دانشگاه صنعتی خواجه نصیرالدین طوسی
دانشکدهٔ مهندسی برق - گروه مهندسی کنترل

درس یادگیری ماشین

گزارش مینی پروژه ۴

نام و نام خانوادگی	دانیال عبداللهی نژاد
شمارهٔ دانشجویی	۴۰۱۰۹۱۶۴
تاریخ	تیر ماه ۱۴۰۳

فهرست مطالب

فهرست تصاویر

۱ سوال اول

۲ سوال دوم

در ابتدای کتابخانه‌ها به صورت زیر برای استفاده در مراحل بعد وارد می‌شوند.

```
import gym
import io
import os
import glob
import torch
import base64
import numpy as np
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

display = Display(visible=0, size=(1400, 900))
display.start()
```

کتابخانه stable-baselines3 یک مجموعه پیاده‌سازی‌های پایدار و بهینه از الگوریتم‌های یادگیری تقویتی در پایتون است. این کتابخانه به راحتی قابل استفاده و توسعه است و شامل الگوریتم‌های معروفی مانند DQN، PPO، A2C و SAC می‌باشد.

```
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")
```

این تابع برای نمایش ویدیوهای ضبط شده از محیط Gym استفاده می‌شود. ویدیوها به صورت base64 رمزگذاری شده و در مرورگر نمایش داده می‌شوند.

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward', 'done'))

class ExperienceReplay():
    def __init__(self, capacity):
        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)
```

Experience Replay یکی از اجزای کلیدی در پیاده‌سازی‌های DQN و DDQN در کد ارائه شده است. این بخش به عامل اجازه می‌دهد تا تجربیات خود را در یک حافظه ذخیره کند و از آن‌ها به صورت نمونه‌گیری تصادفی در فرآیند آموزش استفاده کند.

Experience Replay با استفاده از یک صف دوتایی (deque) پیاده‌سازی شده است که ظرفیت مشخصی دارد. این حافظه تجربیات عامل را به صورت transitoin شامل state، action، next state، reward و done ذخیره می‌کند.

متد store_trans برای ذخیره‌سازی transition در حافظه استفاده می‌شود. این متد یک transition جدید شامل موارد گفته شده را به حافظه اضافه می‌کند.

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

sample برای نمونه‌گیری تصادفی از transition در حافظه استفاده می‌شود. این متد تعداد مشخصی از transitionها را به صورت تصادفی از حافظه انتخاب می‌کند تا در فرآیند به‌روزرسانی شبکه عصبی استفاده شوند.

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

در فرآیند آموزش، عامل transition را در حافظه Experience Replay ذخیره می‌کند و سپس به صورت دوره‌ای از این حافظه نمونه‌گیری کرده و شبکه عصبی خود را به‌روزرسانی می‌کند. این روش به کاهش همبستگی بین داده‌های ورودی کمک می‌کند و به پایداری و کارایی بهتر مدل منجر می‌شود.

```
for i_episode in range(1, n_episodes+1):
    ...
    while not done:
        ...
        agent.experience_replay.store_trans(state, action, next_state, reward, done or truncated)
        agent.update_params()
        ...
```

این تکنیک از بیش‌برازش به داده‌های اخیر جلوگیری می‌کند.

```python
class DeepQNetwork(nn.Module):
    def __init__(self, state_size, action_size):
        super(DeepQNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(state_size, 512),
            nn.ReLU(),
            nn.LayerNorm(512),
            nn.Dropout(0.1),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.LayerNorm(512),
            nn.Dropout(0.1),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, action_size)
        )

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

کلاس DeepQNetwork با استفاده از یک شبکه عصبی عمیق، تخمینی از Q-values را برای هر اقدام در یک حالت خاص تولید می‌کند که به عامل کمک می‌کند تا بهترین اقدام ممکن را انتخاب کند.

ورودی این شبکه، حالت (state) محیط است که به صورت یک بردار با اندازه state_size ارائه می‌شود. این بردار می‌تواند شامل ویژگی‌های مختلفی از محیط باشد که برای تصمیم‌گیری عامل مهم هستند. به عنوان مثال، در محیط LunarLander-v2، ورودی شامل موقعیت و سرعت فضاپیما و زاویه و نرخ چرخش آن است.

خروجی شبکه، یک بردار با اندازه action_size است که هر عنصر آن نشان‌دهنده Q-value برای یک اقدام خاص است. این Q-values تخمینی از مقدار کل پاداشی است که عامل می‌تواند از هر حالت و با انجام هر اقدام خاص انتظار داشته باشد.

شبکه DeepQNetwork شامل لایه‌های زیر است:

- لایه **Linear**: این لایه‌ها عملیات خطی را بر روی ورودی انجام می‌دهند و هر کدام شامل تعداد مشخصی نورون هستند. در این شبکه، سه لایه خطی با ۵۱۲ نورون وجود دارد.
- لایه **ReLU**: این لایه‌ها یک تابع فعال‌سازی غیرخطی را اعمال می‌کنند که به مدل اجازه می‌دهد تا روابط غیرخطی پیچیده را یاد بگیرد. ReLU مخفف Rectified Linear Unit است.
- لایه **LayerNorm**: این لایه‌ها نرمال‌سازی لایه‌ای را انجام می‌دهند که به پایداری و همگرایی سریع‌تر آموزش کمک می‌کند.
- لایه **Dropout**: این لایه‌ها به منظور جلوگیری از بیش‌برازش استفاده می‌شوند و برخی از نورون‌ها را به صورت تصادفی در هر دوره آموزشی غیر فعال می‌کنند.

تابع forward برای عبور ورودی از شبکه و تولید خروجی استفاده می‌شود. این تابع ورودی x را از طریق تمام لایه‌های تعریف شده در __init__ عبور می‌دهد و خروجی نهایی را که شامل Q-values است، باز می‌گرداند.

