ECEN 743: Reinforcement Learning Deep Q-Learning Code tested using 1. gymnasium 0.27.1 2. box2d-py 2.3.5 3. pytorch 2.0.0 4. Python 3.9.12 1 & 2 can be installed using pip install gymnasium[box2d] General Instructions

1. This code consists of TODO blocks, read them carefully and complete each of the blocks
2. Type your code between the following lines

```
###### TYPE YOUR CODE HERE ######
#################################
```

3. The default hyperparameters should be able to solve LunarLander-v2
4. You do not need to modify the rest of the code for this assignment, feel free to do so if needed.

```
!pip install gymnasium[box2d]
!pip3 install matplotlib
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting gymnasium[box2d]
  Downloading gymnasium-0.28.1-py3-none-any.whl (925 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 925.5/925.5 KB 16.1 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.9/dist-packages (from gymnasium[box2d]) (1.22.4)
Collecting jax-jumpy>=1.0.0
  Downloading jax_jumpy-1.0.0-py3-none-any.whl (20 kB)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.9/dist-packages (from gymnasium[box2d]) (2.2.1)
Requirement already satisfied: importlib-metadata>=4.8.0 in /usr/local/lib/python3.9/dist-packages (from gymnasium[box2d]) (6.1.0)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.9/dist-packages (from gymnasium[box2d]) (4.5.0)
Collecting farama-notifications>=0.0.1
  Downloading Farama_Notifications-0.0.4-py3-none-any.whl (2.5 kB)
Collecting pygame==2.1.3
  Downloading pygame-2.1.3-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (13.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.7/13.7 MB 58.8 MB/s eta 0:00:00
Collecting box2d-py==2.3.5
  Downloading box2d-py-2.3.5.tar.gz (374 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 374.4/374.4 KB 14.0 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Collecting swig==4.*
  Downloading swig-4.1.1-py2.py3-none-manylinux_2_5_x86_64.manylinux1_x86_64.whl (1.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 47.9 MB/s eta 0:00:00
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.9/dist-packages (from importlib-metadata>=4.8.0->gymnasium[box2d])
Building wheels for collected packages: box2d-py
  error: subprocess-exited-with-error

  × python setup.py bdist_wheel did not run successfully.
  │ exit code: 1
  ╰─> See above for output.

  note: This error originates from a subprocess, and is likely not a problem with pip.
  Building wheel for box2d-py (setup.py) ... error
  ERROR: Failed building wheel for box2d-py
  Running setup.py clean for box2d-py
Failed to build box2d-py
Installing collected packages: swig, farama-notifications, box2d-py, pygame, jax-jumpy, gymnasium
  Running setup.py install for box2d-py ... done
  DEPRECATION: box2d-py was installed using the legacy 'setup.py install' method, because a wheel could not be built for it. A possib
  Attempting uninstall: pygame
    Found existing installation: pygame 2.3.0
    Uninstalling pygame-2.3.0:
      Successfully uninstalled pygame-2.3.0
Successfully installed box2d-py-2.3.5 farama-notifications-0.0.4 gymnasium-0.28.1 jax-jumpy-1.0.0 pygame-2.1.3 swig-4.1.1
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: matplotlib in /usr/local/lib/python3.9/dist-packages (3.7.1)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (3.0.9)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (23.0)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (4.39.3)
