What is Deep Deterministic Policy Gradient (DDPG)?

DDPG is a reinforcement learning algorithm which concurrently learns a value function and a policy, known as an actor-critic method. Similar to DQN, 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 motivated the same way as DQN: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving:

$$a^*(s) = \arg\max_a Q^*(s,a)$$

DDPG interleaves learning an approximator to with learning an approximator to $Q^*(s,a)$, and it does so in a way which is specifically adapted for environments with continuous action spaces.

DDPG solves the continuous control problems where the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time, $\max_a Q(s,a) \approx Q\big(s,\mu(s)\big)$ can be approximated. For more details, please refer to [3].

Deep Deterministic Policy Gradient (DDPG) by Pytorch for Project 2, Continuous Control:

In this project, I have developed a DDPG code for training two agent with continuous control actions. The solution is based on the DDPG code given in the lecture for the pendulum problem. The documents and codes can be found in my GitHub folder below:

https://github.com/msfallah58/Navigation-project/tree/main/p3_collab-compet

The codes are written using PyCharme IDE and have saved them in three modules namely: main_DDPG, Agent_DDPG and Networks. The main_DDPG.py module receives the input paramaeters and runs episode iterations for training the agent. It also saves the critic and actor networks and plot the results. Agen_DDPG.py module include three classes Agent(), OUNoise and Replay Buffer. networks.py module designs the network architecture for actor and critic functions of DDPG. It includes Actor_Network() and Critic_Network() classes.

Main_DDPG.py module:

First, the necessary libraries are imported to the module. The list of the libraries are as the below:

```
# 1- Start the Environment
from unityagents import UnityEnvironment
import numpy as np
from Agent_DDPG import Agent
import torch
import matplotlib.pyplot as plt
from collections import deque
```

Then the environment is defined. In this work, the unity environment Tennis in which two agents will be trained simultaneously has been used.

```
# Define environment configuration
# TODO: Add the correct path of environment if needed
environment = UnityEnvironment(file_name="/home/saber/deep-reinforcement-
learning/p3_collab-compet/Tennis_Linux"
                                         "/Tennis.x86_64",
no_graphics=False)
brain_name = environment.brain_names[0]
brain = environment.brains[brain_name]

# Get environment information
# Reset the environment
env_info = environment.reset(train_mode=True)[brain_name]
# number of agents
number_of_agents = len(env_info.agents)
states = env_info.vector_observations[0]
state_size = len(states)
action_size = brain.vector_action_space_size
```

train_agent() function is defined to run episodes and collect samples. The function includes two for loops the first one iterates over the number of episodes and the second loop iterates over each episode to collect actions, states, rewards and termination status at each time step. The inputs to the function are env (nvironment), num_agents (the number of agents), n_episodes (the number of episodes) and t_max (the maximum time_steps for each episode)

```
def train_agent(env=environment, num_agents=number_of_agents,
n_episodes=5000, t_max=2000):
    """
    The function trains the network for given number of episodes
    :param env: the environment
    :param agent: the agent
    :param num_agents: number of agents (int)
    :param n_episodes: number of episodes (int)
    :param t_max: Maximum episode time step (int)
    :return:
    """
    scores_window = deque(maxlen=100)
    scores_episode = []

    for episode in range(n_episodes):
        states = env.reset(train_mode=True)[brain_name].vector_observations
# Get initial state
        agent.reset()
        score = np.zeros(num_agents)

        for t_step in range(t_max):
            actions = agent.select_action(states)  # Get the actions
            env_info = env.step(actions)[brain_name]  # Send the actions to
the environment
            next_states = env_info.vector_observations  # Get the next
state
            rewards = env_info.rewards  # Get the reward
            dones = env_info.local_done  # See if episode has finished

            agent.step(states, actions, rewards, next_states, dones)
            score += rewards
            states = next_states

            if np.any(dones):  # if done, break
                break
```

```
        scores_episode.append(np.mean(score))
        scores_window.append(np.mean(score))

        print('\rEpisode: \t{} \tScore: \t{:.2f} \tAverage Score:
\t{:.2f}'.format(episode, np.mean(score),

np.mean(scores_window)), end="")

        if np.mean(scores_window) >= 0.5:
            print(
                '\nEnvironment solved in {:d} episodes!\tAverage Score:
{:.2f}'.format(episode, np.mean(scores_window)))
            torch.save(agent.actor_network_local.state_dict(),
'saved_actor_network_DDPG.pth')
            torch.save(agent.critic_network_local.state_dict(),
'saved_critic_network_DDPG.pth')
            break

    return scores_episode
```

