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

یادگیری ماشین
پاسخ مینی پروژه دوم

نام و نام خانوادگی: سیدمحمد حسینی
شمارهٔ دانشجویی: ۹۹۰۱۳۹۹
تاریخ: مهرماه ۱۴۰۲
GitHub
Drive Google

فهرست مطالب

فهرست تصاویر

فهرست جداول

فهرست برنامه‌ها

۱ پرسش۲

۱.۱ قسمت اول

lunar lander فضاپیمایی است که برای فرود آمدن بر سطح ماه طراحی شده است و مأموریت آن فرود ایمن از مدار ماه به سطح ماه است. هدف اصلی این فضاپیما شامل طی کردن مسیر دقیق فرود و فرود ایمن و نرم است که نیازمند سیستم‌های راهنمایی پیشرفته، ناوبری دقیق و طراحی ساختاری قوی است. سیستم پیشرانه که شامل موتورهای اصلی فرود و موتورهای کنترل وضعیت جانبی و کوچکتر است که نقش مهمی در کنترل سرعت فرود و جهت‌گیری ایفا می‌کند. lunar lander همچنین دارای پایه‌های فرود برای کاهش اثر ضربه و تضمین پایداری هستند.

یادگیری تقویتی (RL) می‌تواند برای مسئله کنترل یک lunar lander استفاده شود. در این زمینه، RL شامل آموزش یک Agent برای یادگیری یک Policy بهینه برای فرود ایمن و کارآمد م lunar lander است. اجزای اصلی یک مسئله RL شامل فضای حالت، فضای عمل و سیستم پاداش است که این موارد به شرح زیر می‌باشند.

فضای حالت

فضای حالت تمام حالات ممکن را که فرودگر قمری می‌تواند در آن‌ها باشد، نشان می‌دهد. برای یک lunar lander فضای حالت می‌تواند شامل موارد زیر باشد:

- مختصات عمودی و افقی فرودگر.
- مولفه‌های سرعت در جهت‌های افقی و عمودی.
- زاویه فرودگر نسبت به عمود.
- سرعت زاویه‌ای فرودگر.
- وضعیت پایه‌های فرودگر

هر حالت نمایانگر شرایط فرودگر در یک گام زمانی معین است.

فضای عمل

فضای عمل تمام Actions ممکن را که Agent می‌تواند انجام دهد، نشان می‌دهد. برای یک lunar lander فضای عمل می‌تواند شامل موارد زیر باشد:

- توان موتور اصلی.
- توان موتور چپ.
- توان موتور راست.
- انجام ندادن هیچ کاری.

این Actions نیروی پیشرانه و جهت‌گیری فرودگر را کنترل می‌کنند.

سیستم پاداش

سیستم پاداش بازخوردی به Agent بر اساس Actions آن و تغییرات حالت ناشی از آن‌ها ارائه می‌دهد. برای یک lunar lander سیستم پاداش ممکن است به این صورت طراحی شود:

- پاداش مثبت برای دستیابی به فرود ایمن با سرعت کم.
- پاداش منفی برای فرودهای سخت یا سقوط‌ها.
- پاداش منفی برای مصرف سوخت زیاد.
- پاداش مثبت برای حفظ جهت‌گیری پایدار.
- پاداش منفی کوچک برای هر گام زمانی برای تشویق فرود سریع‌تر.

