بسمه تعالی

مهدی وحیدمقدم

۴۰۲۱۳۰۷۴

مینی‌پروژه چهار یادگیری ماشین

فهرست مطالب

لینک مربوط به کد:

https://colab.research.google.com/drive/1okOAHkV_xB7qXiHL8N_6HgWwQmsTf-Dm?usp=sharing

لینک گیت هاب:

https://github.com/mvmoghadam1999/ML403_40213074/tree/main/MP4_ML_40213074

سوال ۲)

الف)

آ. در مورد محیط Lunar Landerمطالعه کرده و به‌صورت خلاصه ویژگی‌های آن را شرح دهید. ویژگی‌های مدنظر عبارتند از مشخصات فضای حالت، مشخصات فضای عمل و سیستم پاداش.

در کل فضای Lunar Lander یک شبیه‌ساز از فرود یک فضاپیما بر روی ماه است که در OpenAi Gym صورت می‌گیرد.در کل این محیط برای تست کردن الگوریتم‌های یادگیری تقویتی ساخته شده است.

مشخصات فضای حالت:

در فضای حالت این محیط ۸ پارامتر موردبررسی قرار می‌گیرند که عبارت اند از:

- موقعیت افقی
- موقعیت عمودی
- سرعت افقی
- سرعت عمودی
- زاویه‌ی چرخش
- سرعت زاویه‌ای
- برخورد پایه‌ی چپ فضاپیما با سطح
- برخورد پایه‌ی راست فضاپیما با سطح

مشخصات فضای عمل:

در این فضا اقداماتی بررسی می‌شود که فضاپیما می‌تواند انجام دهد که عبارت‌اند از:

- عدم اعمال نیرو(فضاپیما حرکت خاصی انجام ندهد)
- اعمال نیروی پیشران اصلی
- اعمال نیروی پیشران سمت چپ
- اعمال نیروی پیشران سمت راست

سیستم پاداش:

پاداش به صورت مثبت و منفی تعریف می‌شود که به صورت زیر هستند:

- فرود موفق فضاپیما: فرود موفق روی سطح که همراه با تماس پایه‌ی سمت چپ و سمت راست با سطح است،امتیاز مثبت بزرگی را در پی خواهد داشت.
- برخورد با سطح: برخورد با سطح با زاویه‌ی نامناسب یا سرعت نامناسب پاداش منفی را در پی خواهد داشت.
- سوخت مصرفی: استفاده‌ی کمتر از سوخت منجر به پاداش مثبت خواهد شد(در همین راستا اگر فضاپیما در زمان ۲۰ ثانیه نتواند فرود بیاید،امتیاز منفی بزرگی را دریافت خواهد کرد و فرایند تمام می‌شود)
- سرعت و موقعیت: نزدیک بودن سرعت و موقعیت به وضعیت مطلوب آن،امتیاز مثبت کوچکی به همراه خواهد داشت.

ب)

ب. عملکرد عامل را با رسم پاداش تجمعی در هر episode و برای batch sizeهای ۳۲، ۶۴ و ۱۲۸ بررسی کنید. تنها برای بهترین حالت به ازای episodeهای ۵۰، ۱۰۰، ۱۵۰، ۲۰۰ و ۲۵۰ فیلمی از عملکرد عامل تهیه کنید. در صورتی که عملکرد عامل به‌ازای هر سه مقدار batch size مشابه یکدیگر شد، یکی از آن‌ها را به دلخواه به‌عنوان بهترین حالت انتخاب کنید. در رابطه با انتخاب بهترین حالت علاوه بر معیار سرعت همگرایی به پاداش بهینهٔ معیار regret را نیز به‌صورت شهودی بررسی کنید.

قسمت اول کد:

```
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size):
        super(QNetwork, self).__init__()
        self.fc1 = torch.nn.Linear(state_size, 512)
        self.fc2 = torch.nn.ReLU()
        self.fc3 = torch.nn.LayerNorm(512)
        self.fc4 = torch.nn.Dropout(0.1)
        self.fc5 = torch.nn.Linear(512, 512)
```

```
        self.fc6 = torch.nn.ReLU()
        self.fc7 = torch.nn.LayerNorm(512)
        self.fc8 = torch.nn.Dropout(0.1)
        self.fc9 = torch.nn.Linear(512, 512)
        self.fc10 = torch.nn.ReLU()
        self.fc11 = torch.nn.Linear(512, action_size)

    def forward(self, x):
        return
self.fc11(self.fc10(self.fc9(self.fc8(self.fc7(self.fc6(self.fc5(self.fc4(
self.fc3(self.fc2(self.fc1(x)))))))))))
```

در این بخش از کد شبکه‌ی Q تعریف شده که به وسیله‌ی آن عامل ما آموزش می‌بیند.در واقع تفاوت شبکه DQN با شبکه‌ی Q این است که به جای آموزش با شبکه Q عادی از شبکه‌های deep و این مدل‌ها برای آموزش عامل استفاده می‌شود.این شبکه شامل ۱۱ لایه است که در کد بالا دیده می‌شود.