The module runs train_agent() and returns a list which contains the average score for each episode. It closes the environment and plots the average score of episodes.

```
scores = train_agent()
environment.close()

fig_2 = plt.figure(2)
ax = fig_2.add_subplot(111)
plt.plot(np.arange(1, len(scores) + 1), scores)
plt.title('DDPG agent')

plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```

networks.py module:

This module designs the network architectures for actor and critic functions. The module starts with importing the necessary libraries as listed below:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

Function hidden_init(layer) receives the parameter information of hidden layers and initialises the weights. The function return the std for distribution of weights.

```
def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return -lim, lim
```

The class Actor_Network() defines the architecture of actor function for DDGP while the The class Critic_Network() defines the architecture of critic function.

```
class Actor_Network(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=256,
```

```
fc2_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super().__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):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))


class Critic_Network(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=256,
fc2_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super().__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units + action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        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, action):
        """Build a critic (value) network that maps (state, action) pairs -
> Q-values."""
        x = F.relu(self.fc1(state))
        x = torch.cat((x, action), dim=1)
```

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

Agent_DDPG.py module:

This module is the heart of the algorithm and includes three classes: Agent(), OUNoise() and ReplyBuffer. The imported libraries to the module are:

```
import random
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim as optim
import copy
from networks import Actor_Network, Critic_Network
from collections import namedtuple, deque
```

The final chosen hyper parameters are:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64  # mini batch size
GAMMA = 0.99  # discount factor
TAU = 1e-3  # for soft update of target parameters
LR_ACTOR = 1e-4  # learning rate of actor
LR_CRITIC = 1e-4  # learning rate of critic
```

The class Agent() includes __init__ function, step function, select_action function, reset function, learn function and soft_update function and its input arguments are the number of states (state_size), the number of actions (action_size), the number of agents (n_agents) and random seed.

step function:

This function receives state, action, reward, next_state and termination status of agents at each time step and add them to the replay buffer. If the length of reply buffer is larger than the batch size, the algorithm train the networks at each time step.

```
def step(self, state, action, reward, next_state, done):
    """Save experience in replay memory, and use random sample from buffer
to learn."""
    # Save experience / reward
    for i in range(self.n_agents):
        self.memory.add(state[i, :], action[i, :], reward[i], next_state[i,
:], done[i])

    # If enough samples are available in memory, get random subset and
learn
    if len(self.memory) > BATCH_SIZE:
        experiences = self.memory.sample()
        self.learn(experiences, GAMMA)
```

select_action function:

The function receives state at each time step and calculate the action at the given state and add noises to the action to improve exploration. The default status for adding the noise to actions is True. It is noted that the actions are clipped at -1 and 1 after adding the noise to respect the maximum and minimum of action values.

```python
def select_action(self, state, add_noise=True):
    """
    Returns actions for given state as per current policy
    :param state: current state (array_like)
    :param add_noise: add noise to the network (boolean)
    :return: selected actions (int)
    """
    state = torch.from_numpy(state).float().to(device)
    self.actor_network_local.eval()
    with torch.no_grad():
        action = self.actor_network_local(state).cpu().data.numpy()
    self.actor_network_local.train()

    if add_noise:
        action += self.noise.sample()
    return np.clip(action, -1, 1)
```

reset function:
The function resets the noise to the mean value.

```python
def reset(self):
    self.noise.reset()
```

