DeepRL-PRJ-Collaboration-Competition

This project repository contains my work for the Udacity's Deep Reinforcement Learning Nanodegree Project 3: Collaboration and Competition.

1.Project Details

Project Description

In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a positive reward. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a negative reward. Thus, the goal of each agent is to keep the ball in play.

Reward

In this environment, 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.

State Space

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.

Action

The action space is continuous, which allows each agent to execute more complex and precise movements. There's an unlimited range of possible action values to control the rocket. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

Project Goal

The goal of each agent is to keep the ball in play. 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.

The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5.

2.The Environment

The project environment provided by Udacity is similar to, but not identical to, the Unity's Tennis environment on the Unity ML-Agents GitHub page.

In this environment, 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 movement 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.

The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5.

3.Agent Implementation

Learning Algorithm:

The environment has two key aspects:

a. **Multiple agents**: The Tennis environment has multiple agents – specifically, 2 different agents.
b. **Continuous action space**: Each tennis agent can only move forward, backward, or jump - which is an unlimited range of possible action values that control these movements. The

action space is *continuous*, which allows each agent to execute more complex and precise movements.

Policy-based methods are well-suited for continuous action space. This project uses an off-policy method called Multi Agent Deep Deterministic Policy Gradient (MADDPG) algorithm. MADDPG is based on the original DDPG algorithm which is outlined in this paper, Continuous control with deep reinforcement learning, by researchers at Google Deepmind.

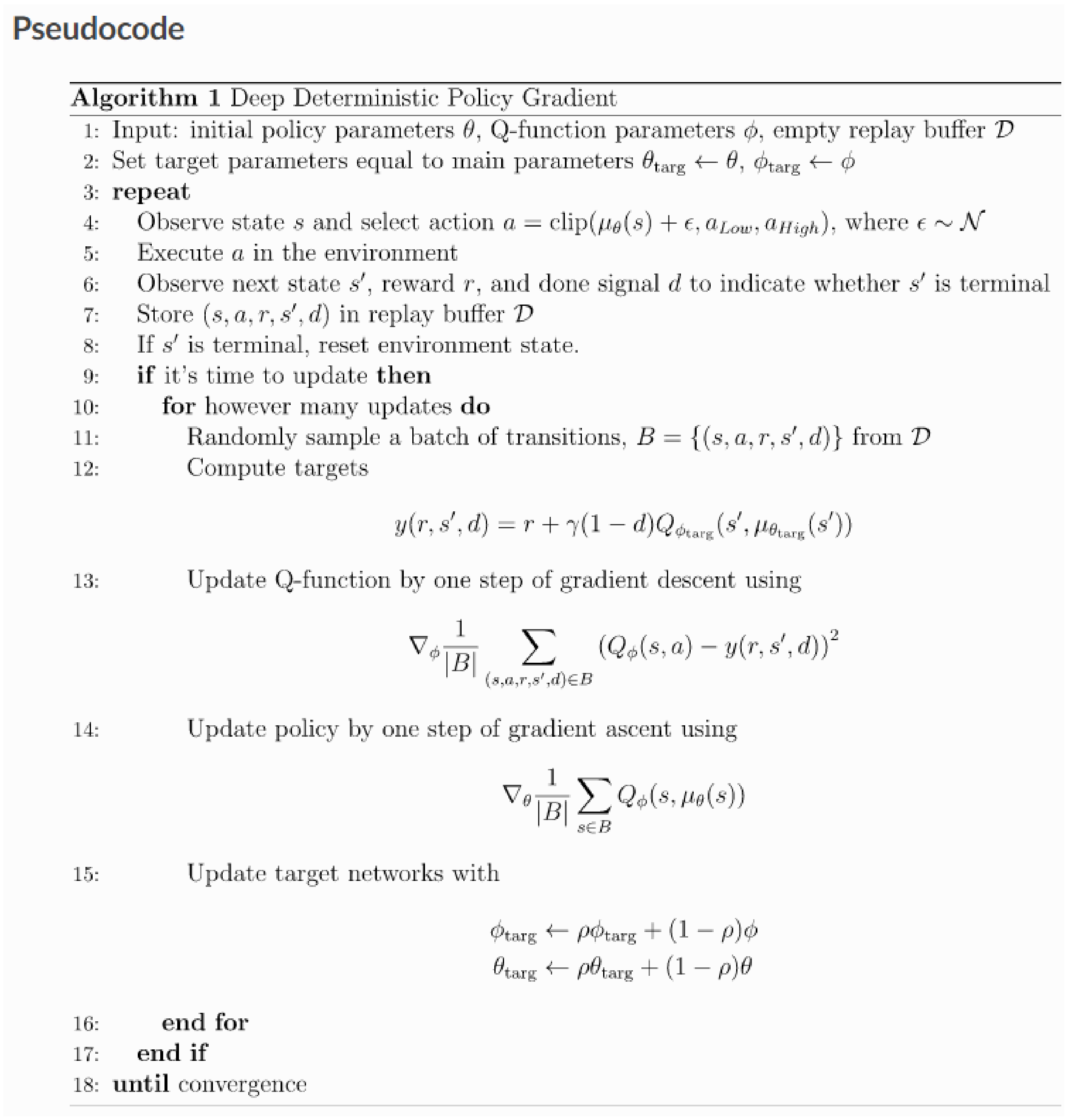
In this paper, the authors present "a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces." They highlight that DDPG can be viewed as an extension of Deep Q-learning to continuous tasks / action spaces. Using the same learning algorithm, network architecture and hyper-parameters, their algorithm robustly solves more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving. Their algorithm is able to find policies whose performance is competitive with those found by a planning algorithm with full access to the dynamics of the domain and its derivatives. They further demonstrate that for many of the tasks the algorithm can learn policies “end-to-end”: directly from raw pixel inputs.