قسمت دوم کد:

```
class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def add(self, state, action, reward, next_state, done):
        e = (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[0] for e in experiences if
e is not None])).float()
        actions = torch.from_numpy(np.vstack([e[1] for e in experiences if
e is not None])).long()
        rewards = torch.from_numpy(np.vstack([e[2] for e in experiences if
e is not None])).float()
        next_states = torch.from_numpy(np.vstack([e[3] for e in
experiences if e is not None])).float()
        dones = torch.from_numpy(np.vstack([e[4] for e in experiences if e
is not None]).astype(np.uint8)).float()
```

```
        return (states, actions, rewards, next_states, dones)

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

دلایل تعریف این کلاس به صورت زیر است:

- یکی از مشکلات اصلی در یادگیری تقویتی، همبستگی بالا بین نمونه‌های متوالی تجربه شده توسط عامل است. این همبستگی می‌تواند منجر به یادگیری ناپایدار و ناکارآمد شود. با استفاده از Replay Buffer، تجربه‌های جمع‌آوری شده در طول زمان ذخیره می‌شوند و سپس به صورت تصادفی از این تجربه‌ها برای به‌روزرسانی شبکه عصبی استفاده می‌شود. این کار باعث می‌شود که همبستگی بین داده‌های ورودی کاهش یابد.
- در سناریوهای یادگیری تقویتی، جمع‌آوری داده‌ها معمولاً پرهزینه است. Replay Buffer این امکان را فراهم می‌کند که تجربه‌های گذشته چندین بار استفاده شوند. این کار باعث می‌شود که عامل یادگیری بهتری از تجربه‌های خود داشته باشد و از هر نمونه به بهترین شکل ممکن استفاده کند.

پس به طور کلی این کلاس وظیفه دارد که تجربه‌های عامل را ذخیره کند و به صورت تصادفی این تجربه‌ها را به عامل منتقل کند تا عامل یک روند را یاد نگیرد یا به اصطلاح بیش برازش نشود.

قسمت سوم کد:

```
class DQNAgent:
    def __init__(self, state_size, action_size, seed, buffer_size,
batch_size, gamma, lr, tau, update_every):
        self.state_size = state_size
        self.action_size = action_size
        self.gamma = gamma
        self.tau = tau
        self.update_every = update_every
        self.batch_size = batch_size
        self.seed = random.seed(seed)
```

```
        self.qnetwork_local = QNetwork(state_size, action_size).to(device)
        self.qnetwork_target = QNetwork(state_size,
action_size).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(),
lr=lr)

        self.memory = ReplayBuffer(buffer_size, batch_size)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        self.memory.add(state, action, reward, next_state, done)
        self.t_step = (self.t_step + 1) % self.update_every
        if self.t_step == 0:
            if len(self.memory) > self.batch_size:
                experiences = self.memory.sample()
                self.learn(experiences, self.gamma)

    def act(self, state, eps=0.):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        states, actions, rewards, next_states, dones = experiences

        Q_targets_next =
self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        loss = nn.MSELoss()(Q_expected, Q_targets)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        self.soft_update(self.qnetwork_local, self.qnetwork_target,
self.tau)
```

```
    def soft_update(self, local_model, target_model, tau):
        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)
```

این قسمت از کد عامل ما را تعریف می‌کند.این کلاس شامل توابع زیر است:

تابع _init_ :

این تابع در واقع شاکله‌ی اصلی کلاس است و پارامترهای اصلی در اینجا تعریف می‌شود و دو شبکه‌ی عصبی محلی یا local و شبکه عصبی هدف یا target تعریف می‌شود.

تابع step :

در این تابع تجربه‌های جدیدی که برای عامل اتفاق می‌افتد،به حافظه‌ی ما اضافه می‌شود.سپس بر اساس گام تعریف شده،با استفاده از این تجربه‌ها بعد از گام‌های تعیین‌شده یادگیری صورت می-گیرد.

تابع act :

در این تابع،با استفاده از شبکه‌ای که تعریف شده بود،عمل‌هایی که عامل انجام می‌دهد شکل می-گیرد.در واقع تعیین کننده‌ی این اعمال در شبکه‌ی عادی سیاست‌های epsilon-greedy بود و در این‌جا هم از شبکه‌ی تعریف شده استفاده می‌شود.

تابع learn :

در این قسمت یادگیری با استفاده از شبکه‌ی تعریف شده صورت می‌گیرد.در واقع این یادگیری تقریبا روندی شبیه یادگیری در شبکه‌های عمیق عادی دارد.به همان صورت ابتدا یک تابع Loss که در این‌جا MSE Loss است تعریف می‌شود.سپس مقدار اتلاف محاسبه شده و بعد از آن optimizer تعریف می‌شود که در این‌جا Adam است و بهینه‌سازی صورت گرفته و یادگیری در نهایت صورت می‌گیرد.

تابع soft_update :

در این قسمت همان‌طور که از نام تابع پیداست،با استفاده از یادگیری که انجام شده،پارامترهای شبکه هدف به سمت پارامترهای شبکه محلی که تعریف کرده‌ایم،به روزرسانی می‌شوند.

قسمت چهارم کد:

```
def train(agent, env, n_episodes=250, max_t=1000, eps_start=1.0,
eps_end=0.01, eps_decay=0.995):
    scores = []
    scores_window = deque(maxlen=100)
    eps = eps_start
    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            next_state, reward, done , info = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        scores_window.append(score)
        scores.append(score)
        eps = max(eps_end, eps_decay*eps)
        print(f'\rEpisode {i_episode}\tAverage Score:
{np.mean(scores_window):.2f}', end="")
        if i_episode % 50 == 0:
            print(f'\rEpisode {i_episode}\tAverage Score:
{np.mean(scores_window):.2f}')
            torch.save(agent.qnetwork_local.state_dict(),
'checkpoint.pth')
        if np.mean(scores_window)>=200.0:
            print(f'\nEnvironment solved in {i_episode-100}
episodes!\tAverage Score: {np.mean(scores_window):.2f}')
            torch.save(agent.qnetwork_local.state_dict(),
'checkpoint.pth')
            break
    return scores
```

این تابع در کل همان عمل یادگیری تقویتی را نشان می‌دهد که مشابه همان روند است.در این تابع،عملی توسط عامل انتخاب می‌شود.سپس این عمل در محیطی که تعریف کرده‌ایم انجام می‌شود و عامل یک بازخورد با توجه به عملی که انجام داده دریافت می‌کند.سپس با توجه به این بازخورد،عامل این تجربه را به حافظه خود اضافه می‌کند و در نهایت امتیاز خود را به‌روزرسانی خواهد کرد.همچنین بعد از هر اپیزود،امتیاز ذخیره می‌شود و مقدار اپسیلون تعریف شده کاهش یافته و از امتیازات کسب شده میانگین گرفته شده و نمایش داده می‌شود.

نتایج هر ۵۰ اپیزود یک بار نمایش داده می‌شوند(البته به ازای هر اپیزود نمایش داده می‌شود اما با تغییر اپیزود تغییر می‌کند اما هر ۵۰ اپیزود این عدد به صورت دائمی نمایش داده خواهد شد.).

همچنین وقتی میانگین امتیازات کسب‌شده توسط عامل به ۲۰۰ برسد،روند آموزش پایان می‌یابد و مدل آموزش‌دیده‌شده ذخیره خواهد شد.

قسمت پنجم کد:

```
env = gym.make('LunarLander-v2')
agent1 = DQNAgent(state_size=8, action_size=4, seed=0,
buffer_size=int(1e5), batch_size=128, gamma=0.99, lr=5e-4, tau=1e-3,
update_every=4)
scores1 = train(agent1, env)
```

در این قسمت،عامل با پارامترهای دلخواه تعریف شده و محیط هم تعریف می‌شود و تابع train اجرا می‌شود و امتیازات ذخیره می‌شود.

برای هر یک از batch_size های ۳۲ و ۶۴ و ۱۲۸ به ترتیب نتایج حاصل شده به صورت زیر است:

BS = 32

```
Episode 50  Average Score: -165.70
Episode 100 Average Score: -160.54
Episode 150 Average Score: -128.28
Episode 200 Average Score: -89.41
Episode 250  Average Score: -68.25
```

BS = 64