Requirement already satisfied: importlib-resources>=3.2.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (5.12.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (1.0.7)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (0.11.0)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (8.4.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (1.4.4)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (1.22.4)
Requirement already satisfied: zipp>=3.1.0 in /usr/local/lib/python3.9/dist-packages (from importlib-resources>=3.2.0->matplotlib) (3
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
```

```
import gymnasium as gym
import random
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import argparse
import numpy as np
from collections import deque, namedtuple
import matplotlib.pyplot as plt
import base64, io

# For visualization
from gymnasium.wrappers.monitoring import video_recorder
from IPython.display import HTML
from IPython import display
import glob
```

```
class ExperienceReplay:
    """
    Based on the Replay Buffer implementation of TD3
    Reference: https://github.com/sfujim/TD3/blob/master/utils.py
    """
    def __init__(self, state_dim, action_dim,max_size,batch_size,gpu_index=0):
        self.max_size = max_size
        self.ptr = 0
        self.size = 0
        self.state = np.zeros((max_size, state_dim))
        self.action = np.zeros((max_size, action_dim))
        self.next_state = np.zeros((max_size, state_dim))
        self.reward = np.zeros((max_size, 1))
        self.done = np.zeros((max_size, 1))
        self.batch_size = batch_size
        self.device = torch.device('cuda', index=gpu_index) if torch.cuda.is_available() else torch.device('cpu')


    def add(self, state, action,reward,next_state, done):
        self.state[self.ptr] = state
        self.action[self.ptr] = action
        self.next_state[self.ptr] = next_state
        self.reward[self.ptr] = reward
        self.done[self.ptr] = done
        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)

    def sample(self):
        ind = np.random.randint(0, self.size, size=self.batch_size)

        return (
            torch.FloatTensor(self.state[ind]).to(self.device),
            torch.FloatTensor(self.action[ind]).long().to(self.device),
            torch.FloatTensor(self.reward[ind]).to(self.device),
            torch.FloatTensor(self.next_state[ind]).to(self.device),
            torch.FloatTensor(self.done[ind]).to(self.device)
        )
```

```
class QNetwork(nn.Module):
    """
    Q Network: designed to take state as input and give out Q values of actions as output
    """

    def __init__(self, state_dim, action_dim):
        """
            state_dim (int): state dimenssion
            action_dim (int): action dimenssion
        """
        super(QNetwork, self).__init__()
        self.l1 = nn.Linear(state_dim, 64)
        self.l2 = nn.Linear(64, 64)
        self.l3 = nn.Linear(64, action_dim)

    def forward(self, state):
        q = F.relu(self.l1(state))
        q = F.relu(self.l2(q))
        return self.l3(q)
```

```
class DQNAgent():

    def __init__(self,
     state_dim,
     action_dim,
     discount=0.99,
     tau=1e-3,
     lr=5e-4,
     update_freq=4,
     max_size=int(1e5),
     batch_size=64,
     gpu_index=0
     ):
        """
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            discount (float): discount factor