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

کلاس DQNAgent نماینده عامل DQN است. این کلاس شامل توابعی برای انتخاب اقدام، به‌روزرسانی پارامترها و ذخیره و بارگذاری مدل است. عامل از یک شبکه عصبی برای انتخاب و ارزیابی اقدامات استفاده می‌کند.

کلاس DQNAgent نماینده عامل Lunar Lander است. در هر اپیزود، این عامل با محیط تعامل می‌کند تا بهترین اقدامات را بر اساس حالت‌های فعلی انجام دهد.

```
class DQNAgent():
    def __init__(self, state_size, action_size, batch_size, gamma=0.99, buffer_size=25000, alpha
    =1e-4):
        self.state_size = state_size
        self.action_size = action_size
        self.batch_size = batch_size
        self.gamma = gamma
        self.experience_replay = ExperienceReplay(buffer_size)
        self.value_net = DeepQNetwork(state_size, action_size).to(device)
        self.optimizer = optim.Adam(self.value_net.parameters(), lr=alpha)

    def take_action(self, state, eps=0.0):
        self.value_net.eval()
        if random.random() > eps:
            with torch.no_grad():
                return torch.argmax(self.value_net(torch.tensor(state).float().unsqueeze(0).to(
    device))).item()
        else:
            return np.random.randint(0, self.action_size)

    def update_params(self):
        if len(self.experience_replay) < self.batch_size:
            return
        batch = Transition(*zip(*self.experience_replay.sample(self.batch_size)))

        state_batch = torch.tensor(np.array(batch.state), dtype=torch.float32).to(device)
        action_batch = torch.tensor(batch.action, dtype=torch.int64).unsqueeze(1).to(device)
        next_state_batch = torch.tensor(np.array(batch.next_state), dtype=torch.float32).to(
    device)
        reward_batch = torch.tensor(batch.reward, dtype=torch.float32).unsqueeze(1).to(device)
        done_batch = torch.tensor(batch.done, dtype=torch.float32).unsqueeze(1).to(device)

        q_expected = self.value_net(state_batch).gather(1, action_batch)
        q_targets_next = self.value_net(next_state_batch).detach().max(1)[0].unsqueeze(1)
        q_targets = reward_batch + (self.gamma * q_targets_next * (1 - done_batch))
        loss = F.mse_loss(q_expected, q_targets)

        self.optimizer.zero_grad()
```

```
        loss.backward()
        self.optimizer.step()

    def save(self, fname):
        torch.save(self.value_net.state_dict(), fname)

    def load(self, fname):
        self.value_net.load_state_dict(torch.load(fname, map_location=device))
```

در هر اپیزود، عامل با تنظیمات اولیه محیط شروع می‌کند. این شامل reset محیط و تنظیم حالت اولیه است. این تابع مقداردهی اولیه برای کلاس DQNAgent را انجام می‌دهد و پارامترهای مختلف را تنظیم می‌کند:

- **state_size**: اندازه بردار حالت‌ها.
- **action_size**: تعداد اقدامات ممکن.
- **batch_size**: اندازه دسته‌ها برای نمونه‌گیری از حافظه Experience Replay.
- **gamma**: نرخ تخفیف برای محاسبه پاداش‌های آینده.
- **buffer_size**: ظرفیت حافظه Experience Replay.
- **alpha**: نرخ یادگیری برای به‌روزرسانی پارامترهای شبکه.

```
def __init__(self, state_size, action_size, batch_size, gamma=0.99, buffer_size=25000, alpha=1e
    -4):
    self.state_size = state_size
    self.action_size = action_size
    self.batch_size = batch_size
    self.gamma = gamma
    self.experience_replay = ExperienceReplay(buffer_size)
    self.value_net = DeepQNetwork(state_size, action_size).to(device)
    self.optimizer = optim.Adam(self.value_net.parameters(), lr=alpha)
```

در هر گام زمانی، عامل یک اقدام را با استفاده از سیاست e-greedy انتخاب می‌کند. اگر عدد تصادفی کمتر از eps باشد، اقدام تصادفی انتخاب می‌شود (exploration). در غیر این صورت، اقدام با بالاترین Q-value تخمین زده شده توسط شبکه عصبی انتخاب می‌شود(exploitation).

عامل اقدام انتخاب شده را اجرا می‌کند و محیط حالت جدید، پاداش، و وضعیت اتمام (done) را باز می‌گرداند.

```
def take_action(self, state, eps=0.0):
    self.value_net.eval()
    if random.random() > eps:
        with torch.no_grad():
            return torch.argmax(self.value_net(torch.tensor(state).float().unsqueeze(0).to(device
    ))).item()
    else:
        return np.random.randint(0, self.action_size)
```

این تابع پارامترهای شبکه عصبی را به‌روزرسانی می‌کند:

- اگر تعداد تجربیات در حافظه کمتر از batch_size باشد، از به‌روزرسانی خودداری می‌کند.
- انتقال‌ها را به صورت دسته‌ای نمونه‌گیری می‌کند و بردارهای حالت، اقدام، حالت بعدی، پاداش و علامت پایان را ایجاد می‌کند.
- Q-valueهای فعلی و هدف را محاسبه می‌کند و از loss میانگین مربعات خطا (MSE) برای به‌روزرسانی شبکه استفاده می‌کند.

```
def update_params(self):
    if len(self.experience_replay) < self.batch_size:
        return
    batch = Transition(*zip(*self.experience_replay.sample(self.batch_size)))

    state_batch = torch.tensor(np.array(batch.state), dtype=torch.float32).to(device)
    action_batch = torch.tensor(batch.action, dtype=torch.int64).unsqueeze(1).to(device)
    next_state_batch = torch.tensor(np.array(batch.next_state), dtype=torch.float32).to(device)
    reward_batch = torch.tensor(batch.reward, dtype=torch.float32).unsqueeze(1).to(device)
    done_batch = torch.tensor(batch.done, dtype=torch.float32).unsqueeze(1).to(device)

    q_expected = self.value_net(state_batch).gather(1, action_batch)
    q_targets_next = self.value_net(next_state_batch).detach().max(1)[0].unsqueeze(1)
    q_targets = reward_batch + (self.gamma * q_targets_next * (1 - done_batch))
    loss = F.mse_loss(q_expected, q_targets)

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

این تابع مدل یادگیری شده را ذخیره می‌کند.

