به نام خدا

مینی پروژه چهارم یادگیری ماشین

استاد: دکتر مهدی علیاری

محمدقاسم امامی مقدم

شماره دانشجویی: ۴۰۲۰۲۳۱۴

پوشه گیت هاب مینی پروژه - پوشه گوگل کولب مینی پروژه

اردیبهشت ۱۴۰۳

فهرست مطالب

۱ سوال اول ۱
۱.۱ بخش اول ۱
۲.۱ بخش دوم ۲
۳.۱ بخش سوم ۵

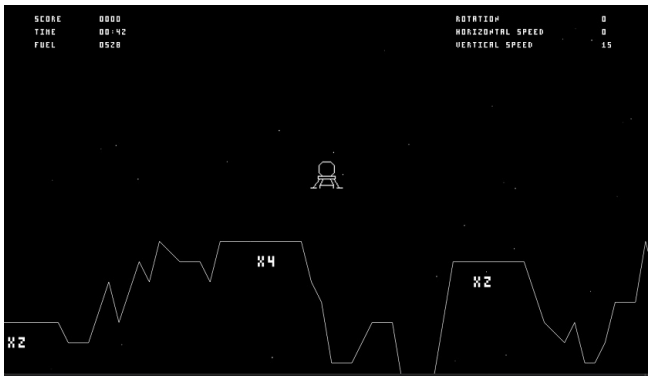
فهرست تصاویر

۱ صورت کلی بازی موجود در اینترنت ۱
۲ مقایسه دو روش ۲
۳ DDQN ۵
۴ دسته های ۳۲ عددی ۵
۵ دسته های ۶۴ عددی ۶
۶ دسته های ۱۲۸ عددی ۶

۱ سوال اول

۱.۱ بخش اول

هدف باری Lunar Lander هدایت یک سفینه میباشد یه طوری که باید در محدوده مورد نظر بنشیند. در واقع باید از محل شروع بازی و رهاسازی سفینه با مولفه های موجود و در دسترس سفینه را در حالت بهینه کنترل کرد و در مقصد نشاند.



شکل ۱: صورت کلی بازی موجود در اینترنت

متغیر های حالت مربوط به سفینه

۱. موقعیت عمودی x
۲. موقعیت افقی y
۳. سرعت عمودی v_y
۴. سرعت افقی v_x
۵. زاویه θ
۶. سرعت زاویه ای w
۷. مقادیر بولین $\leftarrow$

- تماس پایه چپ با سطح left leg contanct
- تماس پایه راست با سطح right leg contant

دقت شود که مقادیر مربوط به موقعیت های سفینه به صورت نسبی در نظر گرفته میشوند بدین صورت که محل فرود را نقطه (۰و۰) در نظر میگیریم و موقعیت عمودی و افقی را نسبت به ان میسنجیم. در این سیستم نیز ورودی های مربوط به سفینه قابل کنترل توسط ما هستند. در واقع با این مولفه ها میتوان تمامی متغیر های حالت مربوط به مسله را کنترل نمود. در هر لحظه موقعیت سفینه برای ما مدنظر می باشد. فضای اکشن مربوط یه سفینه:

۱. هیچ کار انجام ندادن
۲. روشن کردن موتور اصلی
۳. روشن کردن موتور سمت راستی
۴. روشن کردن موتور سمت چپی

برای این سوال می بایست یک سیستم امتیاز دهی در نظر گرفته شود تا بتوان عملکرد سیستم را بهبود بخشید و هر مرحله نسبت به مرحله قبلی عملکرد بهتری داشته باشد. در ابتدا برای عملکرد های مثبت و مثمر ثمر امتیازات بالا و مثبت در نظر گرفته میشود.

۱. ساکن شدن سفینه :۱۰۰ امتیاز مثبت
۲. تماس هر پابه با زمین : ۱۰ امتیاز مثبت

۳. حرکت به سمت محدوده مطبوب : ۱۰۰ تا ۱۴۰ امتیاز مثبت

۴. حل سوال و جاگیری در محدوده مطلوب : ۲۰۰ امتیاز مثبت

در طرف مقابل امتیازات منفی نیز برای سیستم در نظر گرفته میشود:

۱. روشن کردن موتور اصلی(بالابرنده) : ۳.۰ امتیاز منفی

۲. روشن کردن موتور های جانبی : ۰۳.۰ امتیاز منفی

۳. برخورد با زمین در اثر سقوط یا در جهت اشتباه (عدم قود روی پابه های سفینه) : ۱۰۰ امتیاز منفی

حال با توجه به شرایط داده شده و امتیازات اختصاص داده شده به سیستم در ابتدا سفینه از یک نقطه اولیه قرایند اغار می شود. در ایتدا شرایط اولیه وارده یه سیستم به صورت رندوم در نظر گرفته میشود .تنها نیروی وارده به سیستم نیروی جاذبه ی وارده یه سیستم میباشد . فرایند امتیاز دهی و سوال در چند حالت به اتمام میرسد.

