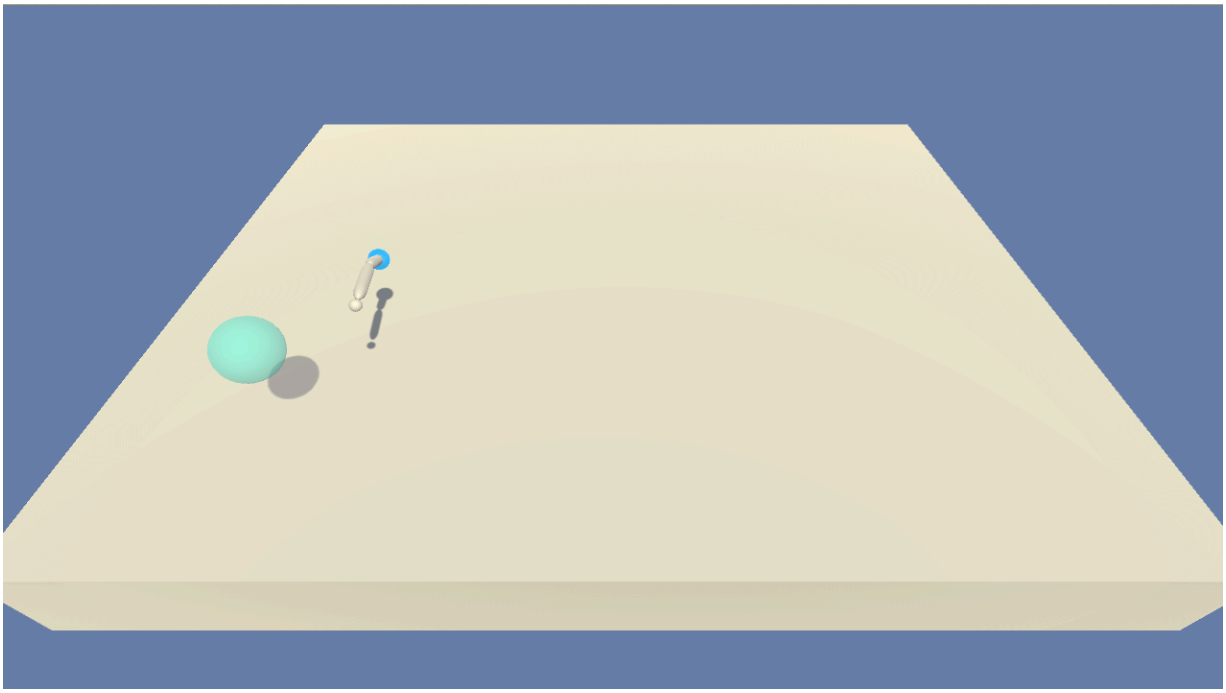
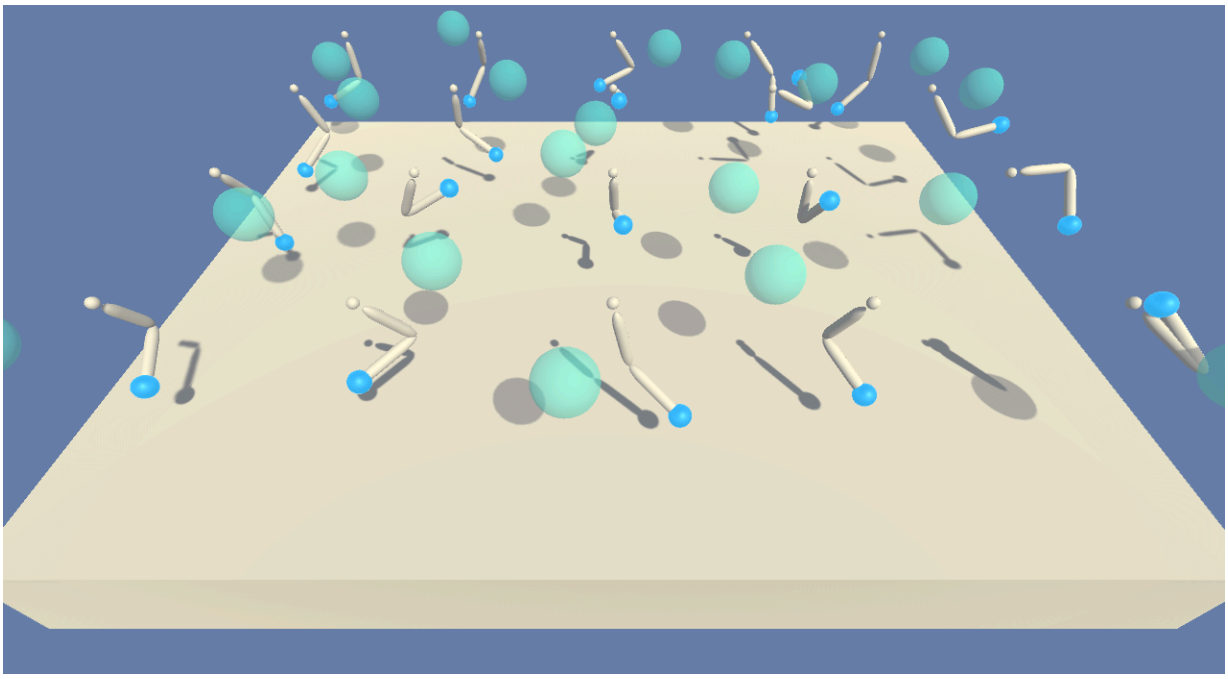
Name: Mark Abrenio