Deep Deterministic Policy Gradient (DDPG)
Below details and the algorithm screenshot are taken from the DDPG algorithm from the Spinning Up website

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function and uses the Q-function to learn the policy.

DDPG is an off-policy algorithm. DDPG can only be used for environments with continuous action spaces. DDPG can be thought of as being deep Q-learning for continuous action spaces.

Deep Deterministic Policy Gradient (DDPG) leverages the class of Actor-Critic Methods. The actor produces a deterministic policy instead of the usual stochastic policy and the critic evaluates the deterministic policy. The critic is updated using the TD-error and the actor is trained using the deterministic policy gradient algorithm.



Pseudocode

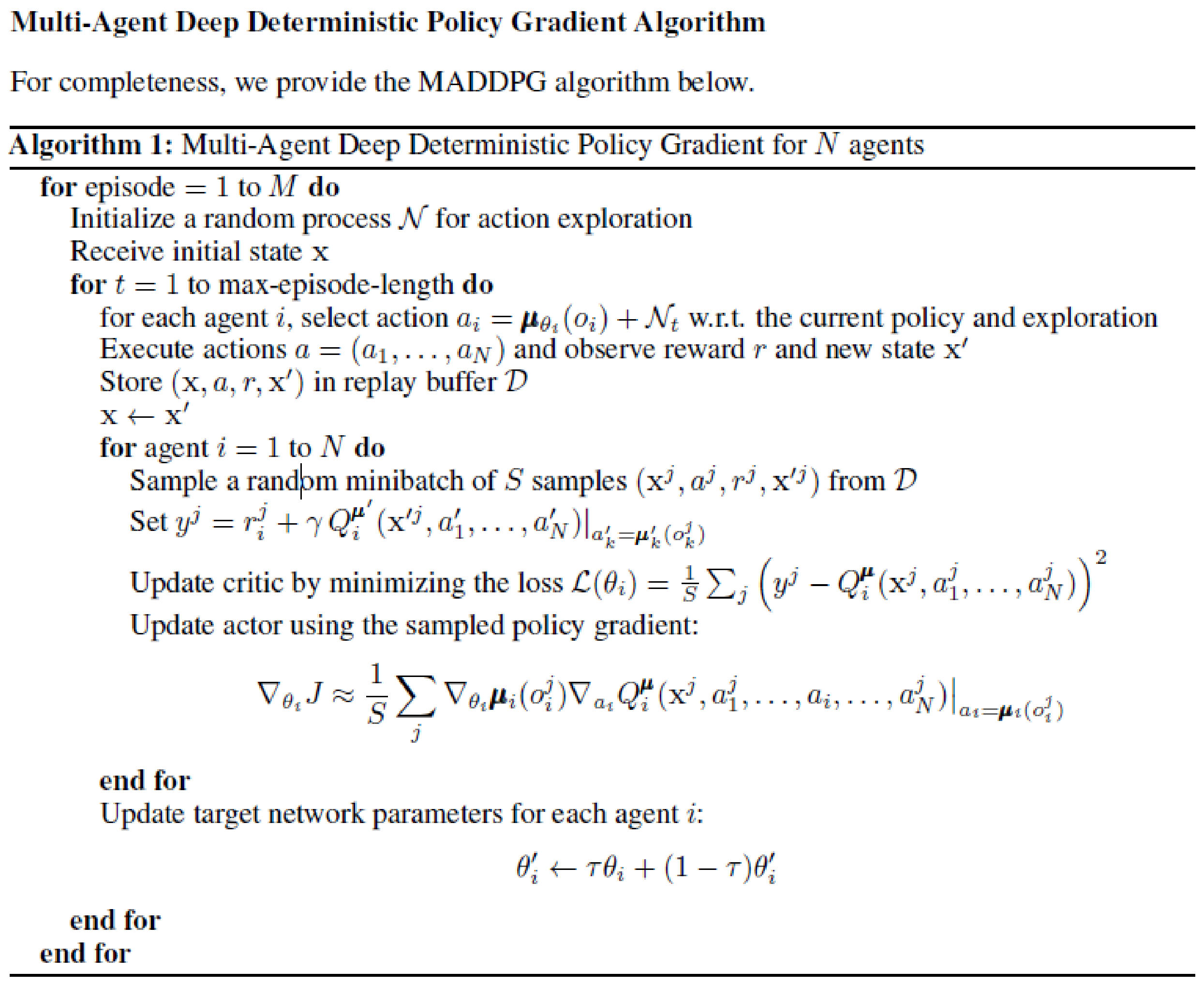
Algorithm 1 Deep Deterministic Policy Gradient

```
1:  Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2:  Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3:  repeat
4:      Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:      Execute $a$ in the environment
6:      Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:      Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:      If $s'$ is terminal, reset environment state.
9:      if it's time to update then
10:         for however many updates do
11:             Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:             Compute targets
                    $y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:             Update Q-function by one step of gradient descent using
                    $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$
14:             Update policy by one step of gradient ascent using
                    $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:             Update target networks with
                    $\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$
                    $\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$
16:         end for
17:     end if
18: until convergence
```