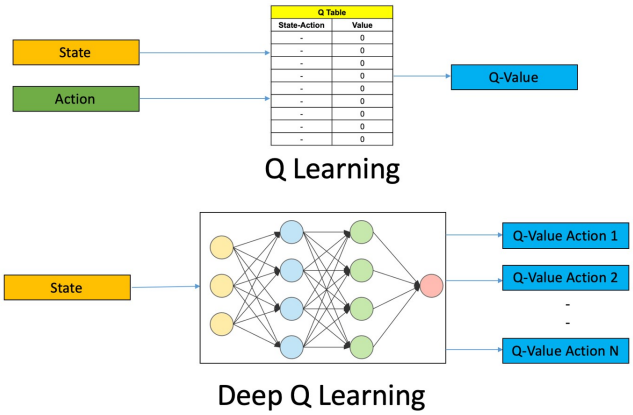
۱. سقوط کردن سفینه و برخورد با سطح

۲. خارج شدن سفینه از محدوده سوال

۳. رسیدن به حالت نهایی

۲.۱ بخش دوم

دقت شود در مورد سوالاتس که فضای سوال مربوطه کوچک باشد و خیلی بزرگ نباشد از الگوریتم Q-Learning استفاده میکینیم و همان طور که در کلاس مورد بررسی قرار گرفت ماتریس مربوط به Q بعد از تعداد مشخصی از حرکات اپدیت شده و ورن دهی میکند تا درنهابت بعد از تعدادی تکرار مقادیر مربوط به هر درابه به مقداری همگرا می شوند. در الگوریتم Deep-Q-Learning در حالاتی که فضای سوال بزرگ باشد و اپدیت کردن ماتریس Q کاری بسار دشوار خواهد بود. در این سوال نیز فضای سیستم بسیار گسترده است و تعداد حالات زیادی باید مورد بررسی قرار گیرد به همین دلیل از این روش استفاده می شود. در این قسمت به بررسی قسمت هایی از کد که مربوط یه این متد می باشد میپردازیم:



شکل ۲: مقایسه دو روش

کلاس ExperienceRelay تعریف شده برای ذخیره سازی ۴ متغیر می باسد که این مقادبر در هر اپیروز تکرار میشود . این متغیر ها برابر هستند با :

۱. حالت کنونی

۲. لحظه بعد

۳. اکشن های موجود در این حالت

۴. پاداش هر حرکت

این مقادیر به صورت یک رشته touple در کنار یکدیگر قرار میگیرند و برای هر خانه و هر state این مقادیر به دست می ایند.

$$Experience_i = (state_i, action_i, state_{i+1}, reward)$$

و همان طوز که ذکر شد این مقادیر ردر کنار یکدیگر قرار میگیرند تا برای تمامی خانه های سوال این touple ها شکل گیرند.

$$Total_{experince} = [e_1, e_2, e_3, ..., e_n]$$

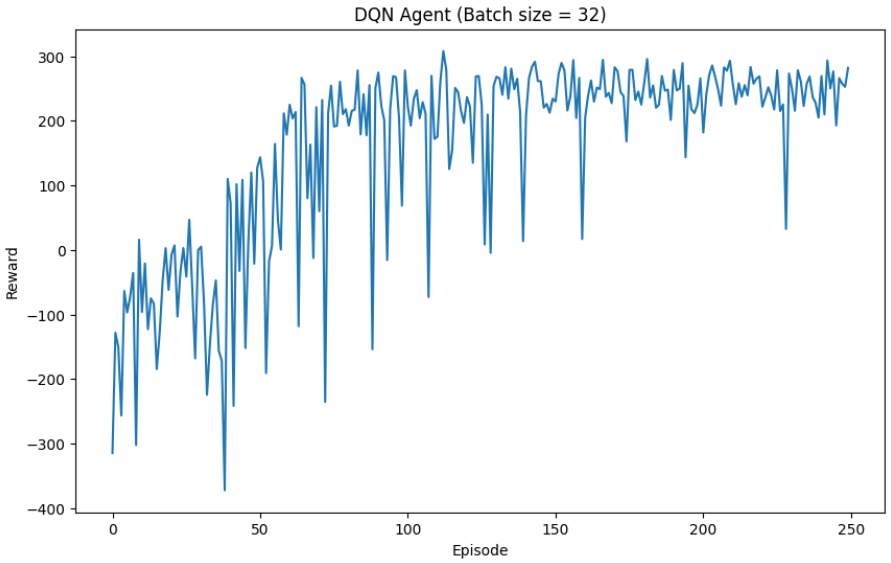
حال در قسمت اموزش mini-batch هایی را به صورت رندوم انتخاب میکنیم.این نمونه ها به صورت تجربه هایی در محیط هستند.
در این کلاس سه متد وجود دارد.