```
Episode 50  Average Score: -160.16
Episode 100 Average Score: -143.85
Episode 150 Average Score: -106.32
Episode 200 Average Score: -85.13
Episode 250  Average Score: -70.79
```

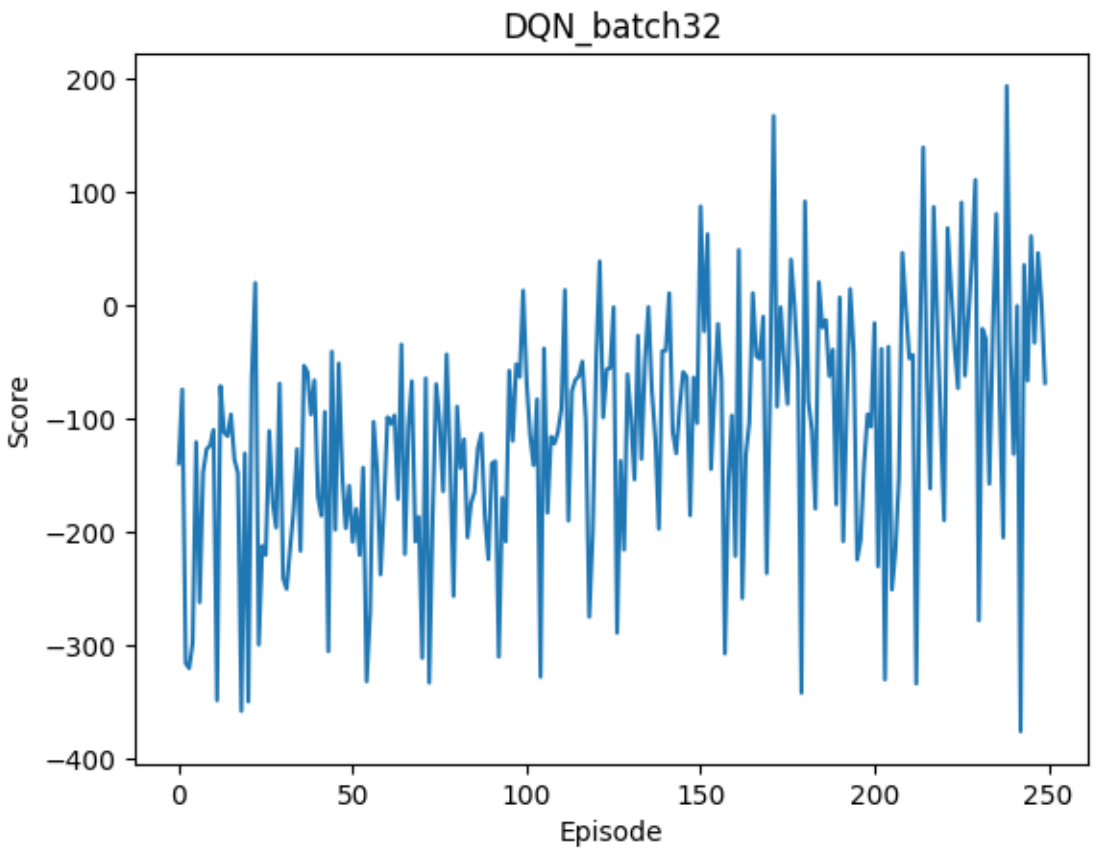
BS = 128

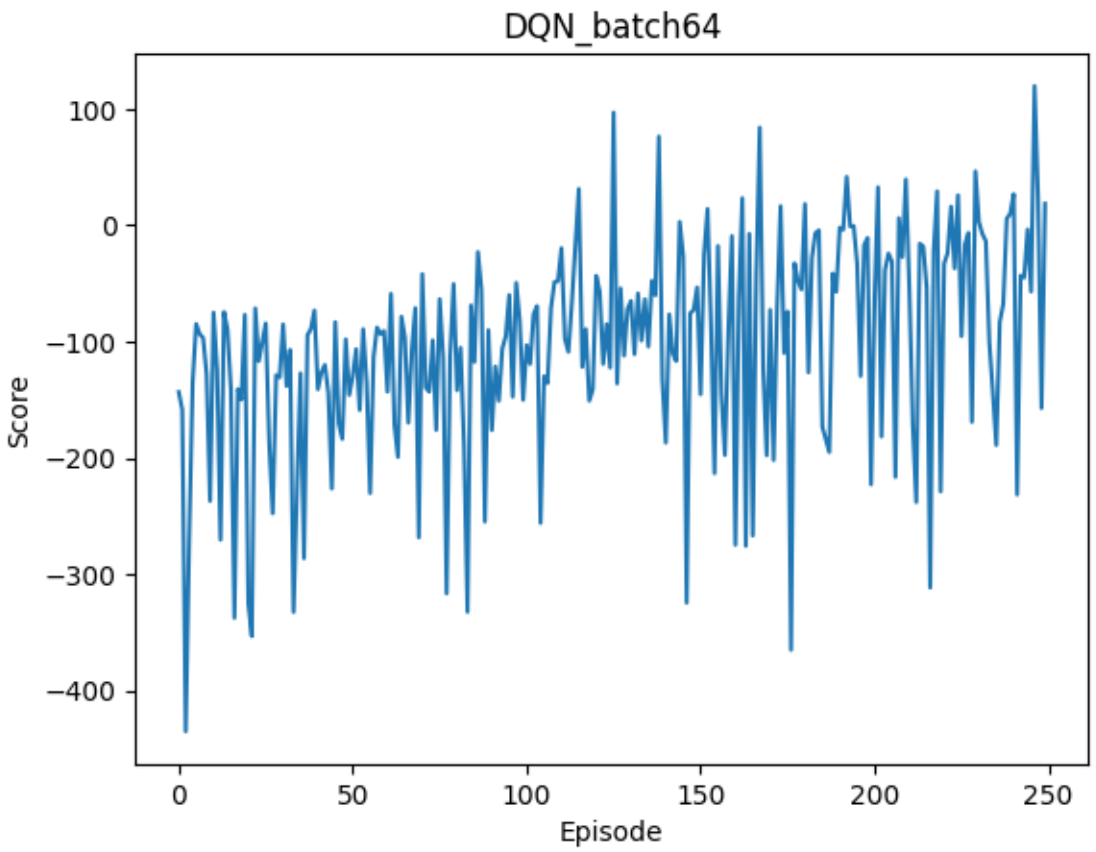
```
Episode 50  Average Score: -171.86
Episode 100 Average Score: -153.24
Episode 150 Average Score: -114.45
Episode 200 Average Score: -86.08
Episode 250  Average Score: -43.08
```

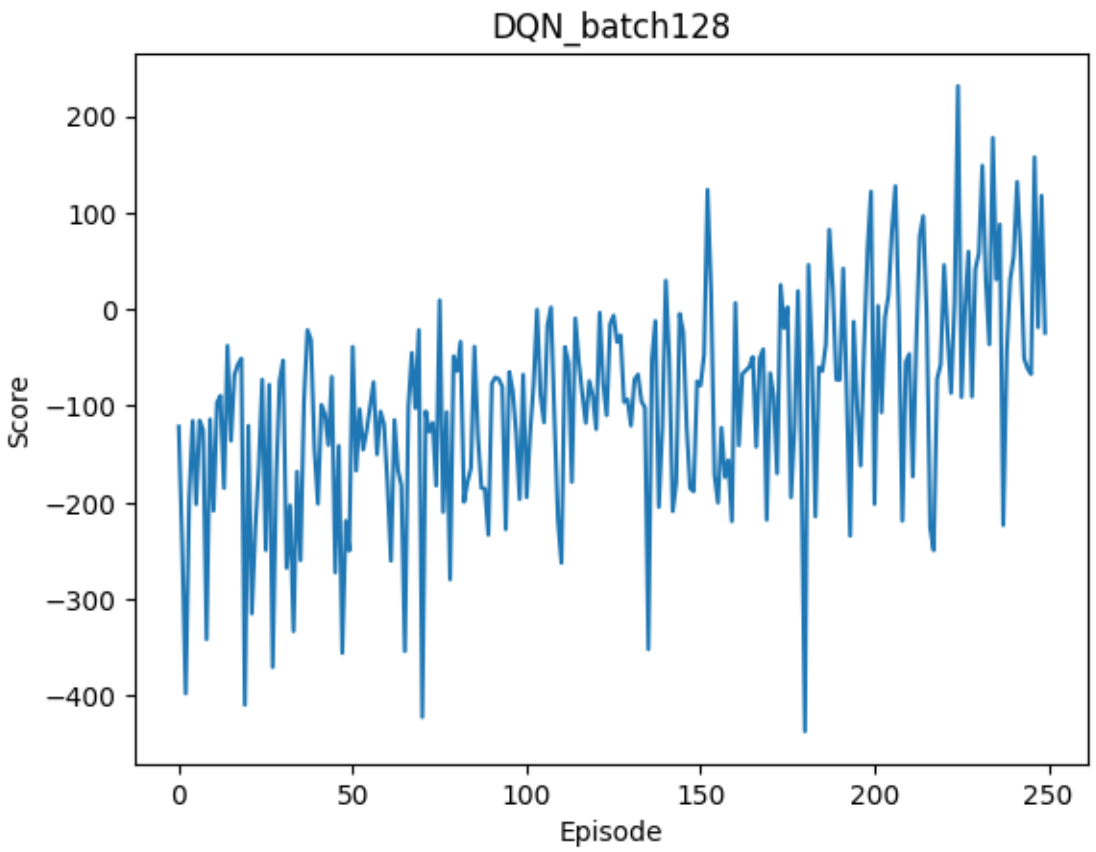
حال با استفاده از تابع زیر،مقادیر امتیاز به ازای اپیزودهای ۱ تا ۲۵۰ رسم می‌شود:

```
import matplotlib.pyplot as plt
fig = plt.figure()
plt.plot(np.arange(len(scores1)), scores1)
plt.ylabel('Score')
plt.xlabel('Episode')
plt.title(f"DQN_batch128")
plt.savefig(f"DQN_batch128.pdf")
plt.show()
```

داریم:


DQN_batch32
Score
Episode
200
100
0
−100
−200
−300
−400
0
50
100
150
200
250


DQN_batch64
100
0
−100
−200
−300
−400
Score
0
50
100
150
200
250
Episode



همان‌طور که مشاهده می‌شود،بهترین حالت برای $BS = 128$ است.

برای این حالت ویدیو را ضبط می‌کنیم.کدی که برای این کار در نظر گرفته شده به صورت زیر است:

```
if not os.path.exists('video'):
    os.makedirs('video')
def show_video(env_name):
    mp4list = glob.glob('video/*.mp4')
    if len(mp4list) > 0:
        mp4 = 'video/{}.mp4'.format(env_name)
        video = io.open(mp4, 'r+b').read()
        encoded = base64.b64encode(video)
        ipy_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")
```

```
def show_video_of_model(agent, env_name):
    virtual_display = Display(visible=0, size=(1400, 900))
    virtual_display.start()

    env = gym.make(env_name)
    vid = VideoRecorder(env, path="video/{}.mp4".format(env_name))


    state = env.reset()
    done = False
    while not done:
        action = agent.act(state)
        state, reward, done, _ = env.step(action)
        vid.capture_frame()

    vid.close()
    env.close()
    virtual_display.stop()
```

حال با استفاده از کد زیر،هم ویدیو ذخیره می‌شود و هم نمایش داده خواهد شد:

```
show_video_of_model(agent1, 'LunarLander-v2')
show_video('LunarLander-v2')
```

ویدیو مربوط به این حالت در این لینک موجود است.

همان‌طور که می‌بینیم،در این حالت اصلا یادگیری به خوبی صورت نگرفته است.حال تعداد اپیزودها را به ۱۰۰۰ افزایش می‌دهیم تا ببینیم نتیجه به چه صورت خواهد بود.

تابع train در این حالت به صورت زیر است:

```
def train2(agent, env, n_episodes=1000, max_t=1000, eps_start=1.0,
eps_end=0.01, eps_decay=0.995):
    scores = []
    scores_window = deque(maxlen=100)
    eps = eps_start
    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
```

```
            action = agent.act(state, eps)
            next_state, reward, done , info = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        scores_window.append(score)
        scores.append(score)
        eps = max(eps_end, eps_decay*eps)
        print(f'\rEpisode {i_episode}\tAverage Score:
{np.mean(scores_window):.2f}', end="")
        if i_episode % 50 == 0:
            print(f'\rEpisode {i_episode}\tAverage Score:
{np.mean(scores_window):.2f}')
            torch.save(agent.qnetwork_local.state_dict(),
'checkpoint.pth')
        if np.mean(scores_window)>=200.0:
            print(f'\nEnvironment solved in {i_episode-100}
episodes!\tAverage Score: {np.mean(scores_window):.2f}')
            torch.save(agent.qnetwork_local.state_dict(),
'checkpoint.pth')
            break
    return scores
```

تنها تفاوت این تابع با تابع قبل در تعداد اپیزودها است.کد زیر برای آموزش این عامل است:

```
env = gym.make('LunarLander-v2')
agent4 = DQNAgent(state_size=8, action_size=4, seed=0,
buffer_size=int(1e5), batch_size=128, gamma=0.99, lr=5e-4, tau=1e-3,
update_every=4)
scores4 = train2(agent4, env)
```

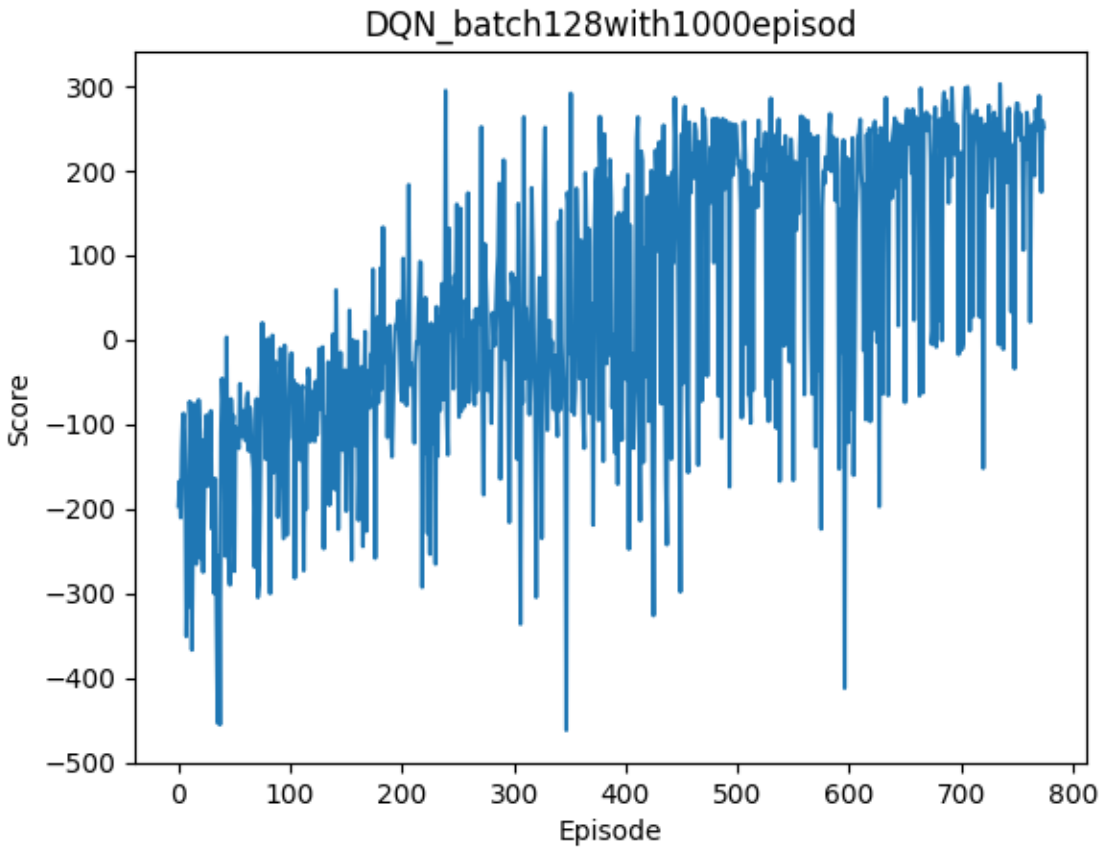
نتیجه به صورت زیر است:

```
Episode 50  Average Score: -181.55
Episode 100 Average Score: -152.64
Episode 150 Average Score: -107.98
Episode 200 Average Score: -75.44
Episode 250 Average Score: -42.41
Episode 300 Average Score: -11.21
Episode 350 Average Score: -5.00
Episode 400 Average Score: 10.46
Episode 450 Average Score: 38.44
Episode 500 Average Score: 103.23
Episode 550 Average Score: 146.37
```

```
Episode 600 Average Score: 129.45
Episode 650 Average Score: 139.17
Episode 700 Average Score: 170.32
Episode 750 Average Score: 192.26
Episode 775 Average Score: 201.79
Average Score: 201.79     Environment solved in 675 episodes!
```

همان‌طور که می‌بینیم،در این حالت با گذشت ۶۷۵ اپیزود آموزش کامل شده است.

همچنین نمودار تغییرات امتیاز به صورت زیر است:



ویدیو مربوط به این حالت هم در این لینک در دسترس است.

ج)

ج. عملکرد مدل DQN و DDQN را با رسم پاداش تجمعی در هر episode و به‌ازای batch size برابر مقایسه کنید. برای هر دو مدل به‌ازای episodeهای ۱۰۰ و ۲۵۰، قیلمی از عملکرد مدل تهیه کنید.

کد مربوط به این قسمت،شبیه به قسمت قبل است فقط کد مربوط به قسمت عامل فرق دارد که به صورت زیر است:

```
class DDQNAgent:
    def __init__(self, state_size, action_size, seed, buffer_size,
batch_size, gamma, lr, tau, update_every , device):
        self.state_size = state_size
        self.action_size = action_size
        self.gamma = gamma
        self.tau = tau
        self.update_every = update_every
        self.batch_size = batch_size
        self.seed = random.seed(seed)

        self.qnetwork_local = QNetwork(state_size, action_size).to(device)
        self.qnetwork_target = QNetwork(state_size,
action_size).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(),
lr=lr)

        self.memory = ReplayBuffer(buffer_size, batch_size , device)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        self.memory.add(state, action, reward, next_state, done)
        self.t_step = (self.t_step + 1) % self.update_every
        if self.t_step == 0:
            if len(self.memory) > self.batch_size:
                experiences = self.memory.sample()
                self.learn(experiences, self.gamma)

    def act(self, state, eps=0.):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
```