I took the DDPG foundation in single-agent DDPG and modified it. For this project I have used a variant of DDPG called **Multi Agent Deep Deterministic Policy Gradient (MADDPG)** which is described in the paper Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments, by Lowe and Wu, along with other researchers. The paper explores deep reinforcement learning methods for multi-agent domains. I implemented their variation of the Multi-Agent Actor Critic method (see Figure 1).

To make this algorithm work for the multiple competitive agents in the Tennis environment, I implemented components discussed in the above paper. I further experimented with components of the DDPG algorithm based on other concepts covered in Udacity's classroom and lessons.

The algorithm is taken from above paper and is as follows:



Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm below.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

```
for episode = 1 to $M$ do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial state $\mathbf{x}$
    for $t = 1$ to max-episode-length do
        for each agent $i$, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
        Execute actions $a = (a_1, \ldots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
        Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
        $\mathbf{x} \leftarrow \mathbf{x}'$
        for agent $i = 1$ to $N$ do
            Sample a random minibatch of $S$ samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
            Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
            Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
            Update actor using the sampled policy gradient:
                $$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$
        end for
        Update target network parameters for each agent $i$:
                $$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$
    end for
end for
```

The concept behind this algorithm is summarized in this illustration taken from the paper:

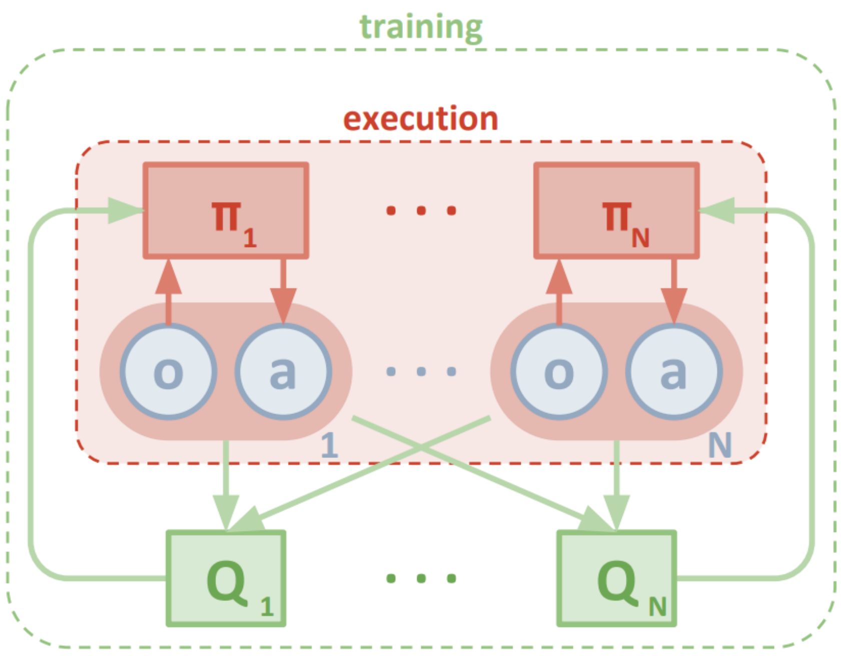


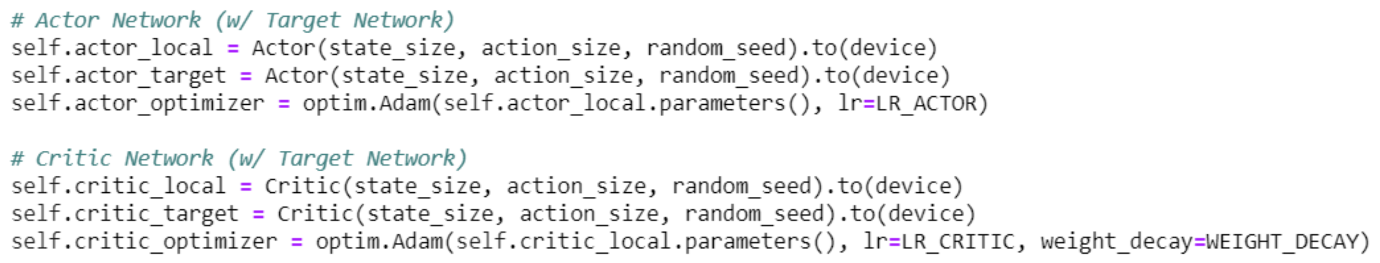
Figure 1: Overview of their multi-agent decentralized actor, centralized critic approach.

Multi-Agent Actor Critic:

The policy-based methods tend to have high variance and low bias and use Monte-Carlo estimate whereas the value-based methods have low variance but high bias as they use TD estimates. The agent can learn the policy directly from the states using Policy-based methods. Using a policy-based approach, the agent (actor) learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent. Now, the actor-critic methods were introduced to solve the bias-variance problem by combining the two methods. In Actor-Critic, the actor is a neural network which updates the policy and the critic is another neural network which evaluates the policy being learned which is, in turn, used to train the actor. In vanilla policy gradients, the rewards accumulated over the episode is used to compute the average reward and then, calculate the gradient to perform gradient ascent. Now, instead of the reward given by the environment, the actor uses the value provided by the critic to make the new policy update.