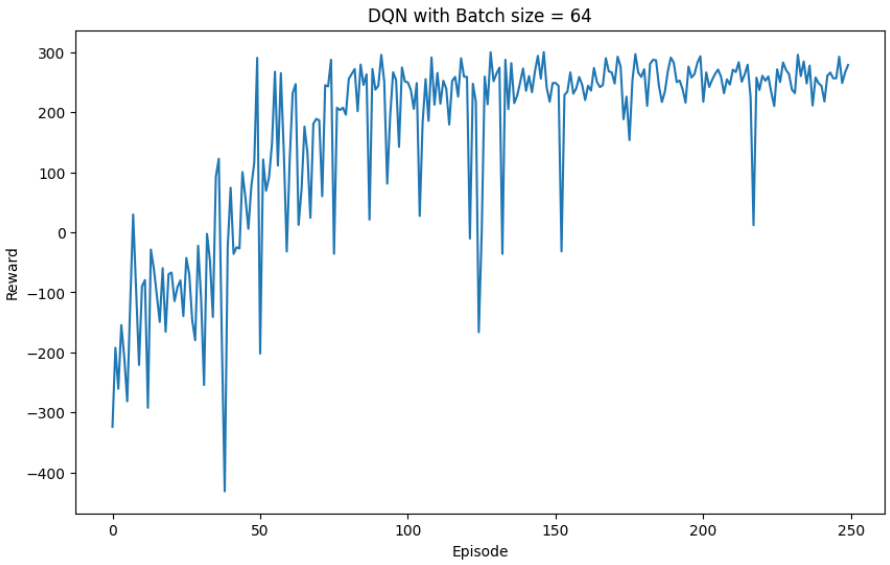
۱. init : در این روش از یک DEQ(Double Ended Queue) استفاده میکنیم برای ذخیره تجربیات

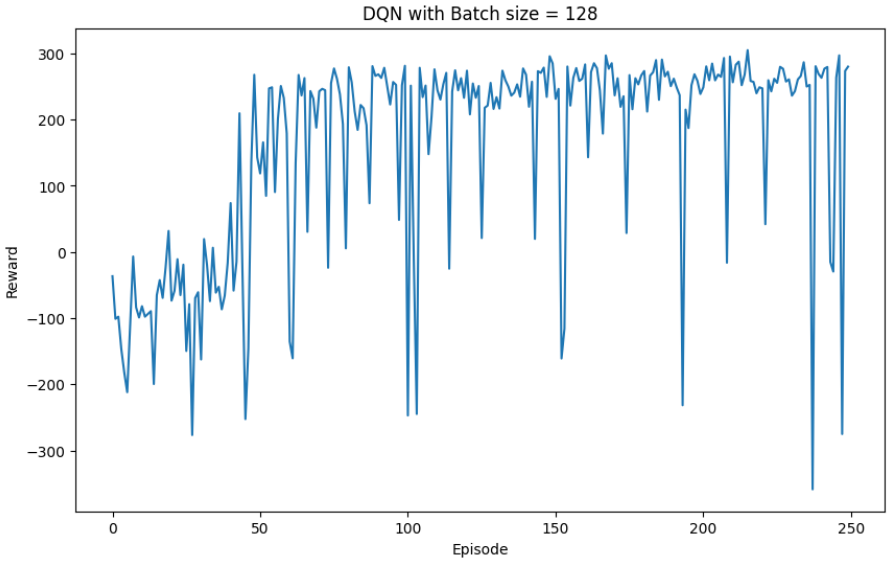
۲. store-trans : تنها برای ذخیره سازی اطلاعات در حافظه