```
def save(self, fname):
    torch.save(self.value_net.state_dict(), fname)
```

این تابع مدل یادگیری شده را بارگذاری می‌کند.

```
def load(self, fname):
    self.value_net.load_state_dict(torch.load(fname, map_location=device))
```

این فرآیند در هر اپیزود تکرار می‌شود تا عامل بتواند بهترین استراتژی را برای تعامل با محیط Lunar Lander یاد بگیرد.

حلقه آموزش برای اندازه دسته ۳۲

```
env = gym.make('LunarLander-v2', render_mode="rgb_array")
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```
n_episodes = 250
eps = 1.0
eps_decay_rate = 0.97
eps_end = 0.01
BATCH_SIZE = 32

agent = DQNAgent(state_size, action_size, batch_size=BATCH_SIZE)
crs = np.zeros(n_episodes)
crs_recent = deque(maxlen=25)

for i_episode in range(1, n_episodes+1):
    env = RecordVideo(gym.make("LunarLander-v2", render_mode="rgb_array"), f"./DQN/batch{BATCH_SIZE}/eps{i_episode}") if i_episode % 50 == 0 else gym.make("LunarLander-v2", render_mode="rgb_array")
    state, info = env.reset()
    done = False
    cr = 0
    while not done:
        action = agent.take_action(state, eps)
        next_state, reward, done, truncated, info = env.step(action)
        agent.experience_replay.store_trans(state, action, next_state, reward, done or truncated)
        agent.update_params()
        state = next_state
        cr += reward

    eps = max(eps * eps_decay_rate, eps_end)
    crs[i_episode - 1] = cr
    crs_recent.append(cr)
    if i_episode % 50 == 0:
        agent.save(f"q_net_batch{BATCH_SIZE}_eps{i_episode}.pt")

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

fig = plt.figure()
plt.plot(np.arange(len(crs)), crs)
plt.ylabel('Reward')
plt.xlabel('Episode')
plt.title(f"DQN_batch{BATCH_SIZE}")
plt.savefig(f"DQN_batch{BATCH_SIZE}.pdf")
plt.show()
os.system("zip -r DQN.zip DQN/")
```

این بخش از کد، حلقه آموزش را برای اندازه دسته ۳۲ اجرا می‌کند. در هر اپیزود، محیط reset می‌شود و عامل با استفاده از سیاست e-greedy اقدام می‌کند. تجربه‌های عامل در ExperienceReplay ذخیره می‌شوند و پارامترهای شبکه به‌روزرسانی می‌شوند. در هر ۵۰ اپیزود، مدل ذخیره می‌شود و در پایان، نتایج آموزش به صورت نمودار نمایش داده می‌شوند.

```
env = gym.make('LunarLander-v2', render_mode="rgb_array")
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

در این بخش، محیط LunarLander-v2 تعریف می‌شود. اندازه‌های حالت‌ها و اقدامات از محیط استخراج می‌شوند.

```
n_episodes = 250
eps = 1.0
eps_decay_rate = 0.97
eps_end = 0.01
BATCH_SIZE = 32
```

در این بخش، پارامترهای آموزشی تنظیم می‌شوند. تعداد اپیزودها، مقدار اولیه epsilon، نرخ کاهش epsilon، مقدار نهایی epsilon و اندازه دسته (Batch Size) تعیین می‌شوند.

```
agent = DQNAgent(state_size, action_size, batch_size=BATCH_SIZE)
crs = np.zeros(n_episodes)
crs_recent = deque(maxlen=25)
```

عامل DQN تعریف می‌شود و متغیرهایی برای ذخیره پاداش‌های کسب شده در هر اپیزود و پاداش‌های اخیر ایجاد می‌شوند.

```
for i_episode in range(1, n_episodes+1):
    env = RecordVideo(gym.make("LunarLander-v2", render_mode="rgb_array"), f"./DQN/batch{
    BATCH_SIZE}/eps{i_episode}") if i_episode % 50 == 0 else gym.make("LunarLander-v2",
    render_mode="rgb_array")
    state, info = env.reset()
    done = False
    cr = 0
    while not done:
        action = agent.take_action(state, eps)
        next_state, reward, done, truncated, info = env.step(action)
        agent.experience_replay.store_trans(state, action, next_state, reward, done or truncated)
        agent.update_params()
        state = next_state
        cr += reward

    eps = max(eps * eps_decay_rate, eps_end)
    crs[i_episode - 1] = cr
    crs_recent.append(cr)
    if i_episode % 50 == 0:
        agent.save(f"q_net_batch{BATCH_SIZE}_eps{i_episode}.pt")

    print(f'\rEpisode {i_episode}\tAverage Reward: {np.mean(crs_recent):.2f}\tEpsilon: {eps:.2f}'
    , end="")
```

```
    if i_episode % 25 == 0:
        print(f'\rEpisode {i_episode}\tAverage Reward: {np.mean(crs_recent):.2f}\tEpsilon: {eps
    :.2f}')
```

در این بخش، حلقه اصلی آموزش اجرا می‌شود:

- در ابتدای هر اپیزود، محیط reset می‌شود و حالت اولیه تنظیم می‌شود.
- در هر گام زمانی، عامل یک اقدام را با استفاده از سیاست e-greedy انتخاب می‌کند.
- اقدام انتخاب شده اجرا می‌شود و حالت جدید، پاداش و وضعیت اتمام (done) از محیط بازگردانده می‌شوند.
- انتقال‌ها در حافظه Experience Replay ذخیره می‌شوند.
- پارامترهای شبکه عصبی به‌روزرسانی می‌شوند.
- حالت فعلی به حالت جدید به‌روزرسانی می‌شود و پاداش کسب شده در این گام به پاداش کلی اپیزود اضافه می‌شود.

```
eps = max(eps * eps_decay_rate, eps_end)
crs[i_episode - 1] = cr
crs_recent.append(cr)
```