This implementation follows the **decentralized actor with centralized critic** approach from the paper [Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments](), by Lowe and Wu. Traditional actor-critic methods have a separate critic for each agent, whereas this implementation utilizes a single critic that receives the actions and state observations as input from all agents. This allows for centralized training with decentralized execution. Each agent still takes actions based on its own unique observations of the environment.

We are leveraging local and target networks to improve stability. This is where one set of parameters w is used to select the best action, and another set of parameters w' is used to evaluate that action. In this project, local and target networks are implemented separately for both the actor and the critic.



```
# Actor Network (w/ Target Network)
self.actor_local = Actor(state_size, action_size, random_seed).to(device)
self.actor_target = Actor(state_size, action_size, random_seed).to(device)
self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

# Critic Network (w/ Target Network)
self.critic_local = Critic(state_size, action_size, random_seed).to(device)
self.critic_target = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)
```

Exploration vs Exploitation

The **exploration-exploitation trade-off** is a well-known problem that occurs in scenarios where a learning system has to repeatedly make a choice with uncertain payoffs. In essence, the dilemma for a decision-making system that only has incomplete knowledge of the world is whether to repeat decisions that have worked well so far (exploit) or to make novel decisions, hoping to gain even greater rewards (explore). In this case, while the agent is still learning the optimal policy, whether the agent should choose an action based on the rewards observed thus far or should the agent try a new action in hopes of earning a higher reward. This is known as the **exploration vs. exploitation dilemma**.

We will use **Ornstein-Uhlenbeck (OU) Noise** as suggested in the paper Continuous control with deep reinforcement learning, by researchers at Google Deepmind. The process adds noise to the action values at each time step. For discrete action spaces, exploration is done via probabilistically selecting a random action such as epsilon-greedy or Boltzmann exploration. For continuous action spaces, exploration is done via adding noise to the action itself. This noise is correlated to previous noise and therefore tends to stay in the same direction for longer durations without canceling itself out. This allows the agent to maintain velocity and explore the action space with more continuity.

In total, there are five hyperparameters related to this noise process. The Ornstein-Uhlenbeck process itself has three hyperparameters that determine the noise characteristics and magnitude:

- mu: the long-running mean
- theta: the speed of mean reversion
- sigma: the volatility parameter. The reduced noise volatility seemed to help the model converge.