learn function:
The learn receives a batch and discount factor, gamma and calculates, Q target functions and the expected Q functions and form the loss function as the mean square value of error between the Q target and expected Q functions. Then, it updates and optimises the critic function through backpropagation. The function also maximises the return (the objective function) of the actor function. Finally, it updates the target networks using soft update technique.

```python
def learn(self, experiences, gamma):
    """
    Update policy and value parameters using given batch of experience
tuples
    Q_targets = r (reward) + γ (discount factor) *
critic_target(next_state, actor_target(next_state))
    where:
        actor_target(state) --> action
        critic_target (state, action) --> Q-value

    :param experiences: (Tuple[torch.Variable]): tuple of (s, a, r, s',
done)
    :param gamma: discount factor (float)
    :return:
    """

    states, actions, rewards, next_states, dones = experiences

    # --------------------------------- update critic ---------------------
-------------#
    # Get predicted next-state actions and Q values (for next states) from
target models
    next_actions = self.actor_network_target(next_states)
    Q_targets_next = self.critic_network_target(next_states, next_actions)
    # Compute Q targets for current states
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
    # Compute critic loss
    Q_expected = self.critic_network_local(states, actions)
```

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

    # -------------------------------- update actor -----------------------
--------------#
    # Compute actor loss
    actions_pred = self.actor_network_local(states)
    actor_loss = -self.critic_network_local(states, actions_pred).mean()
    # Minimise the loss
    self.actor_optimiser.zero_grad()
    actor_loss.backward()
    #torch.nn.utils.clip_grad_norm_(self.actor_network_local.parameters(),
1)
    self.actor_optimiser.step()

    # ------------------------------ update target networks --------------
--------------#
    self.soft_update(self.critic_network_local, self.critic_network_target,
TAU)
    self.soft_update(self.actor_network_local, self.actor_network_target,
TAU)
```

soft_update function:
The function receives the local and target networks as well as update rate, tau. The functions updates the parameter of target network through copying the parameters of the local network.

```
def soft_update(self, local_model, target_model, tau):
    """
    soft update model parameters
    theta_target = tau*theta_local + (1-tau)*theta_target

    :param local_model: weights will be copied from (Pytorch model)
    :param target_model: weights will be copied to (Pytorch model)
    :param tau: interpolation parameter (float)
    :return:
    """

    for target_param, local_param in zip(target_model.parameters(),
local_model.parameters()):
        target_param.data.copy_(tau * local_param.data + (1.0 - tau) *
target_param.data)
```

OUNoise class:

The class generate noises for the actions. In the DDPG paper, the authors use Ornstein-Uhlenbeck Process to add noise to the action output [1]. The Ornstein-Uhlenbeck Process generates noise that is correlated with the previous noise, as to prevent the noise from cancelling out or “freezing” the overall dynamics [2]

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""
```

```
    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        self.size = size
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * 
np.random.standard_normal(self.size)
        self.state = x + dx
        return self.state
```

ReplayBuffer Class:

Similar to Deep Q learning, DDPG uses a replay buffer to sample experience to update neural network parameters. During each trajectory roll-out, all the experience tuples (state, action, reward, next_state, dones) are stored in a list with fixed size named “replay buffer.” Then, random mini-batches of experience are sampled from the replay buffer when we update the value and policy networks.

```
class ReplayBuffer:
    """ Fixed-size buffer to store experience tuples"""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """
        Initialise a ReplayBuffer object

        :param action_size: dimension of action space (int)
        :param buffer_size: maximum size of buffer (int)
        :param batch_size: size of each training batch (int)
        """

        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", 
"action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory"""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        experiences = random.sample(self.memory, k=self.batch_size)
        states = torch.from_numpy(np.vstack([e.state for e in experiences 
if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences 
if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences
```

```
if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in
experiences if e is not None])).float() \
            .to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if
e is not None]).astype(np.uint8)).float() \
            .to(device)

        return states, actions, rewards, next_states, dones

    def __len__(self):
        """Return the current size of internal memory"""
        return len(self.memory)
```