            tau (float): used to update q-target
            lr (float): learning rate
            update_freq (int): update frequency of target network
            max_size (int): experience replay buffer size
            batch_size (int): training batch size
            gpu_index (int): GPU used for training
        """
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.discount = discount
        self.tau = tau
        self.lr = lr
        self.update_freq = update_freq
        self.max_size = max_size
        self.batch_size = batch_size
        self.device = torch.device('cuda', index=gpu_index) if torch.cuda.is_available() else torch.device('cpu')


        # Setting up the NNs
        self.Q = QNetwork(state_dim, action_dim).to(self.device)
        self.Q_target = QNetwork(state_dim, action_dim).to(self.device)
        self.optimizer = optim.Adam(self.Q.parameters(), lr=self.lr)

        # Experience Replay Buffer
        self.memory = ExperienceReplay(state_dim,1,max_size,self.batch_size,gpu_index)

        self.t_train = 0

    def step(self, Exp_rep, targ_net, state, action, reward, next_state, done):
        """
        1. Adds (s,a,r,s') to the experience replay buffer, and updates the networks
        2. Learns when the experience replay buffer has enough samples
        3. Updates target netowork
        """
        self.t_train += 1
        self.memory.add(state, action, reward, next_state, done)


        # Experience Reply and Target Network Enabled
        if Exp_rep and targ_net:

            # Conducting Experience Replay
            if self.memory.size > self.batch_size:
                experiences = self.memory.sample()
                self.learn(experiences, self.discount) #To be implemented

                # Updating Target Network
                if(self.t_train % self.update_freq) == 0:
                    self.target_update(self.Q, self.Q_target, self.tau)

        # ONLY conducting Experience Replay
        elif Exp_rep and not targ_net:

            if self.memory.size > self.batch_size:
                self.memory.add(state, action, reward, next_state, done)
                experiences = self.memory.sample()
                self.learn_exp_rep(experiences, self.discount)
            # Target Network should not be updated

        # ONLY updating Target Network
```

```
        else:

            if self.memory.size > self.batch_size:
                self.learn_targ_net(state, action, reward, next_state, done, self.discount)
                if(self.t_train % self.update_freq) == 0:
                    self.target_update(self.Q, self.Q_target, self.tau)

#To be implemented

    def select_action(self, state, epsilon = 0.):
        if np.random.random() > epsilon:

            state = torch.tensor(state, dtype=torch.float32, device=self.device) # converting our state to pytorch tensor
            #self.Q(state) # writing updated tensor to device
            self.Q.eval()
            with torch.no_grad():
                actions=self.Q.forward(state)
            self.Q.train()
            action = torch.argmax(actions).item() # item() changes from tensor to integer
        else:
                action=np.random.choice(self.action_dim)

        return action

    def learn(self, experiences, discount):
        """
        TODO: Complete this block to update the Q-Network using the target network