۳. Sample : نمونه برداری از حافظه به صورت تصادفی به تعداد Batch Size

کلاس DeepQNetwork مربوط به ساختار شبکه عصبی است. در این شبکه تعداد لایه ورودب برابر با ۸ می باسد که برابر با اندازه فضای حالت می باشد.و از طرفی هم تعداد لابه های خروجی برابر با ۴ که هماتد تعداد اکشن است می باشد.تعداد نورون های هر لایه برابر با ۵۱۲ تا است.در تمامی لایه های میانی(پنهان) خروجی تمامی لایه ها نرمالایز می شوند با استفاده از Layernorm و نیز تمامی تابع فعال ساز این لایه ها ReLU می باشد. برای جلوگیری از overfitting از روش drop out استفاده می شود.
کلاس DQNAgent : در این کلاس ابتدا هایپر پارامتر ها،ساختار شبکه و نوع بهینه ساز با استفاده از متد init تعیین می شوند. سپس بارای اینکه بازیکن بتواند اقدام بعدی را انتخاب کند از استراتژی epsilon greedy استفاده میکنیم . تمام مراحل مربوط به انتخاب اکشن بعدی در take action انتخاب می شوند.در قسمت update-params همان طور که از اسم این بخش مشخص است برای اپدیت کردن پارامتر ها در هر مرحله استعاده میشود سپس این اظلاعات به روز رسیانی شده را با استعاده از load , save ذخیره میکنیم.
حال در بخش اموزش میبایست مراحل زیر انجام شود. ابتدا می بایست مقادیر پارامتر های اموزش تعیین شوند و سپس فرایند اصلی اغاز میشود. یعنی با توجه به موقعیتی که سفینه در ان قرار دارد یک اکشن انتخاب میشود و در قسمت Experience دخیره میشود سپس با توجه با اقدام انتخاب شده وانجام شده تجربیات مربوط به ان ذخیره میشود .در ضمن در ایتدا مقدار epsilon زیاد می باشد در واقع در ایتدا سیستم در حالت جستجئ در کل قسمت جواب می باشد و سپس با انجام دادن افداماتی به تدریج این مقدار کاهش میابد تا به مسیر بهینه دست بیابیم. در واقع یک تعادل بین جستجو در کل مسیر ها و مسیر بهینه میباشد.
در ادامه پاداش تجمعی برای Batch size های مختلف ترسیم شده است:







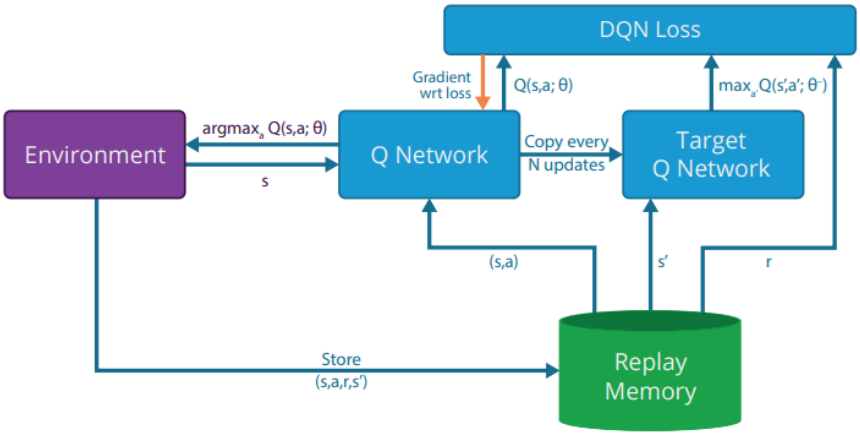
در هر سه نمودار روند مشابهی قابل رویت است . در واقع با افزایش اپیزود ها مقدار پاداش افزایش میابد. در واقع این به معنای بهبود عملکرد سیستم می باش پرا که از انجام اکشن هایی امتیاز منفی دارند امتناع کرده است. حال باید بهترین batch size را انتخاب کنیم از روی روند کلی و شمای ان نمیتون به این امر دست یافت . پس در چندین اپیزئد پابانب به مقایسه می پردازیم . دیده میشود که در حالت ۱۲۸ در حالات پایانی مقدار نوسانات به تسبت بقیه بیشتر می باشد و نیز مقدار ان تیز برابر با −۳۰۰ میباشد که گویا بعد از تکرار های مکرر عملکرد خوبی از خود نشان تداده است. حال به سراغ ۶۴ میرویم . برای این مقدار نیز دز اپیزود های پایانی نوسان داریم ولی مقدار انها کمتر و شدتشان بیشتر میباشد. در حالت ۳۲ تمامی این موارد بهبود یافته است طوری که شدت و مقدار توسانات به شدت کاهش یافته است. پس در نهابت مقدار ۳۲ انتخاب میشود. دقت شود معیار پشیمانی برابر با اختلاف میان حرکت در حال انجام با بهترین حالت ممکنه می باشد. در واقع در این سوال بهترین مقدار برابر با امتیاز ۳۰۰ میباشد و هر چه قدر این اختلاف بیشتر باشد و منفی تر میزان پشیمانی بیشتر است. در مورد نمودار های پیوسته این مقدار برابر با مساحت بالای نمودار تا خط امتیاز ۳۰۰ می باشد. که با توجه به این توضیحات ابن مساحت برای حالت ۶۴ در کمترین حالت خود قرار دارد و معیار پشیمانی برای ۶۴ کمتر است. برای مشاهده فیلم عملکرد عامل در اپیزودهای مختلف به ازای دسته ۳۲ نمونه ای .