Performance Analysis and Discussion:
In this project, I investigated different hyperparameters to study the learning performance. The results indicates that the learning is very sensitive to the hyperparameters and a slight change will deteriorate the learning performance significantly. The best performance was obtained for the agent with two hidden layers with 256 nodes each and the batch size of 64. The number of episodes defined as 1000 and the maximum episode time of 2000 s. In this case, it takes more than 1000 episodes for the agent to start learning however the agent starts learning very quickly and reached to the target average score at episode of 2877 (See Fig. 1). The results showed also increasing batch size more than 64 increases the computation load while slowing down the learning speed (for example see Fig. 2). Larger or smaller number of nodes also did not improve the learning speed (see Figs. 3-5). I also clipped the gradients to see if any improvement can be achieved but the results show that clipping gradients does not have significant impact on learning speed (see Figs. 6 and 7).

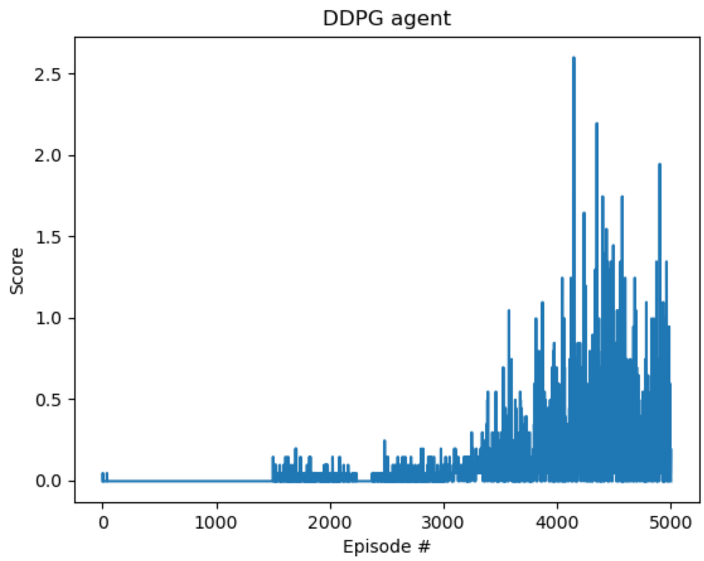

Figure 1 Two hidden layers with 256 nodes each, Batch size: 64, Episode: 2877, Final Score: 1.95, Average Score: 0.51, Environment solved in 2877 episodes!

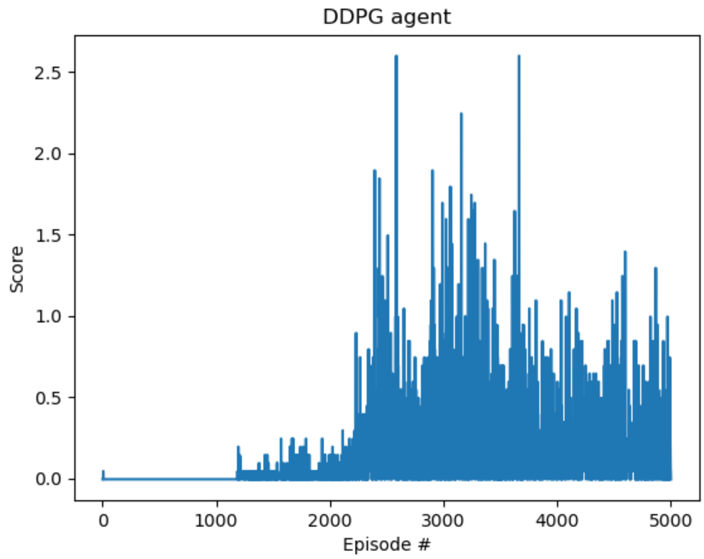

Figure 2 Two hidden layers with 256 nodes each, Batch size: 128, Episode: 4999, Final Score: 1.00, Average Score: 0.42

