate their actions. The blogs out there give a high level explanation of the differences, but I couldn't really find any implementation details. However poking around on blogs such as this one https://openai.com/blog/learning-to-cooperate-compete-and-communicate/, I realized that the primary differences between DDPG and MADDPG is that there are separate actor models for each agent.

There is also a centralized critic model, and a centralized replay buffer. So I simply modified the the DDPG algorithm from my project 2 slightly to fit MADDPG. As illustrated below, I created two separate actor / actor target models for both agents. And then a centralized critic model that helps both of the target models to learn!

```
# Networks

self.actor_left = Actor(self.num_states, hidden_size, hidden_size2, self.num_actions)  #**LEFT RACKET
self.actor_target_left = Actor(self.num_states, hidden_size, hidden_size2, self.num_actions) #**LEFT RACKET

self.actor_right = Actor(self.num_states, hidden_size, hidden_size2, self.num_actions)  #**RIGHT RACKET
self.actor_target_right = Actor(self.num_states, hidden_size, hidden_size2, self.num_actions) #**RIGHT RACKET

self.critic = Critic(self.num_states + self.num_actions, hidden_size, hidden_size2, self.num_actions) #**SINGLE CRITIC
self.critic_target = Critic(self.num_states + self.num_actions, hidden_size, hidden_size2, self.num_actions) #**SINGLE

if load_modelz:         #***LOAD STORED MODELS!!
    self.load_model_inner(model_list)

for target_param, param in zip(self.actor_target_left.parameters(), self.actor_left.parameters()):
    target_param.data.copy_(param.data)

for target_param, param in zip(self.actor_target_right.parameters(), self.actor_right.parameters()):
    target_param.data.copy_(param.data)

for target_param, param in zip(self.critic_target.parameters(), self.critic.parameters()):
    target_param.data.copy_(param.data)
```

I then created two separate update functions within the DDPG class. One is called update_right() and the other update_left(). The update_left() model simply updates the actor models for the left racket using the centralized critic network.

```
def update_left(self, batch_size, train_model):
    if not train_model:  #***Want to make sure we have the ability to play without trianing
        return
    states, actions, rewards, next_states, _ = self.memory.sample(batch_size)
    states = torch.FloatTensor(states)
    actions = torch.FloatTensor(actions)
    rewards = torch.FloatTensor(rewards)
    next_states = torch.FloatTensor(next_states)

    # Critic loss                  #**WE DONT USE ACTOR NETWORK HERE. JUST CRITIC, ACTOR TARGET, CRITIC TARGET.
    Qvals = self.critic.forward(states, actions)  #***GET QVALUES FROM CRITIC
    next_actions = self.actor_target_left.forward(next_states)  #***GET NEXT ACTIONS FROM ACTOR TARGET NETWORK
    next_Q = self.critic_target.forward(next_states, next_actions.detach()) #***GET NEXT Q FROM CRITIC TARGET
    Qprime = rewards + self.gamma * next_Q
    critic_loss = self.critic_criterion(Qvals, Qprime.detach())
```

The update function update_right() updates the right rackets target models using the centralized critic model.

```
def update_right(self, batch_size, train_model):
    if not train_model:  #***Want to make sure we have the ability to play without trianing
        return
    states, actions, rewards, next_states, _ = self.memory.sample(batch_size)
    states = torch.FloatTensor(states)
    actions = torch.FloatTensor(actions)
    rewards = torch.FloatTensor(rewards)
    next_states = torch.FloatTensor(next_states)

    # Critic loss                  #**WE DONT USE ACTOR NETWORK HERE. JUST CRITIC, ACTOR TARGET, CRITIC TARGET.
    Qvals = self.critic.forward(states, actions)  #***GET QVALUES FROM CRITIC
    next_actions = self.actor_target_right.forward(next_states)  #***GET NEXT ACTIONS FROM ACTOR TARGET NETWORK
    next_Q = self.critic_target.forward(next_states, next_actions.detach()) #***GET NEXT Q FROM CRITIC TARGET
    Qprime = rewards + self.gamma * next_Q
    critic_loss = self.critic_criterion(Qvals, Qprime.detach())
```

I also noticed an annoying issue where in the beginning, the rackets sort of acted too similar in nature. Delaying the encountered rewards allowing for bootstrapping. Therefore, I changed the algorithm slightly to alternate between adding the left and right rackets information to the replay buffer between each time step. It does seem like it took a lot of episodes for my algorithm to learn the game effectively, so it is likely this modification as a whole slowed down the training time as it reduced the number of experiences available to the neural network. I think a possible optimization would be to turn off this feature after a set amount of episodes (once the agents begin learning the game a bit).

```
if train_model:
    #throttle_model_update = throttle_model_update + 1 #**lets update the model less often
    #if throttle_model_update >= throttle_model_max:
    shat = len(agent.memory)
    if len(agent.memory) > batch_size:
        if train_side:
            agent.update_left(batch_size, train_model)
        else:
            agent.update_right(batch_size, train_model)
        if train_side:
            train_side = 0
        else:
            train_side = 1
```