Project: Udacity Project 2 (Deep Reinforcement Learning)

For this project we were instructed to solve a simple video game where an arm continually tries to reach out and touch a ball, that is spinning around the arm at varying speeds.





For this project I utilized the DDPG algorithm. When I first saw the game, I thought I could solve it in a simple way utilizing the REINFORCE algorithm. I actually wanted to use the REINFORCE algorithm because it is a nice policy gradient algorithm and it is very easy to understand. The mathematics behind it is not complex. However, I quickly realized that the algorithm would be difficult to utilize for this task.

The reason was, the REINFORCE algorithm chooses actions by sampling a probability distribution. And that makes sense if you have a small number of actions such as going forward, backward, up, or down like in the Banana navigation game from project 1. However, in this game the action itself is made up of 4 floating point numbers. And the variance between these four numbers is massive. So how on earth can you discretize such an action so you can use the REINFORCE algorithm to make

choices within this environment? There is really no possible way I could think to do this in the time frame I had to do this project.

Therefore, I began researching the DDPG algorithm. I already understand the DQN algorithm very well from project 1 and it seemed powerful. I read that the DDPG is much like the DQN except that it can handle discreet spaces. Most of the material I read read on the DDPG is located on the blog: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b. I also used some of the skeleton code from this blog too including the QUNoise class which implements the Ornstein-Ulhenbeck Process used in DDPG for exploration. The model utilized in this blog did not work very well for this project though. Displayed below is the original actor and critic models from the blog:

```
class Critic(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Critic, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state, action):
        """
        Params state and actions are torch tensors
        """
        x = torch.cat([state, action], 1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)

        return x

class Actor(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, learning_rate = 3e-4):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state):
        """
        Param state is a torch tensor
        """
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = torch.tanh(self.linear3(x))

        return x
```

As I was testing with this model I realized that the learning of the agent was not progressive, and was not very stable. When I debugged the data passing through this model it was obvious that a lot of dead weights were