        1. Compute target using  self.Q_target ( tar get = r + discount * max_b [Q_target(s,b)] )
        2. Compute Q(s,a) using self.Q
        3. Compute MSE loss between step 1 and step 2
        4. Update your network
        Input: experiences consisting of states,actions,rewards,next_states and discount factor
        Return: None
        """
        states, actions, rewards, next_states, dones = experiences


        q_eval = self.Q(states).gather(1, actions)
        q_next = self.Q_target(next_states).detach().max(1)[0].unsqueeze(1)

        # Calculating q_target
        q_target = rewards + discount * q_next * (1-dones)

        # Calculating loss and backpropogating
        loss = F.mse_loss(q_eval, q_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        self.target_update(self.Q, self.Q_target, self.tau)


    def learn_exp_rep(self, experiences, discount):
        """
        TODO: Complete this block to update the Q-Network using the target network
        1. Compute target using  self.Q_target ( tar get = r + discount * max_b [Q_target(s,b)] )
        2. Compute Q(s,a) using self.Q
        3. Compute MSE loss between step 1 and step 2
        4. Update your network
        Input: experiences consisting of states,actions,rewards,next_states and discount factor
        Return: None
        """
        states, actions, rewards, next_states, dones = experiences

        q_eval = self.Q(states).gather(1, actions)
        q_next = self.Q_target(next_states).detach().max(1)[0].unsqueeze(1)
        q_target = rewards + discount * q_next * (1-dones)
        loss = F.mse_loss(q_eval, q_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        #self.target_update(self.Q, self.Q_target, self.tau)

    def learn_targ_net(self, states, actions, rewards, next_states, dones, discount):
        """
        TODO: Complete this block to update the Q-Network using the target network
        1. Compute target using  self.Q_target ( tar get = r + discount * max_b [Q_target(s,b)] )
        2. Compute Q(s,a) using self.Q
```

```
        3. Compute MSE loss between step 1 and step 2
        4. Update your network
        Input: experiences consisting of states,actions,rewards,next_states and discount factor
        Return: None
        """
        # Storing states

        states, actions, rewards, next_states, dones = [states], [actions], [rewards], [next_states], [dones]
        states = torch.tensor(states).float().to(self.device)
        actions = torch.tensor(actions).long().unsqueeze(1).to(self.device)  # Fix here
        rewards = torch.tensor(rewards).float().unsqueeze(1).to(self.device)  # Fix here
        next_states = torch.tensor(next_states).float().to(self.device)
        dones = torch.tensor(dones).unsqueeze(1).float().to(self.device)

        # Calculating q_target
        q_eval = self.Q(states).gather(1, actions)
        q_next = self.Q_target(next_states).detach().max(1)[0].unsqueeze(1)

        # Calculating loss and backpropogating
        q_target = rewards + discount * q_next * (1-dones)
        loss = F.mse_loss(q_eval, q_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        self.target_update(self.Q, self.Q_target, self.tau)

    def target_update(self, Q, Q_target, tau):
        """
        TODO: Update the target network parameters (param_target) using current Q parameters (param_Q)