Displayed below is the Torch model I used for the Actor networks. As illustrated the first three layers are simply Linear layers. Each of these Linear layers is passed through a Sigmoid activation function. I found that in RL algorithms, this helps to prevent a cluster of weights from going dead during the forward passes. The final linear layer is passed through a Tanh activation function to ensure it matches the actions necessary for the game — which should be between 1 and -1. The bias flag is set to ensure we start with Xavier initializations to help prevent exploding gradients — which I have learned from this Nanodegree can be a big problem in reinforcement learning.

```python
class Actor(nn.Module):
    def __init__(self, input_size, hidden_size, hidden_size2, output_size):
        super(Actor, self).__init__()
        #self.batch_norm = nn.BatchNorm1d(hidden_size)
        self.linear1 = nn.Linear(input_size, hidden_size, bias=True)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, hidden_size2)
        self.linear4 = nn.Linear(hidden_size2, output_size)

    def forward(self, state):
        """
        Param state is a torch tensor
        """
        #x = self.batch_norm(state)
        x = torch.sigmoid(self.linear1(state))
        x = torch.sigmoid(self.linear2(x))
        x = torch.sigmoid(self.linear3(x))
        x = torch.tanh(self.linear4(x)) #*TAN to scale us to the scores range!
        #throttle_check = 0
        #explode_check = x.tolist()
        #explode_check = explode_check[0]
        #for idx in explode_check:
        #    if idx == -1 or idx == 1:
        #        throttle_check += 1
        #if throttle_check >= 1:
        #    print("GRADIENTS EXPLODED. STOPPING TRAINING ITS USELESS NOW. YOU ARE IN TROUBLE")
        #    import sys
        #    sys.exit(1)
        return x
```

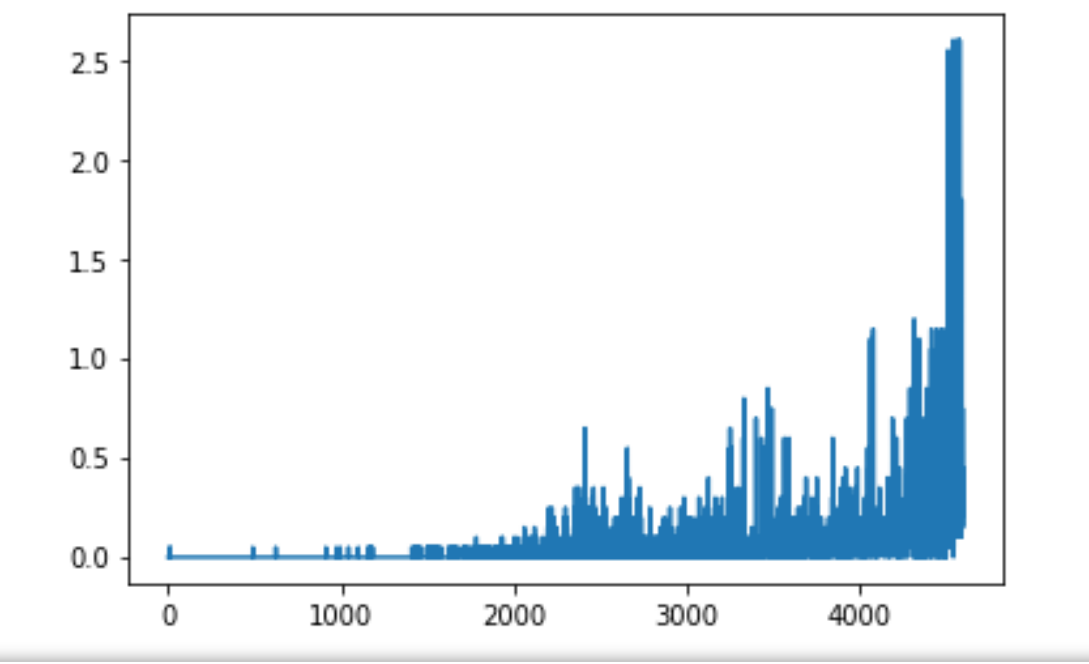
And illustrated below is my critic model which has a similar structure to the Actor model. For both the actor and critic models the learning rates are set at 0.0005. The hidden_size parameter used in layers 1, 2, and 3 is set to 48. And the hidden_size2 parameter used in layer 4 is set to 32. This seems to be the minimum number of parameters necessary to train the model to play the game well. I always try to use the minimum number of parameters possible for a given task because if the data coming in gets massive, training can take too long with too many parameters.

```python
class Critic(nn.Module):
    def __init__(self, input_size, hidden_size, hidden_size2, output_size):
        super(Critic, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size, bias=True) #**XAVIER init
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, hidden_size2)
        self.linear4 = nn.Linear(hidden_size2, output_size)

    def forward(self, state, action):
        """
        Params state and actions are torch tensors
        """
        x = torch.cat([state, action], 1)
        x = torch.sigmoid(self.linear1(x))  #**sigmoid activation to help guard against exploding gradients!
        x = torch.sigmoid(self.linear2(x))
        x = torch.sigmoid(self.linear3(x))
        x = torch.tanh(self.linear4(x))

        return x
```

As the chart below illustrates, this algorithm was able to very strongly solve the game within around 4000 episodes. Obviously this could be reduced by tuning hyper parameters. I didn't really do this I mostly just kept the parameters the same from Project 2. Also as an interesting side note, using the "noise" generated by the Ornstein-Ulhenbeck Process was not as useful in having MADDPG solve this game. In fact, I turned it off completely after 1000 episodes and the algorithm still solved the game just fine.



Ideas For Future Work:

Immediately after I finish the Udacity Reinforcement Learning Nanodegree, I plan on taking the Udacity Flying Cars program. I want to utilize the Reinforcement Learning algorithms I have learned in this Nanodegree program to help me write interesting apps for autonomously flying drones. During the Flying Cars program I plan on purchasing a very nice drone that has a software development kit. And using that to develop an autonomous flying app. I feel like the DDPG and MADDPG algorithms, and other RL algorithms I have picked up in this Nanodegree program, will greatly assist me in that task.