شماره اپیزود	لینک
۵۰	لینک
۱۰۰	لینک
۱۵۰	لینک
۲۰۰	لینک
۲۵۰	لینک

جدول ۱: فیلم های عملکرد عامل در روش DQN

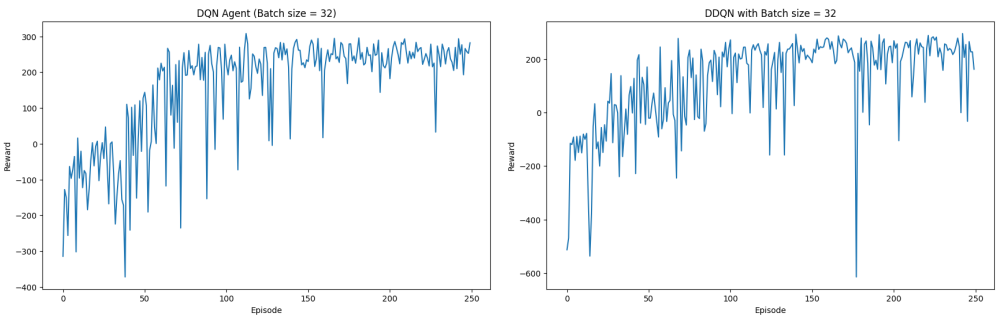
۳.۱ بخش سوم

در ابتدای سوال در مورد DQN , DDQN صحبت شد حال به بررسی بشتر تفاوت این دو میپردازیم. در روش DDQN از دو شبکه عصبی استفاده است به نام های target network و evaluate network . شبکه دوم اضافه شده جهت افزایش دقت در محاسبه ماتریس Q این امر سبب افزایش سرعت همگرایی سیستم و پایداری بیشتر ان میشود.در ضمن در این روش حجم محاسبات و پردازش بیشتر می باشد و از طرفی نیز احتمال رسیدن به بهینه ترین حالت بیشتر است زیرا که تمامی موارد مورد بررسی قرار میگیرند.

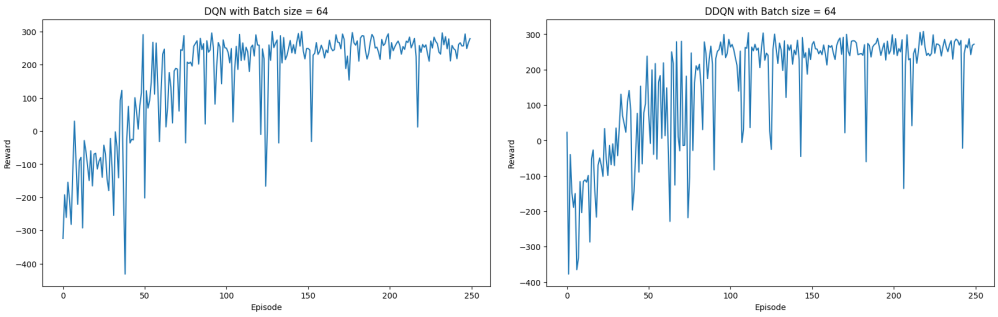


شکل ۳: ساختار DDQN

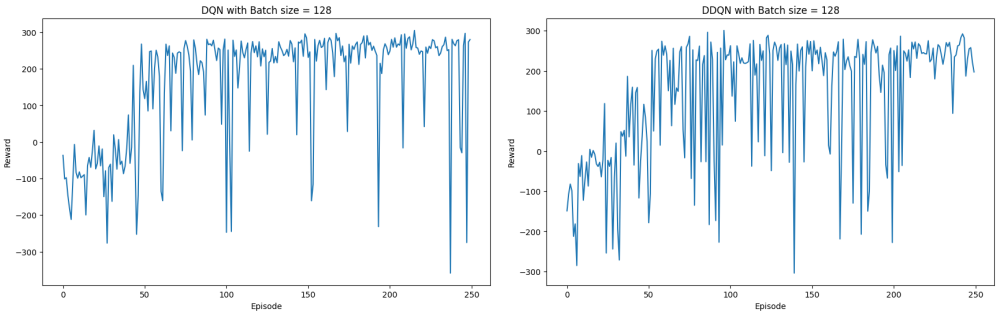
حال به بررسی کد داده شده میپردازیم.
بسیاری از قسمت های کد همانند حالت DQN می باشد. دقت شود در کلاس DDQNAgent تفاوت هایی دارد نظیر اینکه دو شبکه با نام های value net , target net قرار دارند که ساختار کلی انها همانند یکدیگر است. در ضمن تابع هزینه استفاده شده در این قسمت MSE حداقل مربعات است. حال نمودار های مربوط به عملکرد های مختلف را رسم میکینیم.