سیستم پاداش Agent را تشویق می‌کند تا هایPolicy را بیاموزد که منجر به فرودهای ایمن، کارآمد و پایدار شود. [۱]

۲.۱ قسمت دوم

ابتدا به بررسی بلاک‌های کد می‌پردازیم.

```python
# DQN
import torch.nn as nn
import torch.nn.functional as F

class DeepQNetwork(nn.Module):
    def __init__(self, state_size, action_size) -> None:
        super(DeepQNetwork, self).__init__()
        # TODO: define the architecture
        # NOTE: input=observation/state, output=action
        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):
        # TODO: forward propagation
```

```
        # NOTE: use ReLu for activation function in all layers
        # NOTE: last layer has no activation function (predict action)
        # ReLU is created in init, no need here
        x.to(device)
        x = self.net(x)
        return x

```

Code :۱ Deep Neural Network

کد فوق یک شبکه عصبی عمیق را ایجاد می‌کند که به موجب آن ۴ لایه خطی به همراه ۳ لایه غیر خطی ایجاد شده است تا ماتریس پاداش را پیشبینی کند.

```
# DQN agent
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class DQNAgent():
    # NOTE: DON'T change initial values
    def __init__(self, state_size, action_size, batch_size,
                 gamma=0.99, buffer_size=25000, alpha=1e-4):
        # network parameter
        self.state_size = state_size
        self.action_size = action_size

        # hyperparameters
        self.batch_size = batch_size
        self.gamma = gamma

        # experience replay
        self.experience_replay = ExperienceReplay(buffer_size)

        # network
        self.value_net = DeepQNetwork(state_size, action_size).to(device)

        # optimizer
        # TODO: create adam for optimizing network's parameter (learning rate=alpha)
        self.optimizer = optim.Adam(self.value_net.parameters(), lr=alpha)

    def take_action(self, state, eps=0.0):
        # TODO: take action using e-greedy policy
        # NOTE: takes action using the greedy policy with a probability of - 1 and a random action
     with a probability of
        # NOTE:
```

```
        self.value_net.eval()
        if len(state) != 8 :
            state = state[0]
        rand_eps = random.random()
        if rand_eps > eps:
            with torch.no_grad():
                # print(state)
                return torch.argmax(self.value_net(torch.tensor(state).to(device))).detach().cpu
    ().numpy()
        else:
            return np.random.randint(0, self.action_size)

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

        temp = []
        for indx in range(len(batch.state)):
            if len(batch.state[indx]) != 8:
                temp.append(batch.state[indx][0])
            else:
                temp.append(batch.state[indx])
        state_batch = torch.from_numpy(np.vstack(temp)).float().to(device)  # [8, 8]
        action_batch = torch.tensor(np.vstack(batch.action)).long().to(device) # [8, 1]
        next_state_batch = torch.from_numpy(np.vstack(batch.next_state)).float().to(device) # [8,
     8]
        reward_batch = torch.tensor(np.vstack(batch.reward)).float().to(device) # [8, 1]
        done_batch = torch.tensor(np.vstack(batch.done)).to(device)

        # calculate loss w.r.t DQN algorithm
        self.value_net.train()
        # STEP1
        q_expected = self.value_net(state_batch).gather(1, action_batch)
        # TODO: compute the expected Q values [y]
        # STEP2
        # TODO: compute Q values [Q(s_t, a)]
        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*1))
        # 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 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)

```

Code : ۲ Agent Class

در این بخش، کلاس `DQNAgent` تعریف می‌شود که Agent مربوط به DQN را پیاده‌سازی می‌کند. این کلاس مسئول مدیریت شبکه عصبی، اجرای Policy ϵ-Greedy ، به‌روزرسانی پارامترهای شبکه و ذخیره و بارگذاری مدل است.

۱.۲.۱ تعریف متغیرها و مقادیر اولیه

در ابتدای کلاس، مقادیر اولیه و پارامترهای شبکه تعریف می‌شوند. این شامل اندازه State (`state_size`)، اندازه Action (`action_size`)، اندازه Batch (`batch_size`)، (`gamma`)، اندازه Buffer (`buffer_size`) و نرخ یادگیری (`alpha`) است.
در ادامه شبکه عصبی (`DeepQNetwork`) برای پیش‌بینی مقادیر Q و بهینه‌ساز `Adam` با نرخ یادگیری α تعریف می‌شوند.