در پایان هر اپیزود، مقدار epsilon کاهش می‌یابد و پاداش کسب شده در این اپیزود ذخیره می‌شود.

```
if i_episode % 50 == 0:
    agent.save(f"q_net_batch{BATCH_SIZE}_eps{i_episode}.pt")

fig = plt.figure()
plt.plot(np.arange(len(crs)), crs)
plt.ylabel('Reward')
plt.xlabel('Episode')
plt.title(f"DQN_batch{BATCH_SIZE}")
plt.savefig(f"DQN_batch{BATCH_SIZE}.pdf")
plt.show()
os.system("zip -r DQN.zip DQN/")
```

مدل یادگیری شده در فواصل مشخص (هر ۵۰ اپیزود) ذخیره می‌شود و در پایان، نتایج آموزش به صورت نمودار نمایش داده می‌شوند.

حلقه آموزش برای **DDQN**

```
env = gym.make('LunarLander-v2', render_mode="rgb_array")
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

n_episodes = 250
eps = 1.0
```

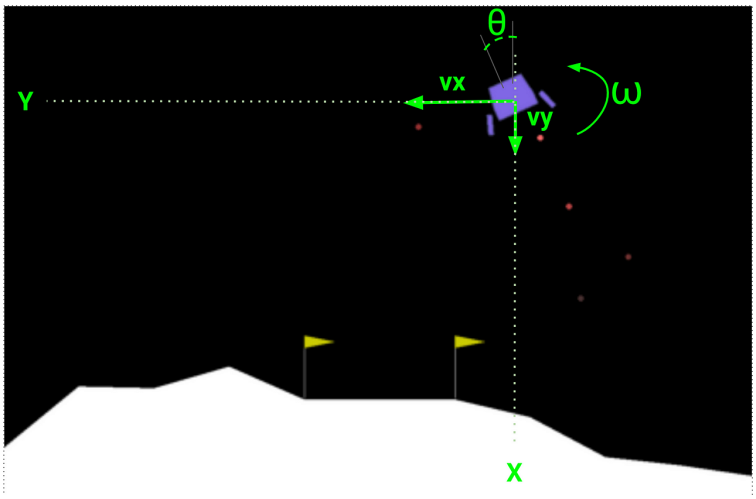
```
eps_decay_rate = 0.97
eps_end = 0.01
BATCH_SIZE = 64

agent = DDQNAgent(state_size, action_size, batch_size=BATCH_SIZE)
crs = np.zeros(n_episodes)
crs_recent = deque(maxlen=25)

for i_episode in range(1, n_episodes+1):
    env = RecordVideo(gym.make("LunarLander-v2", render_mode="rgb_array"), f"./DDQN/batch{BATCH_SIZE}/eps{i_episode}") if i_episode % 50 == 0 else gym.make("LunarLander-v2", render_mode="rgb_array")
    state, info = env.reset()
    done = False
    cr = 0
    action_count = 0
    while not done:
        action = agent.take_action(state, eps)
        next_state, reward, done, truncated, info = env.step(action)
        agent.experience_buffer.store_trans(state, action, next_state, reward, done or truncated)
        agent.update_params()
        state = next_state
        cr += reward
        action_count += 1
        if action_count % 5 == 0:
            agent.update_target_network()

    eps = max(eps * eps_decay_rate, eps_end)
    crs[i_episode - 1] = cr
    crs_recent.append(cr)
    if i_episode % 50 == 0:
        agent.save(f"q_net_batch{BATCH_SIZE}_eps{i_episode}.pt")

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

fig = plt.figure()
plt.plot(np.arange(len(crs)), crs)
plt.ylabel('Reward')
plt.xlabel('Episode')
plt.title(f"DDQN_batch{BATCH_SIZE}")
plt.savefig(f"DDQN_batch{BATCH_SIZE}.pdf")
```

```
plt.show()
os.system("zip -r DDQN64.zip DDQN/")
```

این بخش از کد، حلقه آموزش را برای DDQN با اندازه دسته ۶۴ اجرا می‌کند. در هر اپیزود، محیط بازنشانی می‌شود و عامل با استفاده از سیاست e-greedy اقدام می‌کند. تجربه‌های عامل در ExperienceReplay ذخیره می‌شوند و پارامترهای شبکه به‌روزرسانی می‌شوند. در هر ۵۰ اپیزود، مدل ذخیره می‌شود و در پایان، نتایج آموزش به صورت نمودار نمایش داده می‌شوند.

۱.۲

این محیط یکی از محیط‌های Box۲D در Gymnasium است که هدف آن شبیه‌سازی فرود یک فرودگر روی سطح ماه است. این محیط شامل یک بردار مشاهده (observation) هشت‌بعدی است که مختصات x و y، سرعت‌های خطی در x و y، زاویه، سرعت زاویه‌ای، و دو مقدار بولی که نشان می‌دهند آیا هر پا با زمین در تماس است یا نه، را شامل می‌شود.



شکل ۱: محیط Lunar lander

این محیط یک محیط پیاده‌سازی‌شده توسط Box۲D است که به شما امکان می‌دهد یک فرودگر را به طور مستقل کنترل کنید و آن را با موفقیت روی سطح ماه فرود بیاورید. فضای عمل (action) شامل چهار عمل گسسته است:

۱. عدم انجام کاری

۲. فعال کردن موتور سمت چپ

۳. فعال کردن موتور اصلی

۴. فعال کردن موتور سمت راست

فضای مشاهده (observation) شامل یک بردار هشت‌بعدی است که شامل موارد زیر است:

- مختصات فرودگر در x و y
- سرعت‌های خطی در x و y

- زاویه
- سرعت زاویه‌ای
- دو مقدار بولی که نشان می‌دهند هر پا فرودگر با زمین در تماس است یا نه

پاداش‌ها به گونه‌ای طراحی شده‌اند که فرود با موفقیت و استفاده بهینه از سوخت را تشویق کنند. جریمه‌ها برای برخورد با سطح با سرعت بالا، سقوط یا مصرف زیاد سوخت اعمال می‌شوند.