We used an epsilon parameter to decay the noise level over time. This decay mechanism ensures that more noise is introduced earlier in the training process (i.e., higher exploration), and the noise decreases over time as the agent gains more experience (i.e., higher exploitation). The starting value for epsilon and its decay rate are two hyperparameters that were tuned during experimentation.

The final noise parameters were set as follows:

```
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.12         # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
EPS_START = 5.5         # initial value for epsilon in noise decay process
EPS_EP_END = 250        # episode to end the noise decay process
EPS_FINAL = 0           # final value for epsilon after decay
```

The EPS_START parameter is set at 5.0. This higher EPS_START helped the model to converge better in less number of episodes. This helped the agent to discover signal early in the process. By boosting the noise output from the Ornstein-Uhlenbeck (OU) process, it encouraged aggressive exploration of the action space and therefore improved the chances of making contact with the ball. This extra signal seemed to improve learning later in training once the noise decayed to zero.

Learning Interval

I implemented an interval in which the learning step is performed every episode. As part of each learning step, the algorithm then samples experiences from the buffer and runs the Agent.learn() method 10 times.

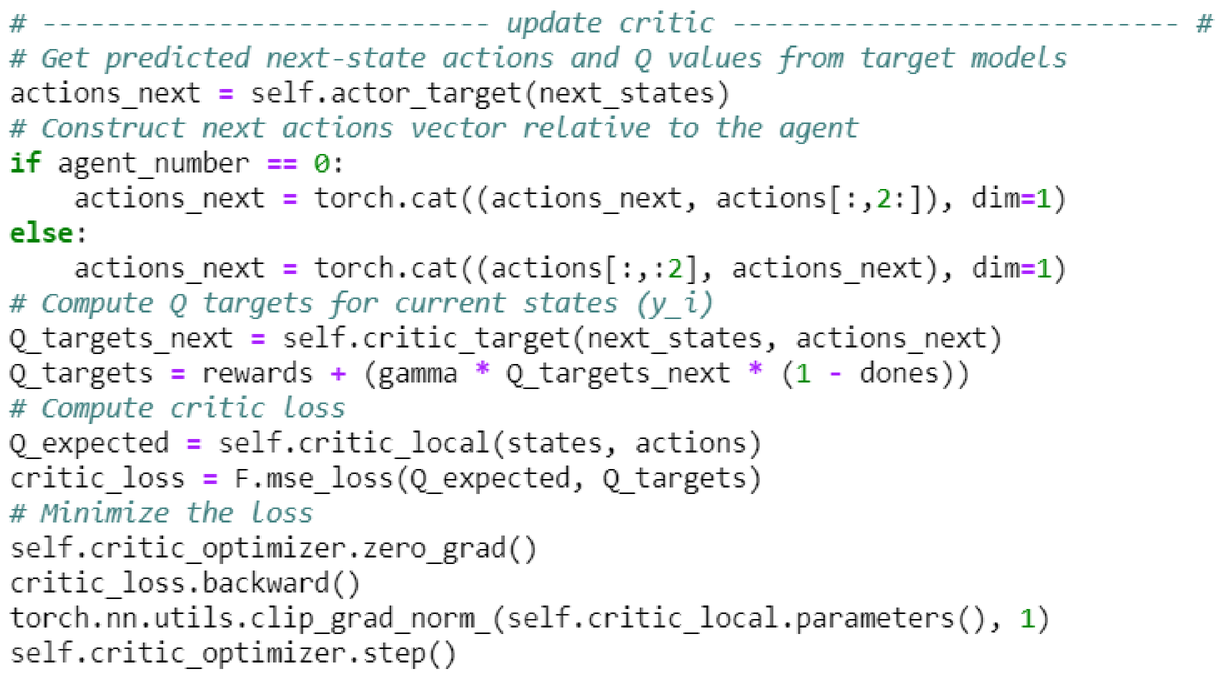
```
LEARN_EVERY = 1         # learning timestep interval (# of episodes)
LEARN_NUM = 1           # number of learning passes per timestep
```

Gradient Clipping

The issue of exploding gradients is described in [A Gentle Introduction to Exploding Gradients in Neural Networks]() by Jason Brownlee.

Exploding gradients are a problem where large error gradients accumulate and result in very large updates to neural network model weights during training. This has the effect of your model being unstable and unable to learn from your training data. Essentially, each layer of your net amplifies the gradient it receives. This becomes a problem when the lower layers of the network accumulate huge gradients, making their respective weight updates too large to allow the model to learn anything.