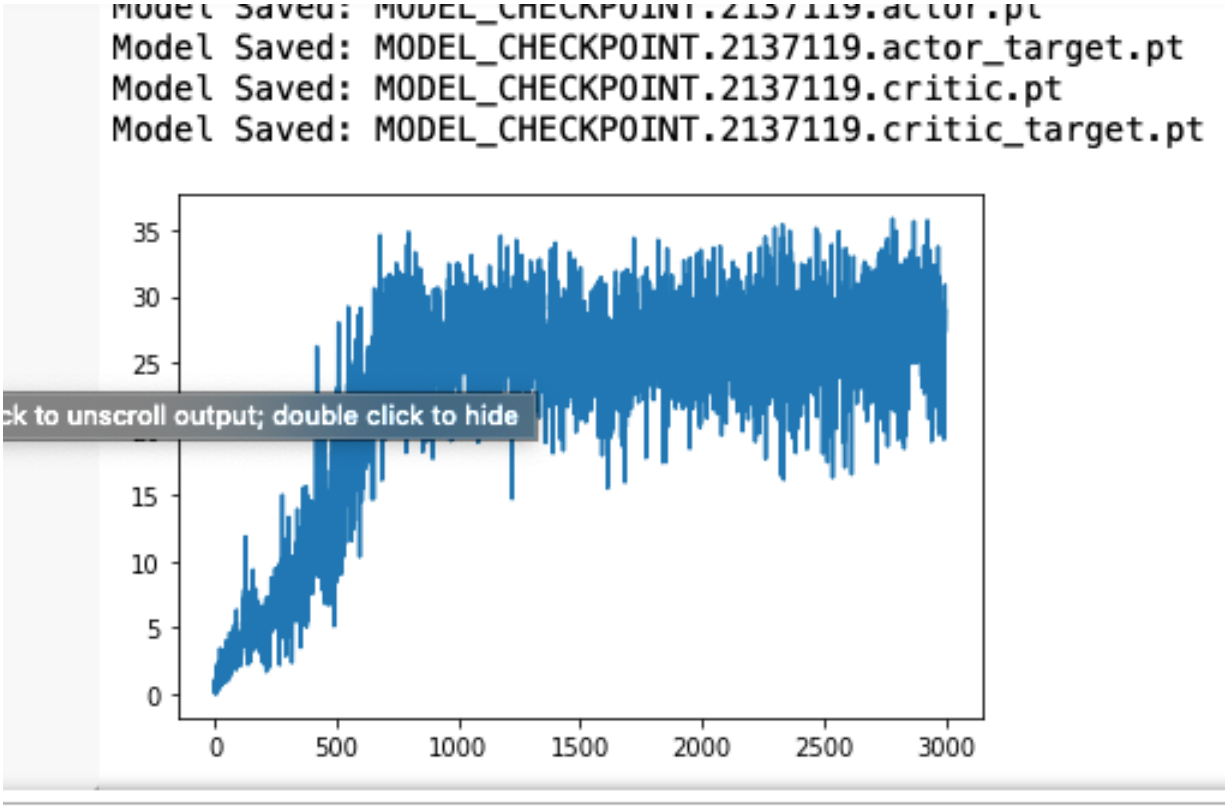
beginning to form after the Relu activation functions. Due to this I added a third hidden layer and also flipped the RELU activation functions over to Sigmoid to help prevent dead weights from congregating. The final Critic model ended up looking like this:

```
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
```

I also added the bias=True to the first layer to allow for Xavier initializations of the weights because I noticed that exploding gradients were a huge problem with this game training with DDPG. In fact, I tried for hours to train the model with the a learning rate of .005. The model would never learn and this made me very frustrated. I thought perhaps I had implemented DDPG incorrectly in regards to this game. But when I debugged the forward pass through the actor model I saw the following, indicating I had an exploding gradient problem.

```
    """
    #x = self.batch_norm(state)
    x = torch.sigmoid(self.linear1(state))  x: tensor([[-1.0000,  1.0000, -1.0000, -1.0000]])
    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  throttle_check: 0
    explode_check = x.tolist()  explode_check: [-0.9999998211860657, 0.9999995827674866, -0.9999994039535522, -0.9999994039535522]
    explode_check = explode_check[0]
    for idx in explode_check:
        if idx == -1 or idx == 1:
            throttle_check += 1
    if throttle_check >= 1:
```

Therefore, I lowered the initial learning rate to .0005 for both the actor and the critic models. After this the single agent game began learning very well. I implemented a learning rate decay strategy that basically lowered the learning rates when the score hit 30. It took a while but I finally was able to solve the single agent game Continuous_Control.ipynb:



Next I tried to solve the 20 agent version of the game.