۲.۲.۱ متد `take_action`

این تابع برای انتخاب اقدام با استفاده از Policy ϵ-Greedy استفاده می‌شود. با احتمال $\epsilon - 1$، اقدام بهینه انتخاب می‌شود و با احتمال ϵ، یک اقدام تصادفی انتخاب می‌شود.

۳.۲.۱ متد `update_params`

این تابع پارامترهای شبکه عصبی را به‌روزرسانی می‌کند. اگر تعداد تجربیات کمتر از `batch_size` باشد، تابع متوقف می‌شود. در ادامه تجربیات نمونه‌گیری شده و به فرمت مناسب برای استفاده در شبکه عصبی تبدیل می‌شوند.
حال مقادیر Q مورد انتظار (`q_expected`) و سپس مقادیر هدف Q (`q_targets`) محاسبه می‌شوند. سپس با استفاده از (MSE) تابع هزینه محاسبه شده و مدل بهینه‌سازی می‌شود.

```
# training phase

# 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)
        # Modify the unpacking to handle the extra value if present
        result = env.step(action)
        if len(result) == 5:
            next_state, reward, done, truncated, info = result
        else:
            next_state, reward, done, info = result
        # 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
        # print("next" ,next_state)
        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"q_net_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))

```

Code :۳ Train

Training ۴.۲.۱

در این بخش، کدهای مربوط به بخش Training توضیح داده می‌شود. این فاز شامل ایجاد Agent، اجرای حلقه آموزش، انتخاب Actions، به‌روزرسانی پارامترها، و ذخیره مدل است.

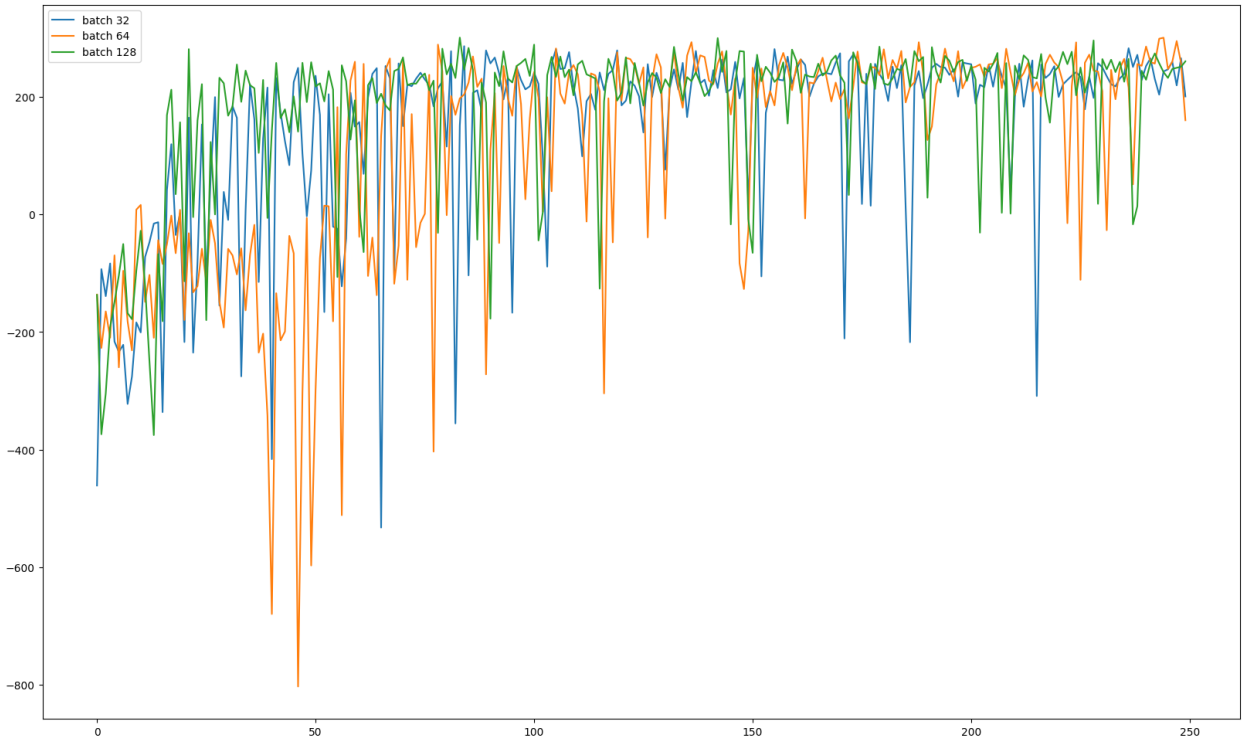
در گام نخست Agent مد نظر ما با مقادیر اولیه متناسب با مسئله ایجاد می‌شود. در ادامه متغیر crs برای ذخیره پاداش‌های تجمعی هر اپیزود و crs_recent برای ذخیره پاداش‌های تجمعی اخیر تعریف می‌شود.