I implemented gradient clipping using the torch.nn.utils.clip_grad_norm function. I set the function to "clip" the norm of the gradients at 1, therefore placing an upper limit on the size of the parameter updates and preventing them from growing exponentially. This change along with batch normalization, the model became much more stable and the agent started learning at a much faster rate.



```
# ---------------------------- update critic ---------------------------- #
# Get predicted next-state actions and Q values from target models
actions_next = self.actor_target(next_states)
# Construct next actions vector relative to the agent
if agent_number == 0:
    actions_next = torch.cat((actions_next, actions[:,2:]), dim=1)
else:
    actions_next = torch.cat((actions[:,:2], actions_next), dim=1)
# Compute Q targets for current states (y_i)
Q_targets_next = self.critic_target(next_states, actions_next)
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
# Compute critic loss
Q_expected = self.critic_local(states, actions)
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()
```

Experience Replay

In reinforcement learning, Experience Replay allows the agent to learn from past experience. The algorithm employs a replay buffer to gather experiences. Experiences are stored in a single replay buffer as each agent interacts with the environment. These experiences are then utilized by the central critic, therefore allowing the agents to learn from each other's' experiences.

The agent interacts with the environment and learns from sequence of state(S), actions(A) and rewards(R) and next state(S') which is stored in the form of an experienced tuple. These sequence of experience tuples can be highly correlated which can lead to oscillation or divergence in the action values. This can be prevented by having a replay buffer, from which experience replay tuples are used to randomly sample from the buffer. The replay buffer contains a collection of experience tuples (S, A, R, S´). The tuples are gradually added to the buffer as we are interacting with the environment. The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

4.Code Implementation

Github link: https://github.com/kishoregajjala/DeepRL-PRJ-Collaboration-Competition

Readme.md provides more details of the project and environment.

The implementation of this project consists these files:

a. **model.py**: This implements the **Actor** and the **Critic** classes which each implements a Target and a Local Neural Networks used for the training.

b. **maddpg_agent.py**: This implements the Multi agent based DDPG agent, OUNoise and replay buffer memory which is used by the agent.
   - The Actor's Local and Target neural networks, and the Critic's Local and Target neural networks are instantiated by the Agent's constructor

   - The **step()** method saves experience in replay memory, and use random sample from buffer to learn

   - The **act()** method returns actions for both agents as per current policy, given their respective states

   - The **learn()** method updates the policy and value parameters using given batch of experience tuples.

   - The **soft_update()** method:

In DDPG, target networks are updated using a soft update strategy. In DDPG, you have two copies of your network weights for each network - regular for the actor, regular for the critic, and a target for the actor, and a target for the critic. A soft update strategy consists of slowly blending in your regular network weights with your target network weights. So, every timestep you make your target network be 99.99 percent of your target network weights and only 0.01 percent of your regular network weights. You are slowly mixing your regular network weights into your target network weights. The regular network is the most up to date network because it is the one, we are training while the target network is the one, we use for prediction to stabilize strain. This way you will get faster convergence by using the update strategy.

- **Implements Ornstein-Uhlenbeck (OU) Noise** as explained above in the section #3

- **Implements ReplayBuffer class** as explained above in the section #3

c. **Tennis.ipynb**

This Jupyter notebooks allows to instantiate and train the agents – high level steps are below

- Prepare the Unity environment and Import the necessary packages
- Check the Unity environment
- Examine the State and Action Spaces
- Instantiate and train two agents
- Train and evaluate agents using Multi-Agent Deep Deterministic Policy Gradient (MADDPG) and various Hyperparameters
- Plot the score results

d. **MADDPG Hyperparameters**
Few notes:

- Reducing the sigma value used in Ornstein-Uhlenbeck (OU) Noise class also helped the agent in learning
- Adjusted the learning rate to improve the learning
- Hyperparameters and their values:

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 7e-2              # 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