This one was much more difficult. I essentially just took the agents and split them up. I took each of their states, actions, next states, rewards, etc and pushed them through the agent.memory.push() function the same way I did with the single agent game. So essentially for every time step in an episode, there would be information from 20 different agents added to the replay memory buffer. This strategy did not work, and I again started getting the exploding gradient problem. To fix this I tried to drastically lower the learning rate. No matter how low I would set the learning rate, the exploding gradient problem would appear almost immediately.

This was very confusing to me because why would the training work with the single agent game, but not with the multi agent game? The only thing I was doing different was pushing 20 agents through instead of just 1. So as an experiment I tried to lower the number of agents from the 20 agent game I pushed through the memory replay buffer. First I randomly picked four. I figured if it could learn the tasks of four of them it could do all their tasks — because all 20 are doing kind of the same thing.

After I only starting pushing the first four agents through the replay buffer, the agents began to learn the game very quickly. I ended up settling for pushing five agents through the replay buffer. I do not really understand why it would not learn while pushing all 20 agents through at once — but I am assuming it was just too much data and noise going into the neural network at one single pass. It just simply confused the network and it couldn't understand the patterns from all 20 agents at once. But

once I moved the number of agents down to 4 or 5 it learned much faster than even the single agent game. Likely because it had a higher variety of actions to learn from.

Ideas For Future Work:

The main reason I first chose the REINFORCE algorithm to try to implement was because it is simple and it looked like very low overheard. I was excited about this because I plant to take the Udacity Flying Car nano degree and I figured I could utilize the REINFORCE algorithm to help fly the drone because it is so low overhead due its simplicity. Since it will be running on a drone I wanted something with low overheard. But then I realized the limitations of REINFORCE from this project. The training piece of the DDPG network is pretty heavy as illustrated below:

```
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
Qvals = self.critic.forward(states, actions)
next_actions = self.actor_target.forward(next_states)
next_Q = self.critic_target.forward(next_states, next_actions.detach())
Qprime = rewards + self.gamma * next_Q
critic_loss = self.critic_criterion(Qvals, Qprime.detach())

# Actor loss
policy_loss = -self.critic.forward(states, self.actor.forward(states)).mean()

# update networks
self.actor_optimizer.zero_grad()
policy_loss.backward()
self.actor_optimizer.step()

self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()
```

It requires four Neural networks. The state must first be pushed through an Actor network, getting back your action space (Q Values) while training/playing the game. Then during training the critic model is utilized to generate another group of Q values. The actor_target network is utilized to attain another batch of Next Q Values. Notably, the critic model requires the state space values and action space values to be ocncantenated together which actually means your first layer has an input size of len(state) plus len(action).

```
def forward(self, state, action):  self: Critic(\n  (linear1): Linear(in_fea
    """
    Params state and actions are torch tensors
    """
    x = torch.cat([state, action], 1)
    x = torch.sigmoid(self.linear1(x))  #**sigmoid activation to help guard
    x = torch.sigmoid(self.linear2(x))
    x = torch.sigmoid(self.linear3(x))
    x = torch.tanh(self.linear4(x))

    return x
```

The critic loss is then of course generated from adding the rewards to the critic Q values times gamma. All of this requires a lot of overhead for a drone. My future work involves attempting to implement at least the actor model in C++ (using pytorch c++) and also using image pixels as state spaces for input in the DDPG network so that the agent can navigate around using video feeds.