شکل ۴: دسته های ۳۲ عددی



شکل ۵: دسته های ۶۴ عددی



شکل ۶: دسته های ۱۲۸ عددی

با مقایسه نمودار ها دیده میشود که نمودار مربوط به تمامی مقداری عملکردشان بهبود یافته و شدت و میزان نوسانات در پایان اپیزود ها کاهش یافته استو البه دقت شود در این حالت بهترین مقدار برای ۱۲۸ می باشد چرا که کمترین مقدار نوسانات و کمترین شدت نوسانات را دارد..
فیلم مربوط به عملکرد سیستم در اپیزودهای ۱۰۰و۲۵۰ به ازای الگوریتم های مختلف و batch size های مختلف

الگوریتم	شماره اپیزود	Mini Batch	لینک
DDQN	۱۰۰	۳۲	لینک
DQN	۱۰۰	۳۲	لینک
DDQN	۲۵۰	۳۲	لینک
DQN	۲۵۰	۳۲	لینک
DDQN	۱۰۰	۶۴	لینک
DQN	۱۰۰	۶۴	لینک
DDQN	۲۵۰	۶۴	لینک
DQN	۲۵۰	۶۴	لینک
DDQN	۱۰۰	۱۲۸	لینک
DQN	۱۰۰	۱۲۸	لینک
DDQN	۲۵۰	۱۲۸	لینک
DQN	۲۵۰	۱۲۸	لینک