        Perform the update using tau, this ensures that we do not change the target network drastically
        1. param_target = tau * param_Q + (1 - tau) * param_target
        Input: Q,Q_target,tau
        Return: None
        """
        ###### TYPE YOUR CODE HERE ######
        for param_target, param_local in zip(Q_target.parameters(), Q.parameters()):
            param_target.data.copy_(tau*param_local.data + (1.0-tau)*param_target.data)


        #param_target = tau * param_Q + (1 - tau) * param_target
```

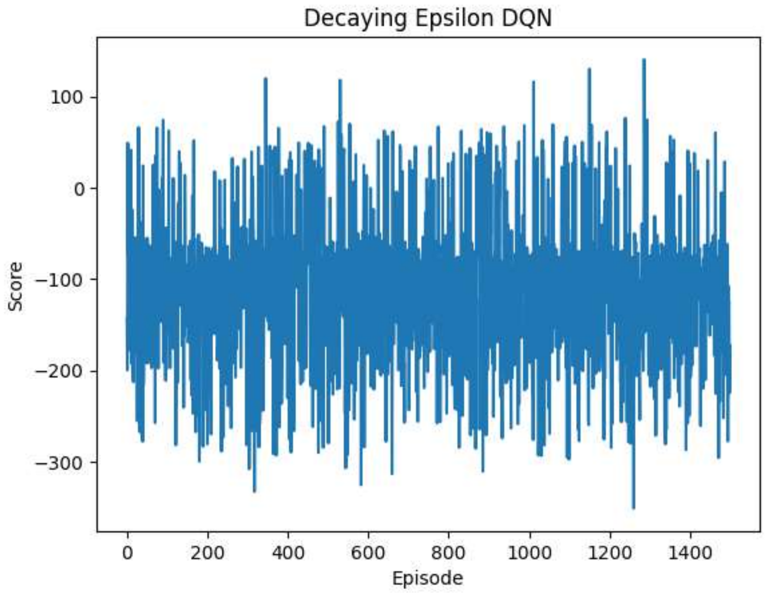
Now I'm comparing only target network and Experience Replay. \

This is the code for ONLY Experience Replay

```
def exp_rep_only():
    seed = 0
    n_episodes = 1500
    batch_size = 64
    discount = 0.99
    lr = 5e-4                       # learning rate
    tau = 0.001                     # soft update of target network
    max_size = int(1e5)
    update_freq = 4
    gpu_index = 0
    max_eps_len = 1000
    #exploration strategy

    epsilon = 1

    # making the environment
    env = gym.make("LunarLander-v2")

    #setting seeds
    torch.manual_seed(seed)
    np.random.seed(seed)
    random.seed(seed)

    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.n

    kwargs = {
        "state_dim":state_dim,
        "action_dim":action_dim,
        "discount":discount,
        "tau":tau,
```

```
        "lr":lr,
        "update_freq":update_freq,

        "max_size":max_size,
        "batch_size":batch_size,
        "gpu_index":gpu_index
    }
    exp_rep_only_learner = DQNAgent(**kwargs) #Creating the DQN learning agent
    moving_window = deque(maxlen=100)
    reward_store = []
    print("Experience Replay Only")


    for e in range(n_episodes):
        state, _ = env.reset(seed=seed)
        curr_reward = 0
        for t in range(max_eps_len):
            action = exp_rep_only_learner.select_action(state, epsilon) #To be implemented
            n_state,reward,terminated,truncated,_ = env.step(action)
            done = terminated or truncated
            exp_rep_only_learner.step(True, False, state, action, reward, n_state, done) #To be implemented
            state = n_state
            curr_reward += reward
            if done:
                break
        moving_window.append(curr_reward)
        reward_store.append(curr_reward)


        """"
        TODO: Write code for decaying the exploration rate using args.epsilon_decay
        and args.epsilon_end. Note that epsilon has been initialized to args.epsilon_start
        1. You are encouraged to try new methods
        """
        ###### TYPE YOUR CODE HERE ######
        #################################
    plt.plot(np.arange(len(reward_store)), reward_store)
    plt.ylabel('Score')
    plt.xlabel('Episode')
    plt.title('Decaying Epsilon DQN')
    plt.show()

    if e % 100 == 0:
            print('Episode Number {} Average Episodic Reward (over 100 episodes): {:.2f}'.format(e, np.mean(moving_window)))


        ###### TYPE YOUR CODE HERE ######
        #################################
    averages = []
    window_size=50

    for i in range(len(reward_store) - window_size + 1):
        window = reward_store[i:i+window_size]
        average = sum(window) / window_size
        averages.append(average)

    plt.plot(averages, color='g')
    plt.ylabel('Score')
    plt.xlabel('Episodes')
    plt.title('Only Experience Replay DQN')
    plt.show()

exp_rep_only()
```

```