```
        states, actions, rewards, next_states, dones = experiences

        Q_local_next =
self.qnetwork_local(next_states).detach().max(1)[1].unsqueeze(1)
        Q_targets_next =
self.qnetwork_target(next_states).detach().gather(1, Q_local_next)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        loss = nn.MSELoss()(Q_expected, Q_targets)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        self.soft_update(self.qnetwork_local, self.qnetwork_target,
self.tau)

    def soft_update(self, local_model, target_model, tau):
        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)
```

تفاوت‌های اصلی این شبکه با شبکه‌ی DQN در موارد زیر است:

شبکه‌ی DQN ، از یک شبکه Q واحد برای مقادیر Q هدف و فعلی استفاده می کند که ممکن است منجر به تخمین بیش از حد بالقوه مقادیر Q یا overestimation شود.اما DDQN از دو شبکه Q مجزا استفاده می کند: یکی برای مقادیر Q هدف و دیگری برای مقادیر Q فعلی که مشکل تخمین بیش از حد موجود در DQN را کاهش می دهد.

شبکه‌ی DQN، از یک به‌روزرسانی Q-value هدف ساده با استفاده از حداکثر Q-value وضعیت بعدی مطابق با شبکه Q فعلی استفاده می کند.اما DDQN، از شبکه Q هدف برای انتخاب عملکرد برای حالت بعدی استفاده می کند و سپس با استفاده از شبکه Q فعلی، Q-value را محاسبه می کند. این به کاهش تخمین بیش از حد کمک می کند.

شبکه‌ی DQN، بدون پرداختن به سوگیری برآورد بیش از حد را دارد اما DQN، به طور خاص برای پرداختن به سوگیری تخمین بیش از حد با معرفی رویکرد یادگیری Q دوگانه، که از دو شبکه Q استفاده می کند، طراحی شده است.

شبکه‌ی DQN، مستعد برآورد بیش از حد، که می تواند منجر به یادگیری خط‌مشی غیربهینه شود.اما DDQN، تمایل به ارائه تخمین های Q-value دقیق تری دارد که منجر به بهبود ثبات و همگرایی بهتر در فرآیند یادگیری می شود.

البته از نظر هزینه‌های پیاده‌سازی،شبکه‌ی DQN پیاده‌سازی راحت‌تری را خواهد داشت اما شبکه‌ی DDQN به دلیل این که از دو شبکه استفاده می‌کند،طبعا پیاده‌سازی پرهزینه‌تر و مشکل‌تری نیز خواهد داشت.

در شبکه‌ی تعریف‌شده،این تفاوت‌ها اعمال شده‌اند.

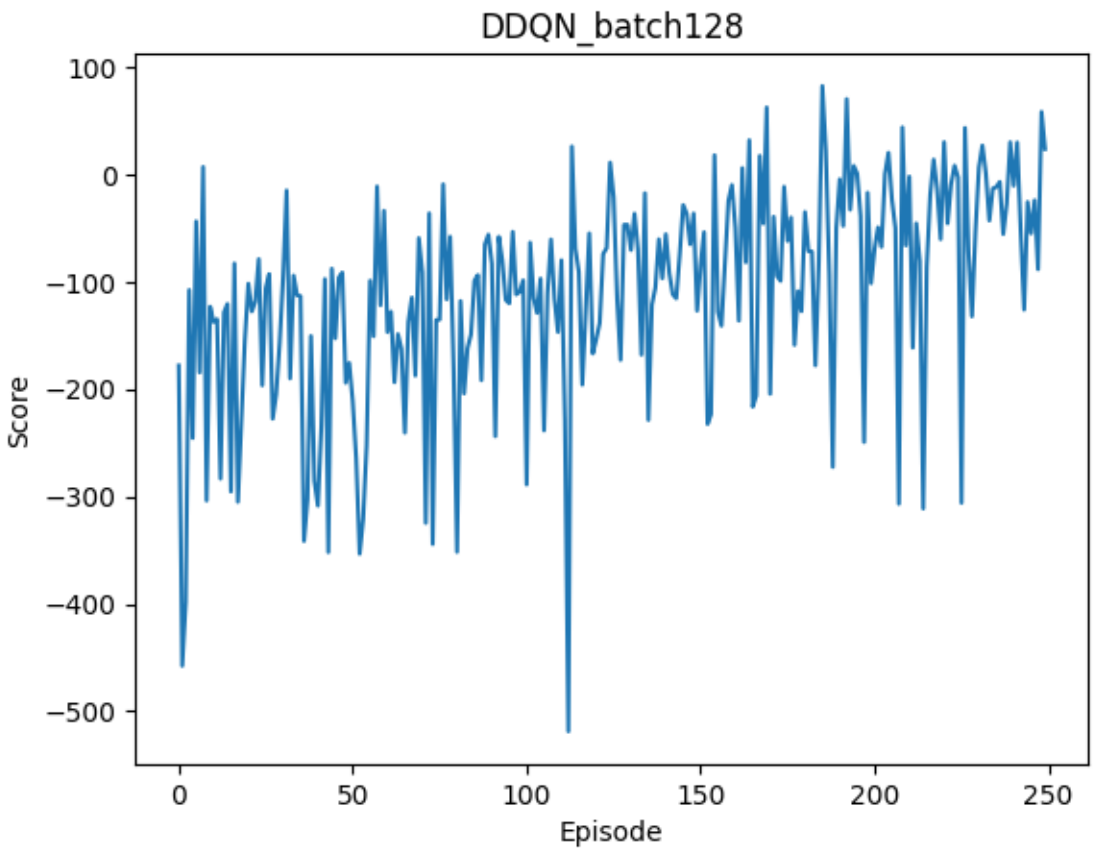
بقیه‌ی قسمت‌های کد شبیه حالت DQN است.

برای حالت DDQN با BS = 128 داریم:

```
Episode 50  Average Score: -176.15
Episode 100 Average Score: -161.41
Episode 150 Average Score: -127.47
Episode 200 Average Score: -91.01
Episode 250 Average Score: -59.23
```

می‌بینیم که مقدار نهایی امتیاز کمی بدتر از حالت DQN است.

نمودار تغییرات امتیاز به صورت زیر است:



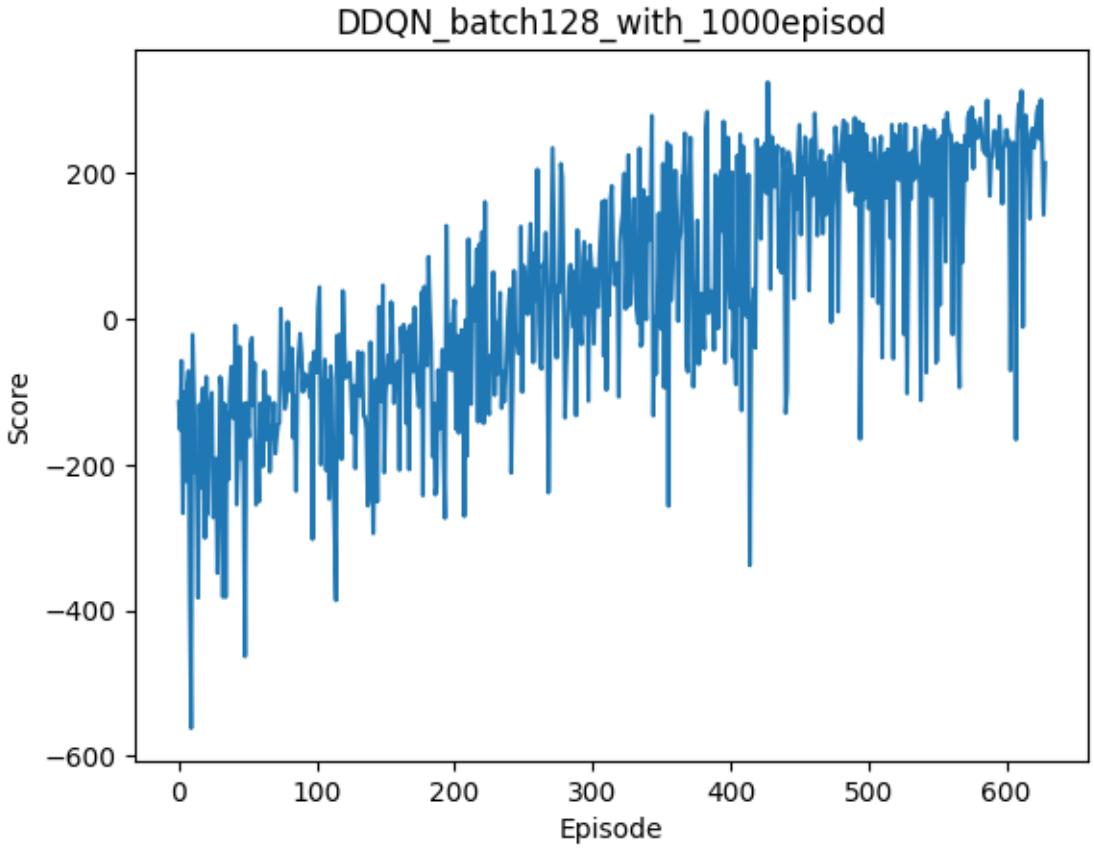
همچنین فیلم مربوط به این حالت در این لینک موجود است.

حال مثل حالت قبل این کار با ۸۰۰ اپیزود انجام شده است.(دلیل این که ۱۰۰۰ اپیزود قرار داده نشد این است که در حالت قبل دیدیم که در ۶۷۵ اپیزود آموزش به طور کامل صورت گرفت).نتایج به صورت زیر است:

```
Episode 50  Average Score: -184.27
Episode 100 Average Score: -149.37
Episode 150 Average Score: -115.85
Episode 200 Average Score: -94.59
Episode 250 Average Score: -57.11
Episode 300 Average Score: -3.79
Episode 350 Average Score: 51.65
Episode 400 Average Score: 78.01
Episode 450 Average Score: 102.97
Episode 500 Average Score: 152.49
Episode 550 Average Score: 177.34
Episode 600 Average Score: 187.12
Episode 629 Average Score: 201.12
Average Score: 201.12     Environment solved in 529 episodes!
```

همان‌طور که می‌بینیم در این حالت،با گذشت ۵۲۹ اپیزود آموزش به طور کامل انجام می‌شود و نسبت به حالت DQN آموزش کامل سریعتر صورت گرفته است.

نمودار تغییرات امتیاز به صورت زیر است:



پس در حالت آموزش با ۲۵۰ اپیزود،شبکه‌ی DQN عملکرد بهتری داشته است اما در حالت آموزش کامل،شبکه‌ی DDQN سریعتر آموزش دیده است.

فیلم مربوط به این حالت هم در این لینک قابل مشاهده است.

اگر به نحوه‌ی حرکت ربات در حالت ۲۵۰ اپیزود در دو شبکه نگاه کنیم،می‌بینیم که در حالت DQN ربات در مکانی غیر از مکان معین‌شده به زمین می‌نشیند اما در حالت DDQN،ربات به محل تعیین‌شده نمی‌رسد اما در همان راستا در تلاش است که به زمین بنشیند و در مکان دیگری فرود ندارد.