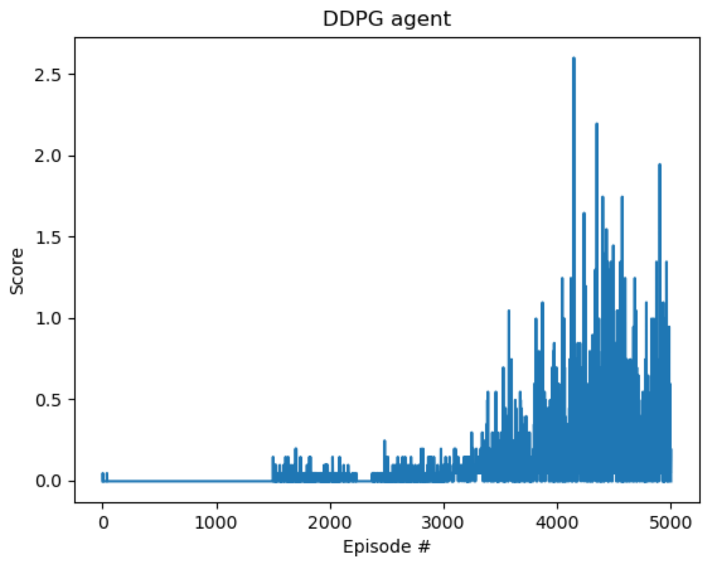

Figure 3 Two hidden layers with 256 and 128 nodes each, Batch size: 64, Episode: 4999, Final Score: -0.00, Average Score: 0.26

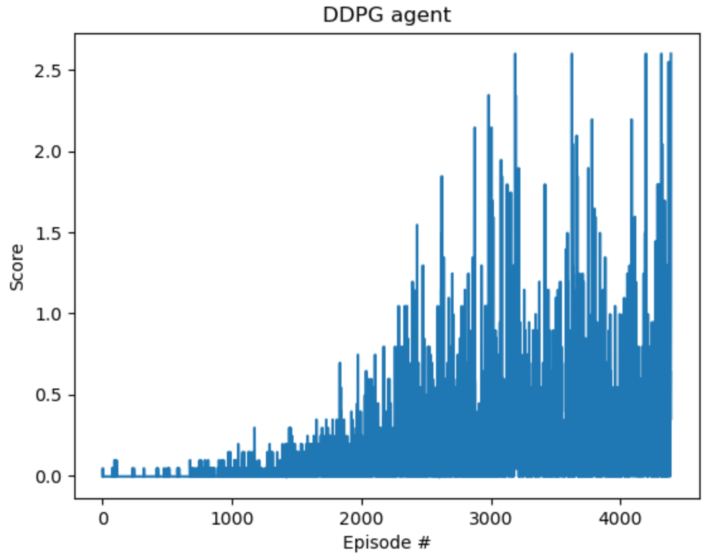

Figure 4 Two hidden layers with 128 nodes each, Batch size: 64, Final Score: 2.60, Average Score: 0.51

Environment solved in 4389 episodes!

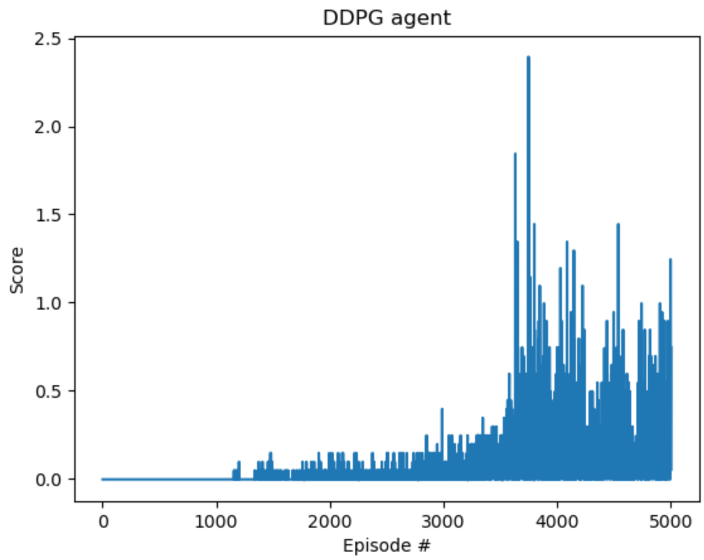

Figure 5 Two hidden layers with 512 nodes each, Batch size: 64, Episode: 4999, Final Score: 1.4, Average Score: 0.26