ضمیمه

کدهای سوال ۲

```
# Install packages
!apt-get update
!apt-get install -y swig
!sudo apt-get update
!sudo apt-get install xvfb
!pip install rarfile --quiet
!pip install stable-baselines3 > /dev/null
!pip install box2d-py > /dev/null
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!sudo apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
import gym
import io
import os
import glob
import torch
import base64
import numpy as np
import copy
import matplotlib.pyplot as plt
from stable_baselines3 import DQN
from stable_baselines3.common.results_plotter import ts2xy, load_results
from stable_baselines3.common.callbacks import EvalCallback
from gym.wrappers import RecordVideo
from IPython.display import HTML
from IPython import display as ipythondisplay
from pyvirtualdisplay import Display
import random
from collections import namedtuple, deque
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# Creat envoiroment
env = gym.make("LunarLander-v2", new_step_api=True)

env.reset()

state_size = env.observation_space.shape[0]
action_size = env.action_space.n

# Introduction to state and action space
print(f'size state space : {state_size}')
print(f'size of action space: {action_size}')

print(f'Sample state space: {env.observation_space.sample()}')
print(f'Sample action space: {env.action_space.sample()}')

# Set up virtual display
display = Display(visible=0, size=(1400, 900))
display.start()

# Utility function to enable video recording of gym environment and displaying
    it
def show_video():
    mp4list = glob.glob('video/*.mp4')
    if len(mp4list) > 0:
```

```
        mp4 = mp4list[0]
        video = io.open(mp4, 'r+b').read()
        encoded = base64.b64encode(video)
        ipythondisplay.display(HTML(data='''<video alt="test" autoplay loop
    controls style="height: 400px;">
                    <source src="data:video/mp4;base64,{0}" type="video/mp4" />
                </video>'''.format(encoded.decode('ascii'))))
    else:
        print("Could not find video")

        # Set random seed
SEED = 14
random.seed(SEED)
torch.manual_seed(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device

# Experience replay
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward', 'done'))

class ExperienceReplay():
    def __init__(self, capacity) -> None:
        self.memory = deque([], maxlen=capacity)

    def store_trans(self, s, a, sp, r, done):
        transition = Transition(s, a, sp, r, done)
        self.memory.append(transition)

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

# Deep Q-Network Structure

class DeepQNetwork(nn.Module):
    def __init__(self, state_size, action_size) -> None:
        super(DeepQNetwork, self).__init__()
        net_list = nn.ModuleList([
            torch.nn.Linear(state_size, 512),
            torch.nn.ReLU(),
            torch.nn.LayerNorm(512),
            torch.nn.Dropout(0.1),
            torch.nn.Linear(512, 512),
            torch.nn.ReLU(),
            torch.nn.LayerNorm(512),
            torch.nn.Dropout(0.1),
            torch.nn.Linear(512, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, action_size)
        ])
        self.net = torch.nn.Sequential(*net_list).to(device)

    def forward(self, x):
        x = x.to(device)
        x = self.net(x)
        return x
```

```
115
116 # DQN agent
117
118 class DQNAgent():
119     def __init__(self, state_size, action_size, batch_size,
120                  gamma=0.99, buffer_size=25000, alpha=1e-4):
121         self.state_size = state_size
122         self.action_size = action_size
123         self.batch_size = batch_size
124         self.gamma = gamma
125         self.experience_replay = ExperienceReplay(buffer_size)
126         self.value_net = DeepQNetwork(state_size, action_size).to(device)
127         self.optimizer = optim.Adam(self.value_net.parameters(), lr=alpha)
128
129     def take_action(self, state, eps=0.0):
130         self.value_net.eval()
131         if random.random() > eps:
132             with torch.no_grad():
133                 state_tensor = torch.tensor(state, dtype=torch.float32).
    unsqueeze(0).to(device)
134                 action_values = self.value_net(state_tensor)
135                 return torch.argmax(action_values).item()
136         else:
137             return np.random.randint(0, self.action_size)
138
139     def update_params(self):
140         if len(self.experience_replay) < self.batch_size:
141             return
142         batch = Transition(*zip(*self.experience_replay.sample(self.batch_size))
    )
143         state_batch = torch.tensor(batch.state, dtype=torch.float32).to(device)
144         action_batch = torch.tensor(batch.action).unsqueeze(1).to(device)
145         next_state_batch = torch.tensor(batch.next_state, dtype=torch.float32).
    to(device)
146         reward_batch = torch.tensor(batch.reward, dtype=torch.float32).unsqueeze
    (1).to(device)
147         done_batch = torch.tensor(batch.done, dtype=torch.float32).unsqueeze(1).
    to(device)
148
149         self.value_net.train()
150         q_expected = self.value_net(state_batch).gather(1, action_batch)
151         q_targets_next = self.value_net(next_state_batch).detach().max(1)[0].
    unsqueeze(1)
152         q_targets = reward_batch + (self.gamma * q_targets_next * (1 -
    done_batch))
153         loss = nn.functional.mse_loss(q_expected, q_targets)
154
155         self.optimizer.zero_grad()
156         loss.backward()
157         self.optimizer.step()
158
159     def save(self, fname):
160         torch.save(self.value_net.state_dict(), fname)
161
162     def load(self, fname, device):
163         self.value_net.load_state_dict(torch.load(fname, map_location=device))
164 n_episodes = 250
165 eps = 1.0
166 eps_decay_rate = 0.97
167 eps_end = 0.01
```

```
BATCH_SIZE = 128



# TODO: create agent
agent = DQNAgent(state_size, action_size, batch_size=BATCH_SIZE)

crs = np.zeros(n_episodes) # cummulative rewards
crs_recent = deque(maxlen=25) # recent cummulative rewards

# training loop
for i_episode in range(1, n_episodes+1):
    # TODO: initialize the environment and state
    if i_episode % 50 == 0:
        env = RecordVideo(gym.make("LunarLander-v2"), f"./DQN/batch{BATCH_SIZE}/eps{i_episode}")
    else:
        env = gym.make("LunarLander-v2")
    state = env.reset()
    done = False
    cr = 0 # episode cummulative rewards
    while not done:
        env.render()
        # TODO: select and perform an action
        action = agent.take_action(state, eps)
#         print(action)
        next_state, reward, done, info = env.step(action)
        # TODO: store transition in experience replay