پاداش‌ها

- پس از هر گام ،(step) پاداشی اعطا می‌شود. مجموع پاداش یک قسمت جمع تمام پاداش‌ها برای همه گام‌های آن قسمت است.
- برای هر گام، پاداش با نزدیکتر شدن/دور شدن فرودگر به/از پد فرود افزایش/کاهش می‌یابد.
- پاداش با کندتر/سریعتر شدن حرکت فرودگر افزایش/کاهش می‌یابد.
- پاداش با کج شدن فرودگر (زاویه غیر افقی) کاهش می‌یابد.
- برای هر پا که با زمین تماس دارد، ۱۰ امتیاز پاداش افزایش می‌یابد.
- با هر فریم که یک موتور جانبی روشن است، 0.03 امتیاز کاهش می‌یابد.
- با هر فریم که موتور اصلی روشن است، 0.3 امتیاز کاهش می‌یابد.
- هر قسمت یک پاداش اضافی به ترتیب 100+ یا 100- امتیاز برای سقوط یا فرود ایمن دریافت می‌کند.
- یک قسمت به عنوان یک راه حل در نظر گرفته می‌شود اگر حداقل ۲۰۰ امتیاز کسب کند.

شروع و پایان هر قسمت (episode) به صورت زیر تعریف شده است.

شروع حالت

فرودگر از بالای مرکز صفحه با نیروی اولیه تصادفی به مرکز جرم خود شروع می‌کند.

خاتمه قسمت

قسمت به پایان می‌رسد اگر:

- فرودگر سقوط کند (بدنه فرودگر با ماه تماس بگیرد).
- فرودگر از محدوده صفحه خارج شود (مختصات x بزرگتر از ۱ شود).
- فرودگر بیدار نباشد. از مستندات ،Box۲D بدنه‌ای که بیدار نیست بدنه‌ای است که حرکت نمی‌کند و با هیچ بدنه دیگری برخورد نمی‌کند.

برای استفاده از محیط پیوسته، باید آرگومان continuous=True را به شکل زیر مشخص کنید:

```
import gymnasium as gym
env = gym.make(
    "LunarLander-v2",
    continuous: bool = False,
    gravity: float = -10.0,
    enable_wind: bool = False,
    wind_power: float = 15.0,
    turbulence_power: float = 1.5,
)
```

اگر continuous=True تنظیم شود، اعمال پیوسته (مربوط به قدرت موتور) استفاده می‌شود و فضای عمل Box(-1, +1, (2,), dtype=np.float32) خواهد بود. مختصات اول عمل، قدرت موتور اصلی و مختصات دوم، قدرت موتورهای جانبی را تعیین می‌کند. اگر آرایه np.array([main, lateral]) باشد، موتور اصلی کاملاً خاموش خواهد شد اگر main < 0 و قدرت موتور از 50% تا 100% برای 0 <= main <= 1 تنظیم می‌شود. به همین ترتیب، اگر -0.5 < lateral < 0.5 باشد، موتورهای جانبی روشن نمی‌شوند. اگر lateral < -0.5 باشد، موتور چپ و اگر lateral > 0.5 باشد، موتور راست روشن می‌شود. باز هم، قدرت موتور از 50% تا 100% بین -1 و -0.5 (و 0.5 و 1) تنظیم می‌شود.

gravity مقدار ثابت گرانش را تعیین می‌کند که بین 0 و -12 تنظیم می‌شود.

اگر enable_wind=True تنظیم شود، اثرات باد به فرودگر اعمال می‌شود. باد با استفاده از تابع

$$\tanh(\sin(2k(t+C)) + \sin(\pi k(t+C)))$$

تولید می‌شود. k به 0.01 تنظیم شده و C به طور تصادفی بین -9999 و 9999 نمونه‌برداری می‌شود.

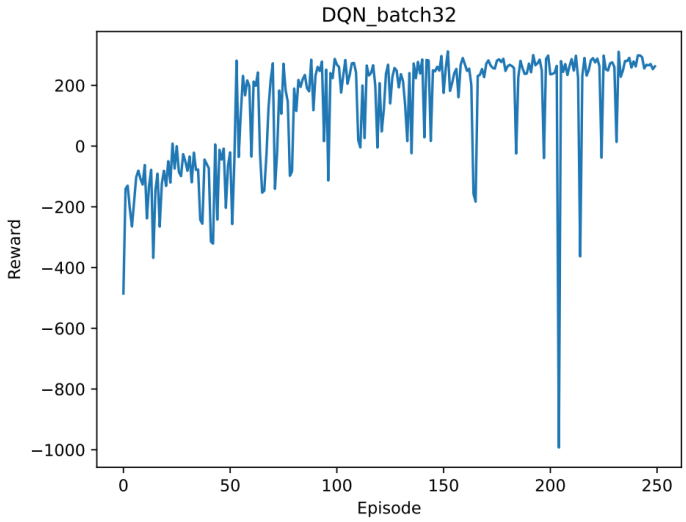
wind_power حداکثر اندازه باد خطی اعمال‌شده به وسیله نقلیه را تعیین می‌کند. مقدار پیشنهادی برای wind_power بین 0.0 و 20.0 است. turbulence_power حداکثر اندازه باد چرخشی اعمال‌شده به وسیله نقلیه را تعیین می‌کند. مقدار پیشنهادی برای turbulence_power بین 0.0 و 2.0 است.