LEARN_EVERY = 1         # learning timestep interval (# of episodes)
LEARN_NUM = 1           # number of learning passes per timestep
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.12         # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
EPS_START = 5.5         # initial value for epsilon in noise decay process
EPS_EP_END = 250        # episode to end the noise decay process
EPS_FINAL = 0           # final value for epsilon after decay
```

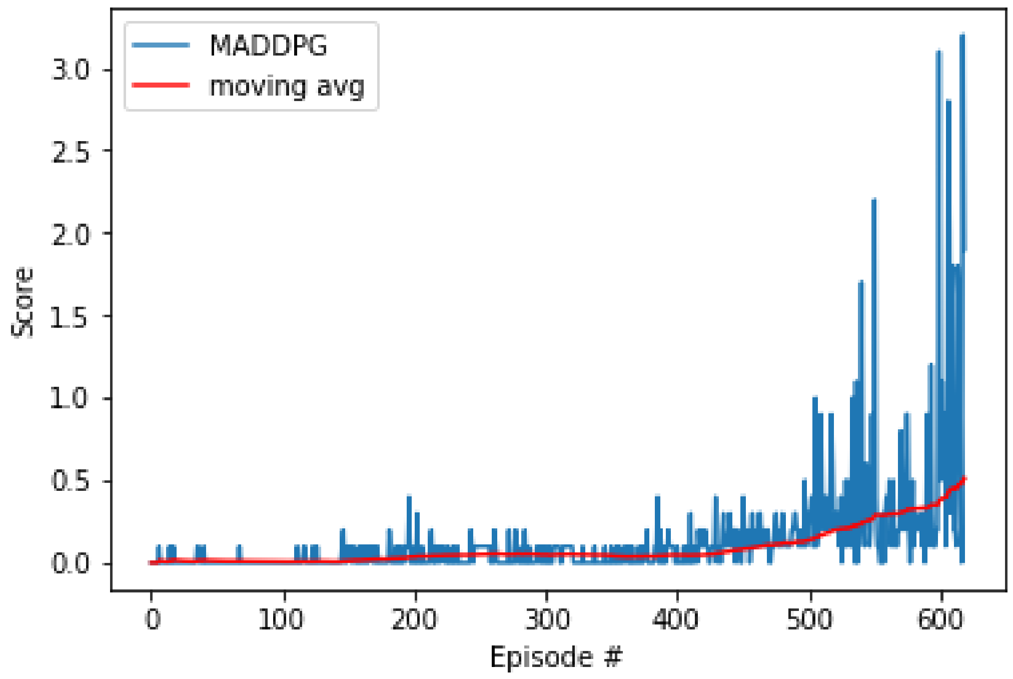
5. Results

The agents solved the Tennis environment with an average reward of at least +0.5 over 100 episodes, taking the best score from either agent for a given episode.

The graph below shows the final training results. The agents were able to solve the environment in 519 episodes and top moving average of 0.512. The result depends significantly on the fine tuning of the hyperparameters.

The plot of the rewards across episodes is shown below:

```
Episodes 0000-0010    Max Reward: 0.100     Moving Average: 0.010
Episodes 0010-0020    Max Reward: 0.100     Moving Average: 0.015
Episodes 0020-0030    Max Reward: 0.000     Moving Average: 0.010
Episodes 0030-0040    Max Reward: 0.100     Moving Average: 0.013
Episodes 0040-0050    Max Reward: 0.100     Moving Average: 0.012
Episodes 0050-0060    Max Reward: 0.000     Moving Average: 0.010
Episodes 0060-0070    Max Reward: 0.100     Moving Average: 0.010
Episodes 0070-0080    Max Reward: 0.000     Moving Average: 0.009
Episodes 0080-0090    Max Reward: 0.000     Moving Average: 0.008
Episodes 0090-0100    Max Reward: 0.000     Moving Average: 0.007
Episodes 0100-0110    Max Reward: 0.000     Moving Average: 0.006
Episodes 0110-0120    Max Reward: 0.100     Moving Average: 0.007
Episodes 0120-0130    Max Reward: 0.100     Moving Average: 0.009
Episodes 0130-0140    Max Reward: 0.000     Moving Average: 0.007
Episodes 0140-0150    Max Reward: 0.200     Moving Average: 0.010
Episodes 0150-0160    Max Reward: 0.100     Moving Average: 0.015
Episodes 0160-0170    Max Reward: 0.100     Moving Average: 0.020
Episodes 0170-0180    Max Reward: 0.100     Moving Average: 0.022
Episodes 0180-0190    Max Reward: 0.200     Moving Average: 0.027
Episodes 0190-0200    Max Reward: 0.400     Moving Average: 0.036
Episodes 0200-0210    Max Reward: 0.300     Moving Average: 0.040
Episodes 0210-0220    Max Reward: 0.200     Moving Average: 0.043
Episodes 0220-0230    Max Reward: 0.100     Moving Average: 0.045
Episodes 0230-0240    Max Reward: 0.100     Moving Average: 0.047
Episodes 0240-0250    Max Reward: 0.200     Moving Average: 0.050
Episodes 0250-0260    Max Reward: 0.100     Moving Average: 0.054
Episodes 0260-0270    Max Reward: 0.200     Moving Average: 0.051
Episodes 0270-0280    Max Reward: 0.200     Moving Average: 0.054
Episodes 0280-0290    Max Reward: 0.200     Moving Average: 0.053
Episodes 0290-0300    Max Reward: 0.100     Moving Average: 0.048
Episodes 0300-0310    Max Reward: 0.100     Moving Average: 0.050
Episodes 0310-0320    Max Reward: 0.100     Moving Average: 0.052
Episodes 0320-0330    Max Reward: 0.100     Moving Average: 0.050
Episodes 0330-0340    Max Reward: 0.100     Moving Average: 0.049
Episodes 0340-0350    Max Reward: 0.100     Moving Average: 0.043
Episodes 0350-0360    Max Reward: 0.100     Moving Average: 0.037
Episodes 0360-0370    Max Reward: 0.100     Moving Average: 0.038
Episodes 0370-0380    Max Reward: 0.200     Moving Average: 0.039
Episodes 0380-0390    Max Reward: 0.400     Moving Average: 0.043
Episodes 0390-0400    Max Reward: 0.200     Moving Average: 0.047
Episodes 0400-0410    Max Reward: 0.100     Moving Average: 0.044
Episodes 0410-0420    Max Reward: 0.300     Moving Average: 0.047
Episodes 0420-0430    Max Reward: 0.400     Moving Average: 0.057
Episodes 0430-0440    Max Reward: 0.300     Moving Average: 0.068
Episodes 0440-0450    Max Reward: 0.300     Moving Average: 0.080
Episodes 0450-0460    Max Reward: 0.400     Moving Average: 0.094
Episodes 0460-0470    Max Reward: 0.300     Moving Average: 0.108
Episodes 0470-0480    Max Reward: 0.300     Moving Average: 0.114
Episodes 0480-0490    Max Reward: 0.300     Moving Average: 0.123
Episodes 0490-0500    Max Reward: 0.500     Moving Average: 0.135
Episodes 0500-0510    Max Reward: 1.000     Moving Average: 0.169
Episodes 0510-0520    Max Reward: 0.900     Moving Average: 0.197
Episodes 0520-0530    Max Reward: 0.500     Moving Average: 0.208
Episodes 0530-0540    Max Reward: 1.100     Moving Average: 0.231
Episodes 0540-0550    Max Reward: 1.700     Moving Average: 0.270
Episodes 0550-0560    Max Reward: 2.200     Moving Average: 0.291
Episodes 0560-0570    Max Reward: 0.500     Moving Average: 0.298
Episodes 0570-0580    Max Reward: 0.900     Moving Average: 0.328
Episodes 0580-0590    Max Reward: 0.300     Moving Average: 0.330
Episodes 0590-0600    Max Reward: 3.100     Moving Average: 0.378
Episodes 0600-0610    Max Reward: 2.800     Moving Average: 0.452
<-- Environment solved in 519 episodes!
<-- Moving Average: 0.512 over past 100 episodes
```



6. Future Improvements

1. Improve our Multi-Agents project performance by implementing Prioritized experience replay. Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This can improve learning by increasing the probability that rare or important experience vectors are sampled.

   As taken from the abstract from the above paper:

   “Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. In this paper we develop a framework for prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently. We use prioritized experience replay in Deep Q-Networks (DQN), a reinforcement learning algorithm that achieved human-level performance across many Atari games. DQN with prioritized experience replay achieves a new state-of-the-art, outperforming DQN with uniform replay on 41 out of 49 games.”

2. I want to apply TD3 algorithm Twin Delayed DDPG (TD3) as presented in the OpenAI's Spinning Up website to further improve the stability and performance of our Multi agents environment.

   As taken key facts from the above paper:

While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyperparameters and other kinds of tuning. A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function. Twin Delayed DDPG (TD3) is an algorithm that addresses this issue by introducing three critical tricks:

Trick One: Clipped Double-Q Learning. TD3 learns *two* Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Trick Two: "Delayed" Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

Trick Three: Target Policy Smoothing. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Together, these three tricks result in substantially improved performance over baseline DDPG.

Quick Facts

- TD3 is an off-policy algorithm.
- TD3 can only be used for environments with continuous action spaces.
- The Spinning Up implementation of TD3 does not support parallelization.

3. **Batch Normalization** — I probably should have used batch normalization as I have used on my previous Deep RL project Continuous Control algorithm to train an

agent (robot with double-jointed arm) to move (it’s arm) to target locations. Running computations on large input values and model parameters can inhibit learning. Batch normalization addresses this problem by scaling the features to be within the same range throughout the model and across different environments and units. In additional to normalizing each dimension to have unit mean and variance, the range of values is often much smaller, typically between 0 and 1.