حلقه آموزش برای تعداد اپیزودهای مشخص شده (n_episodes) اجرا می‌شود تا Agent پاسخ بهینه به سمئله را پیدا کند. در این حلقه آموزش به ازای هر اپیزود، محیط و حالت اولیه مقداردهی می‌شوند. هر ۵۰ اپیزود، محیط برای RecordVideo تنظیم می‌شود.

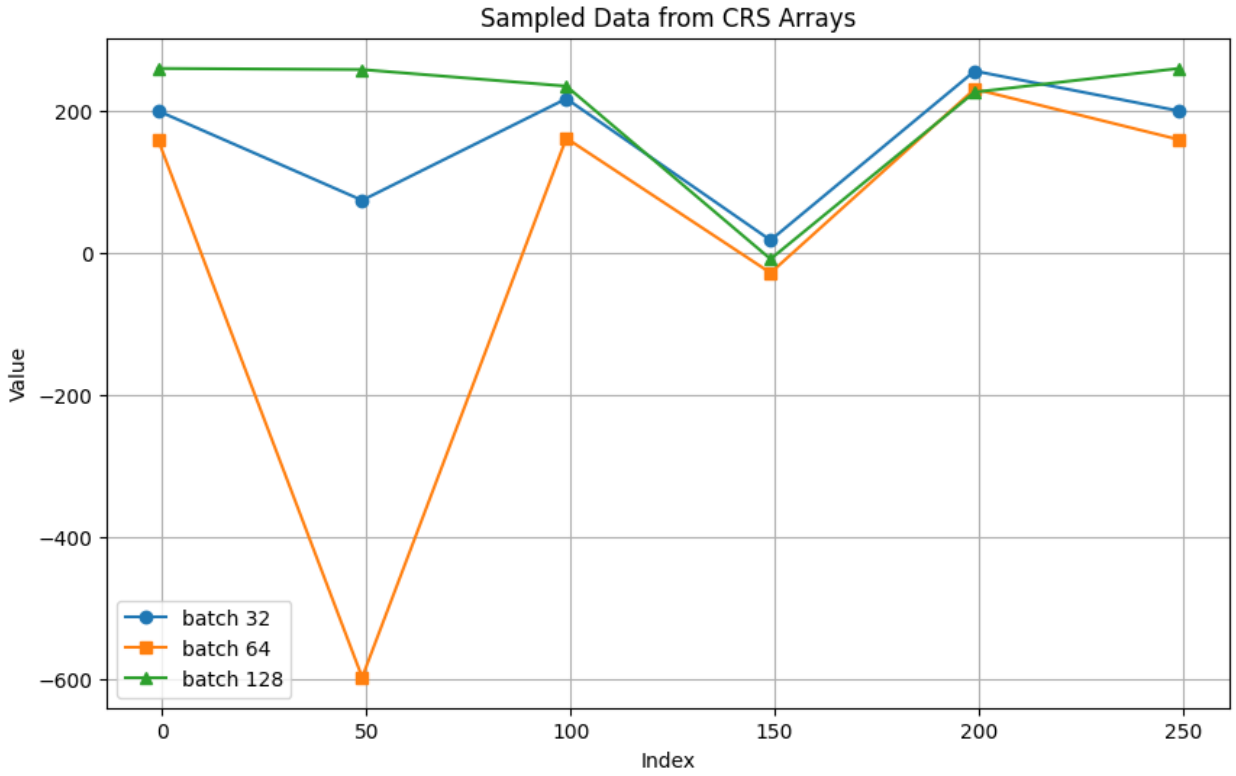
تا زمانی که اپیزود به پایان نرسیده است، حلقه steps اجرا می‌شود. در هر گام، محیط رندر می‌شود و Agent یک اقدام انتخاب می‌کند. نتیجه اقدام انجام شده دریافت می‌شود و انتقال در تجربه‌های Agent ذخیره می‌شود. پارامترهای Agent به‌روزرسانی می‌شوند و حالت فعلی و پاداش‌های تجمعی اپیزود به‌روز می‌شوند. پس از پایان اپیزود، مقدار epsilon کاهش می‌یابد تا به تدریج Agent کمتر به Actions تصادفی روی بیاورد. پاداش‌های تجمعی اپیزود جاری و پاداش‌های تجمعی اخیر به‌روز می‌شوند. هر ۵۰ اپیزود، مدل Agent ذخیره می‌شود.