        agent.experience_replay.store_trans(state, action, next_state, reward, done)
        # TODO: update agent
        agent.update_params()
        # TODO: update current state and episode cummulative rewards
        state = next_state
        cr += reward

    # TODO: decay epsilon
    eps = eps * eps_decay_rate
    eps = max(eps, eps_end)
    # TODO: update current cummulative rewards and recent cummulative rewards
    crs[i_episode - 1] = cr
    crs_recent.append(cr)
    # TODO: save agent every 50 episodes
    if i_episode % 50 == 0:
        agent.save(f"DQN_batch{BATCH_SIZE}_eps{i_episode}.pt")

    # print logs
    print('\rEpisode {}\tAverage Reward: {:.2f}\tEpsilon: {:.2f}'.format(i_episode, np.mean(crs_recent), eps), end="")
    if i_episode % 25 == 0:
        print('\rEpisode {}\tAverage Reward: {:.2f}\tEpsilon: {:.2f}'.format(i_episode, np.mean(crs_recent), eps))
# Plot cumulative reward of each episode
fig = plt.figure()
plt.figure(figsize=(10,6))
plt.plot(np.arange(len(crs)), crs)
plt.ylabel('Reward')
plt.xlabel('Episode')
plt.title(f"DQN with Batch size = {BATCH_SIZE}")
plt.show()
```

```
224
225 # Double DQN Agent
226
227 class DDQNAgent():
228     # NOTE: DON'T change initial values
229     def __init__(self, state_size, action_size, batch_size,
230                  gamma=0.99, buffer_size=25000, alpha=1e-4):
231         # network parameter
232         self.state_size = state_size
233         self.action_size = action_size
234
235         # hyperparameters
236         self.batch_size = batch_size
237         self.gamma = gamma
238
239         # experience replay
240         self.experience_buffer = ExperienceReplay(buffer_size)
241
242         # networks
243         self.value_net = DeepQNetwork(state_size, action_size).to(device)
244         self.target_value_net = DeepQNetwork(state_size, action_size).to(device)
245         self.update_target_network()
246
247         # optimizer
248         # TODO: create adam for optimizing network's parameter (learning rate=
    alpha)
249         # NOTE: target network parameters DOSEN'T update with optimizer
250         self.optimizer = optim.Adam(self.value_net.parameters(), lr=alpha)
251
252     def take_action(self, state, eps=0.0):
253         # TODO: take action using e-greedy policy
254         # NOTE: takes action using the greedy policy with a probability of 1-
    and a random action with a probability of
255         # NOTE:
256         self.value_net.eval()
257         rand_eps = random.random()
258         if rand_eps > eps:
259             with torch.no_grad():
260                 return torch.argmax(self.value_net(torch.tensor(state).to(device
    ))).detach().cpu().numpy()
261         else:
262             return np.random.randint(0, self.action_size)
263
264     def update_params(self):
265         if len(self.experience_buffer) < self.batch_size:
266             return
267         # transition batch
268         batch = Transition(*zip(*self.experience_buffer.sample(self.batch_size))
    )
269
270         state_batch = torch.from_numpy(np.vstack(batch.state)).float().to(device
    )
271         action_batch = torch.tensor(np.vstack(batch.action)).long().to(device)
272         next_state_batch = torch.from_numpy(np.vstack(batch.next_state)).float()
    .to(device)
273         reward_batch = torch.tensor(np.vstack(batch.reward)).float().to(device)
274         done_batch = torch.tensor(np.vstack(batch.done)).to(device)
275
276         # calculate loss w.r.t DQN algorithm
277         self.value_net.train()
```

```
        # STEP1
        q_targets_next = self.target_value_net(next_state_batch).detach().max(1)
    [0].unsqueeze(1)
        # STEP2
        # TODO: compute Q values [Q(s_t, a)]
        q_targets = reward_batch + self.gamma * q_targets_next * (1 - done_batch
    *1)
        q_expected = self.value_net(state_batch).gather(1, action_batch)
        # STEP3
        # TODO: compute mse loss
        loss = nn.functional.mse_loss(q_expected, q_targets)
        # TODO: optimize the model
        # NOTE: DON'T forget to set the gradients to zeros
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def update_target_network(self):
        # TODO: copy main network parameters to target network parameters
        self.target_value_net = copy.deepcopy(self.value_net)

    def save(self, fname):
        # TODO: save checkpoint
        torch.save(self.value_net, fname)

    def load(self, fname, device):
        # TODO: load checkpoint
        self.value_net = torch.load(fname, device)

# Set training parameters
n_episodes = 250
eps = 1.0
eps_decay_rate = 0.97
eps_end = 0.01
BATCH_SIZE = 32

# Training phase

# TODO: create agent
agent = DDQNAgent(state_size, action_size, batch_size=BATCH_SIZE)

crs = np.zeros(n_episodes) # cummulative rewards
crs_recent = deque(maxlen=25) # recent cummulative rewards

# training loop
for i_episode in range(1, n_episodes+1):
    # TODO: initialize the environment and state
    if i_episode % 50 == 0:
        env = RecordVideo(gym.make("LunarLander-v2"), f"./DDQN/batch{BATCH_SIZE
    }/eps{i_episode}")
    else:
        env = gym.make("LunarLander-v2")
    state = env.reset()
    done = False
    cr = 0 # episode cummulative rewards
    action_count = 0
    while not done:
        env.render()
        # TODO: select and perform an action
        action = agent.take_action(state, eps)
```

```
        next_state, reward, done, info = env.step(action)
        # TODO: store transition in experience replay
        agent.experience_buffer.store_trans(state, action, next_state, reward, done)
        # TODO: update agent
        agent.update_params()
        # TODO: update current state and episode cummulative rewards
        state = next_state
        cr += reward
        action_count += 1
        if action_count % 5 == 0:
            agent.update_target_network()

    # TODO: decay epsilon
    eps = eps * eps_decay_rate
    eps = max(eps, eps_end)

    # TODO: update current cummulative rewards and recent cummulative rewards
    crs[i_episode - 1] = cr
    crs_recent.append(cr)

    # TODO: save agent every 50 episodes
    if i_episode % 50 == 0:
        agent.save(f"DDQN/DDQN_batch{BATCH_SIZE}_eps{i_episode}.pt")

    # print logs
    print('\rEpisode {}\tAverage Reward: {:.2f}\tEpsilon: {:.2f}'.format(i_episode, np.mean(crs_recent), eps), end="")
    if i_episode % 25 == 0:
        print('\rEpisode {}\tAverage Reward: {:.2f}\tEpsilon: {:.2f}'.format(i_episode, np.mean(crs_recent), eps))

fig = plt.figure()
plt.figure(figsize=(10,6))
plt.plot(np.arange(len(crs)), crs)
plt.ylabel('Reward')
plt.xlabel('Episode')
plt.title(f"DDQN with Batch size = {BATCH_SIZE}")
plt.show()
```

Listing 1: Code Question 1