تاریخچه نسخه‌ها

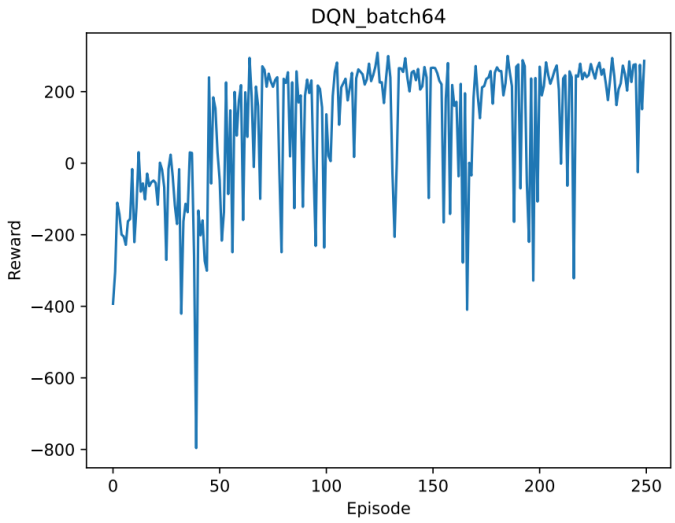
- نسخه v2: شمارش انرژی مصرفی و در نسخه 0.24، اضافه شدن توربولانس با پارامترهای wind power و turbulence_power.
- نسخه v1: اضافه شدن تماس پاها با زمین در بردار حالت؛ تماس با زمین 10 امتیاز پاداش و از دست دادن تماس 10 امتیاز جریمه؛ پاداش به 200 بازنرمالیزه شده؛ فشار اولیه تصادفی سخت‌تر.
- نسخه v0: نسخه اولیه.

۲.۲

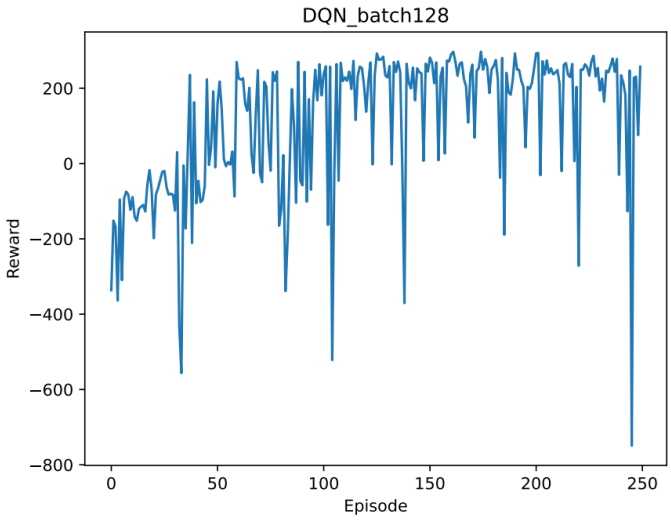
با پیاده سازی این کد برای batch size های متفاوت نمودار reward بر حسب تعداد episode به صورت زیر است.



شکل ۲: نمودار reward برای batch size=32



شکل ۳: نمودار reward برای batch size=64



شکل ۴: نمودار reward برای batch size=128

به طور کلی اندازه دسته در شبکه‌های عصبی عمیق Q (DQN) می‌تواند به طور قابل توجهی بر عملکرد و پایداری فرآیند آموزش تأثیر بگذارد. اندازه‌های دسته کوچک‌تر می‌توانند منجر به به‌روزرسانی‌های پایدارتر در یادگیری شوند زیرا واریانس بیشتری در تخمین گرادیان‌ها ایجاد می‌کنند که می‌تواند به خروج از کمینه‌های محلی کمک کند. با این حال، این واریانس افزایش یافته نیز می‌تواند منجر به ناپایداری در یادگیری شود زیرا به‌روزرسانی‌ها می‌توانند نویزی باشند. در مقابل، دسته‌های بزرگ‌تر تخمین‌های گرادیانی پایدارتر فراهم می‌کنند زیرا میانگین‌گیری بیشتری بر روی نمونه‌ها انجام می‌دهند که منجر به به‌روزرسانی‌های روان‌تر می‌شود. با این حال، ممکن است سرعت خروج از کمینه‌های محلی را کند کنند و به حافظه بیشتری نیاز داشته باشند. اندازه‌های دسته کوچک ممکن است در ابتدا سریع‌تر همگرا شوند به دلیل به‌روزرسانی‌های مکرر، اما به دلیل به‌روزرسانی‌های نویزی همگرایی ممکن است بهینه نباشد. اندازه‌های دسته بزرگ تمایل دارند در ابتدا به آرامی همگرا شوند اما در بلندمدت می‌توانند به یک سیاست بهینه‌تر برسند به دلیل به‌روزرسانی‌های پایدار و دقیق‌تر. دسته‌های کوچک می‌توانند به مدل کمک کنند تا بهتر تعمیم یابد و از بیش‌برازش جلوگیری کند. در مقابل، دسته‌های بزرگ ممکن است منجر به بیش‌برازش شوند زیرا مدل می‌تواند بیش از حد به نمونه‌های خاص در دسته بزرگ وابسته شود. اندازه‌های دسته کوچک می‌توانند از نظر محاسباتی ناکارآمد باشند به دلیل بهینه نبودن استفاده از سخت‌افزار (مثل GPUها). هر گام به‌روزرسانی ممکن است شامل داده کمتری باشد که منجر به کم‌استفاده شدن از توان پردازشی شود. در مقابل، دسته‌های بزرگ بر روی سخت‌افزاری مثل GPUها به دلیل استفاده بهتر از قابلیت‌های پردازش موازی کارآمدترند، اما ممکن است به حافظه و منابع محاسباتی بیشتری نیاز داشته باشند. محدودیت‌های حافظه یکی از ملاحظات عملی مهم است؛ اندازه‌های دسته بزرگ‌تر به حافظه بیشتری نیاز دارند. دسته‌های کوچک‌تر ممکن است در ابتدا به اکتشاف بهتر کمک کنند به دلیل به‌روزرسانی‌های نویزی، در حالی که دسته‌های بزرگ‌تر ممکن است به بهره‌برداری از سیاست‌های یادگرفته شده کمک کنند به دلیل به‌روزرسانی‌های پایدار. اندازه دسته بهینه اغلب نیاز به تعیین تجربی دارد و می‌تواند به مشکل خاص و محیط بستگی داشته باشد.