Experience Replay Only
```

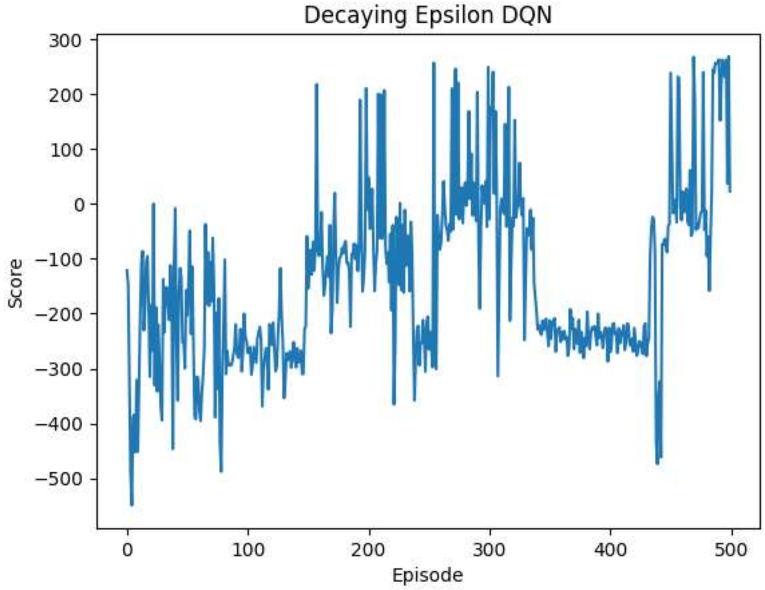


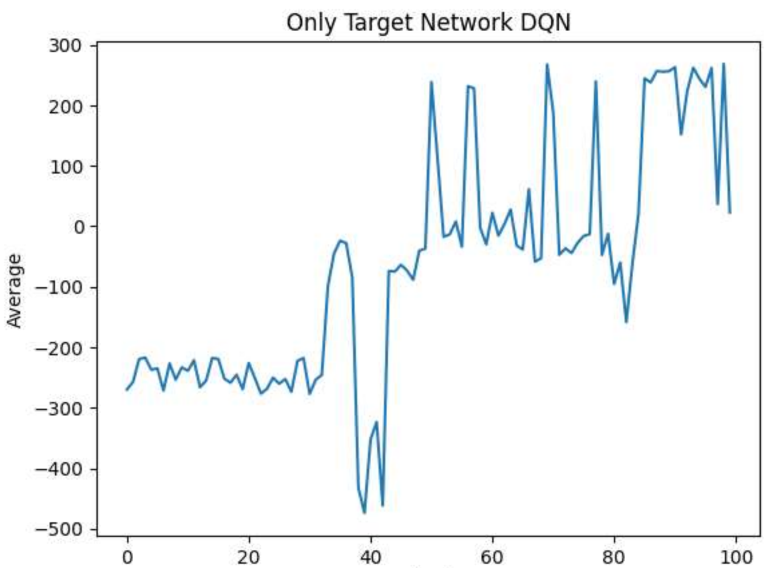
This is the code for only Target Network

```
def only_tar_net_trainer():
  seed = 0
  n_episodes = 500
  batch_size = 64
  discount = 0.99
  lr = 5e-4                     # learning rate
  tau = 0.001                   # soft update of target network
  max_size = int(1e5)
  update_freq = 4
  gpu_index = 0
  max_eps_len = 1000

  #exploration strategy

  # making the environment
  env = gym.make("LunarLander-v2")
  #setting seeds
  torch.manual_seed(seed)
  np.random.seed(seed)
  random.seed(seed)
  state_dim = env.observation_space.shape[0]
  action_dim = env.action_space.n
  kwargs = {
    "state_dim":state_dim,
    "action_dim":action_dim,
    "discount":discount,
    "tau":tau,
    "lr":lr,
    "update_freq":update_freq,
    "max_size":max_size,
    "batch_size":batch_size,
    "gpu_index":gpu_index
  }
  disabled_decaying_learner = DQNAgent(**kwargs) #Creating the DQN learning agent
  moving_window = deque(maxlen=100)

  reward_store = []

```

```
  print("Target Network DQN")

  epsilon_start = 1               # start value of epsilon
  epsilon_end = 0.01              # end value of epsilon
  epsilon_decay = 0.995           # decay value of epsilon

  epsilon_by_step = lambda step: float(epsilon_end+(epsilon_start - epsilon_end)*np.exp(-1. * step / epsilon_decay))


  for e in range(n_episodes):
    state, _ = env.reset(seed=seed)
    curr_reward = 0
    for t in range(max_eps_len):
      action = disabled_decaying_learner.select_action(state, epsilon_by_step(t)) #To be implemented
      n_state,reward,terminated,truncated,_ = env.step(action)
      done = terminated or truncated
      disabled_decaying_learner.step(False, True, state, action, reward, n_state, done) #To be implemented
      state = n_state
      curr_reward += reward
      if done:
        break
    reward_store.append(curr_reward)
    moving_window.append(curr_reward)

    """"
      TODO: Write code for decaying the exploration rate using args.epsilon_decay
      and args.epsilon_end. Note that epsilon has been initialized to args.epsilon_start
      1. You are encouraged to try new methods
    """
    ###### TYPE YOUR CODE HERE ######
    #################################

    if e % 100 == 0:
      print('Episode Number {} Average Episodic Reward (over 100 episodes): {:.2f}'.format(e, np.mean(moving_window)))

    """"
      TODO: Write code for
      1. Logging and plotting
      2. Rendering the trained agent
    """
    ###### TYPE YOUR CODE HERE ######
    #################################
  plt.plot(np.arange(len(reward_store)), reward_store)
  plt.ylabel('Score')
  plt.xlabel('Episode')
  plt.title('Decaying Epsilon DQN')
  plt.show()

  plt.plot(np.arange(len(moving_window)), moving_window)
  plt.ylabel('Average')
  plt.xlabel('Episodes')
  plt.title('Only Target Network DQN')
  plt.show()

only_tar_net_trainer()
```

```
Target Network DQN
Episode Number 0 Average Episodic Reward (over 100 episodes): -121.30
Episode Number 100 Average Episodic Reward (over 100 episodes): -246.68
Episode Number 200 Average Episodic Reward (over 100 episodes): -170.87
Episode Number 300 Average Episodic Reward (over 100 episodes): -64.01
Episode Number 400 Average Episodic Reward (over 100 episodes): -152.57
```





Colab paid products - Cancel contracts here

6m 12s completed at 11:03 PM