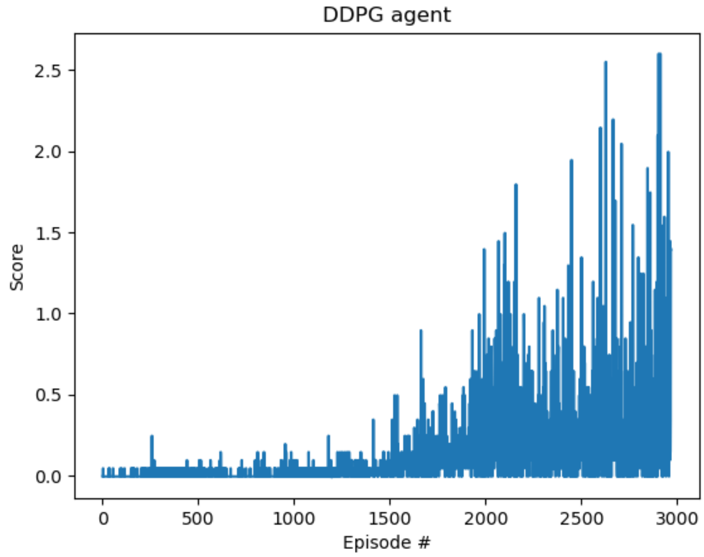

Figure 6: Two hidden layers with 256 nodes each, Batch size: 64, with clipping gradients, Episode: 4999, Final Score: -0.00, Average Score: 0.17

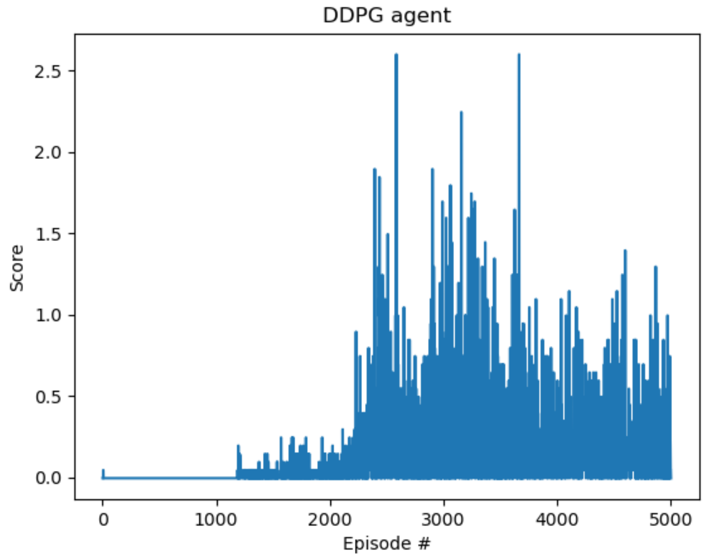

Figure 7 Two hidden layers with 256 nodes each, Batch size: 64, with clipping gradients, Final Score: 1.40, Average Score: 0.50, Environment solved in 2967 episodes!

Future work:

I did not check the effects of noises on exploration and the performance. It is recommended to analysis it as well to see how it will change the exploration and the convergence speed of the algorithm. It is recommended that other RL algorithms such as A2C, A3C or SAC to be investigated.

References:

[1] https://arxiv.org/abs/1509.02971

[2] Maria J. P. Peixoto, Akramul Azim, "Using time-correlated noise to encourage exploration and improve autonomous agents performance in Reinforcement Learning", The 18th International Conference on Mobile Systems and Pervasive Computing (MobiSPC), 2021.

[3] https://spinningup.openai.com/en/latest/algorithms/ddpg.html