با توجه به نمودارها و توضیحات فوق مشاهده می‌شود که با دسته ۳۲ نتایج بهتری داریم چراکه تعداد قسمت‌های ما زیاد نیست و پایدار شدن دسته‌های بزرگتر نیاز به زمان و قست‌های بیشتری دارد.

حال به سراغ ارزیابی این سه حالت به کمک متریک regret می‌رویم. ابتدا تعریفی از این معیار را با هم مرور کنیم. این معیار برای اندازه‌گیری اوضاع در شرایط تصمیم‌گیری و در فضای یادگیری تقویتی به کار می‌رود. این معیار تفاوت بین پاداشی که توسط ایجنت دریافت شده و پاداشی که دریافت می‌شد، اگر ایجنت بهترین تصمیم را در هر اپیزود می‌گرفت، ارزیابی می‌کند. پس می‌توان آن را به

صورت رابطه ۱ فرموله کرد.

$$R_t = \mu^\star - \mu_a \quad (1) \qquad (۱)$$

در رابطه ۱، $\mu^\star$ پاداشی است که اگر ایجنت بهترین اکشن را انتخاب می‌کرد دریافت می‌کرد. در طرف دیگر μ_a پاداشی است که ایجنت با انتخاب کردن اکشن a در زمان t آن را دریافت کرده است.

حال اگر ما رابطه ۱ را در تمامی اپیزودها حساب کنیم، می‌توانیم regret تجمعی برای ایجنت را به دست آوریم. در واقع می‌توان گفت regret هزینه انتخاب نکردن بهترین تصمیم در هر اپیزود را به ما نشان می‌دهد.

مطابق نتایج، حالت batch size = 32 در ابتدا regret متوسطی دارد، سپس این معیار به صورت واضحی کم می‌شود چون پاداش‌ها در حال افزایش هستند (حدود اپیزود ۵۰) و در نهایت در اپیزودهای بالاتر این معیار به کمترین حالت خود می‌رسد از آنجایی که پاداش‌ها در حال افزایش هستند.

برای حالت batch size = 128 ما کمترین میزان regret را در فازهای مختلف شاهد هستیم چون بیشترین پاداش‌ها را برای این حالت دریافت می‌کنیم و می‌توان این‌طور توصیف کرد که در این حالت ایجنت به بهترین عملکرد خود نزدیک است که این مورد با توضیحات قبلی در رابطه با پایداری در دسته‌های بزرگتر و عملکرد مثبت در طولانی مدت همخوانی دارد.

۳.۲

تفاوت اصلی بین Deep Q-Network (DQN) و Double Deep Q-Network (DDQN) در روش به‌روزرسانی Q-values است. در DQN، شبکه عصبی برای تقریب Q-values استفاده می‌شود و برای به‌روزرسانی Q-values از شبکه فعلی استفاده می‌شود. این امر می‌تواند منجر به بیش‌برازش شود زیرا شبکه تمایل دارد به صورت ناپایدار مقادیر Q بزرگ‌تری را برای اقدامات خاص اختصاص دهد. در مقابل، DDQN از دو شبکه عصبی جداگانه استفاده می‌کند: یک شبکه برای انتخاب اقدامات (online network) و یک شبکه برای ارزیابی مقادیر Q (target network). این روش به کاهش بیش‌برازش کمک می‌کند زیرا اقدامات بر اساس شبکه آنلاین انتخاب می‌شوند اما مقادیر Q بر اساس شبکه هدف به‌روزرسانی می‌شوند. این رویکرد منجر به تخمین‌های پایدارتر و دقیق‌تر از مقادیر Q می‌شود و از برخی از مشکلات ناپایداری که در DQN مشاهده می‌شود جلوگیری می‌کند. به عبارت دیگر، DDQN با استفاده از دو شبکه جداگانه برای انتخاب و ارزیابی اقدامات، اطمینان حاصل می‌کند که مقادیر Q به صورت متوازن‌تری به‌روزرسانی می‌شوند و این منجر به بهبود عملکرد و پایداری در فرآیند یادگیری تقویتی می‌شود. این تفاوت اساسی بین DQN و DDQN، DDQN را به گزینه‌ای مطلوب‌تر برای بسیاری از کاربردهای یادگیری تقویتی تبدیل کرده است، به ویژه در محیط‌هایی که پیچیدگی و ناپایداری بالایی دارند.

حال این تفاوت را در کد بررسی می‌کنیم.

- در DQN، تنها یک شبکه عصبی (value_net) برای ارزیابی اقدامات استفاده می‌شود.
- در DDQN، دو شبکه عصبی مجزا وجود دارد: شبکه ارزش (value_net) و شبکه هدف (target_value_net). شبکه هدف به صورت دوره‌ای به‌روزرسانی می‌شود تا از بیش‌برازش جلوگیری شود.

```
1    agent.update_target_network()
2
```

این خط کد در DDQN استفاده می‌شود تا شبکه هدف به‌روزرسانی شود. در DQN، چنین به‌روزرسانی وجود ندارد.

- در هر دو مدل، (Experience Replay) برای ذخیره انتقال‌ها استفاده می‌شود. با این حال، در DDQN این حافظه با نام experience_buffer شناخته می‌شود.

```
    agent.experience_buffer.store_trans(state, action, next_state, reward, done or truncated)

```

این خط کد نشان‌دهنده استفاده از Experience Replay در DDQN است. در DQN، این حافظه با نام experience_replay شناخته می‌شود.

- در DQN، به‌روزرسانی پارامترها تنها با استفاده از شبکه ارزش انجام می‌شود.
- در DDQN، به‌روزرسانی پارامترها با استفاده از شبکه ارزش و شبکه هدف انجام می‌شود تا از مشکلات ناشی از تخمین‌های مغرضانه جلوگیری شود.

```
    q_targets_next = self.target_value_net(next_state_batch).detach().max(1)[0].unsqueeze(1)

```

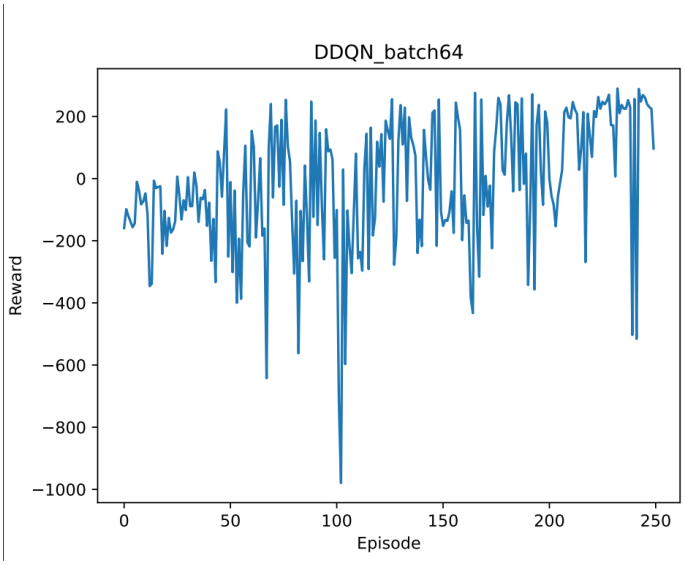
این خط کد نشان‌دهنده استفاده از شبکه هدف در DDQN برای به‌روزرسانی پارامترها است. در DQN، این به‌روزرسانی تنها با استفاده از شبکه ارزش انجام می‌شود.

- در DDQN، شبکه هدف هر چند گام به‌روزرسانی می‌شود (در اینجا هر ۵ اقدام).

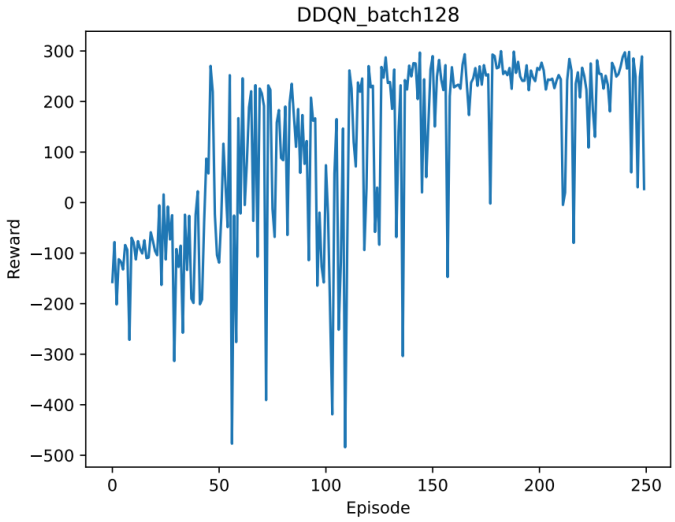
```
    if action_count % 5 == 0:
        agent.update_target_network()

```

این کد در DDQN استفاده می‌شود تا شبکه هدف هر ۵ اقدام به‌روزرسانی شود. در DQN، چنین فرکانسی برای به‌روزرسانی وجود ندارد.

نتایج پیاده‌سازی با استفاده از DDQN به صورت زیر است.



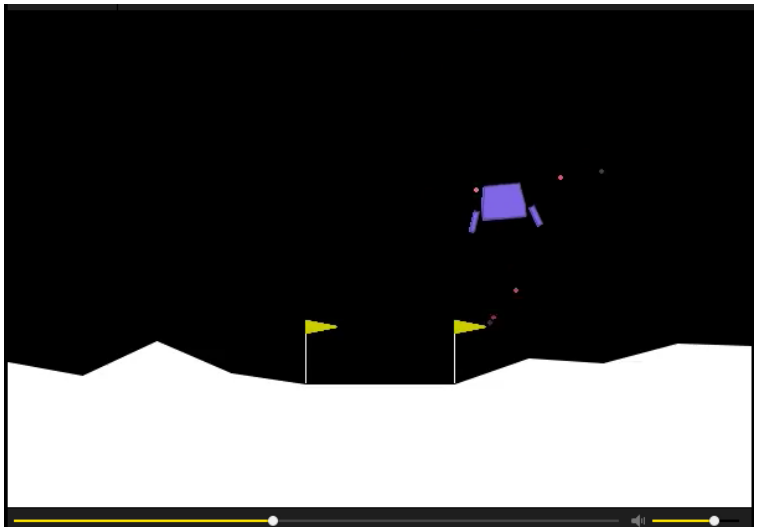
شکل ۵: نمودار reward برای batch size=64



شکل ۶: نمودار reward برای batch size=128

مشاهده می‌شود که عملکرد در دسته با اندازه ۱۲۸ بهتر است ولی به طور کلی ناپایداری در آن مشاهده می‌شود.

در مقایسه با حالت قبلی، سریعتر همگرا می‌شود و همچنین از نظر میزان پایداری در پاداش مقداری بهتر عمل می‌کند هرچند که هنوز نوسان زیادی در آن مشاهده می‌شود که دلیل آن می‌تواند پایین بودن تعداد episode ها باشد.



شکل ۷: نمونه خروجی ویدیویی

عملکرد ویدیویی این پیاده‌سازی در تعداد batch size های متفاوت در لینک‌های زیر بارگذاری شده است.

لینک Github

لینک Colab