نتایج ۵.۲.۱

نمودارهای زیر برای نمایش نتایج حاصله از آموزش مدل رسم شده‌اند.



شکل ۱: نتایج به ازای تمام اپیزودها



شکل ۲: نتایج به ازای هر ۵۰ اپیزود

همانطور که انتظار می‌رفت نتاج به ازای بچ ۱۲۸ زودتر همگرا شده است اما به ازای بچ ۶۴ نتایج همگرایی ضعیفتری داشته است نسبت به حالتی که بچ ۳۲ بوده است. بنابراین بهترین حالت مربوط به بچ۱۲۸ بوده و به منظور توجه به سرعت همگرایی در اپیزود ۵۰ بهترین مدل انتخاب می‌شود.
کد اجرایی به منظور ذخیره ویدیو از آموزش Agent همراه با یک ارور بوده که امکان حل آن وجود نداشت. ارور در فایل نوتبوک موجود می‌باشد.

۳.۱ قسمت سوم

در یادگیری تقویتی، DQN و DDQN دو روش محبوب برای یادگیری مقادیر-Q هستند. هر دو روش به منظور بهبود عملکرد Agent در انتخاب بهترین Actions در یک محیط مشخص توسعه داده شده‌اند. در اینجا تفاوت‌ها و شباهت‌های کلیدی بین DQN و DDQN را بررسی می‌کنیم.

۱.۳.۱ DQN

- **معماری:** DQN از یک شبکه عصبی عمیق برای تقریب مقادیر-Q استفاده می‌کند. این شبکه عصبی ورودی‌هایی شامل حالت‌های محیط را دریافت کرده و مقادیر-Q مربوط به هر اقدام ممکن را تولید می‌کند.

- **به‌روزرسانی :مقادیر-Q** در DQN، از یک شبکه هدف استفاده می‌شود که کپی‌ای از شبکه اصلی (Q-Network) است و در فواصل زمانی منظم به‌روزرسانی می‌شود. این کار به پایداری یادگیری کمک می‌کند.
- **معادله به‌روزرسانی:** معادله به‌روزرسانی مقادیر-Q در DQN به صورت زیر است:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q'(s',a') - Q(s,a) \right)$$

که در آن $Q'(s',a')$ مقدار Q پیش‌بینی شده توسط شبکه هدف است.

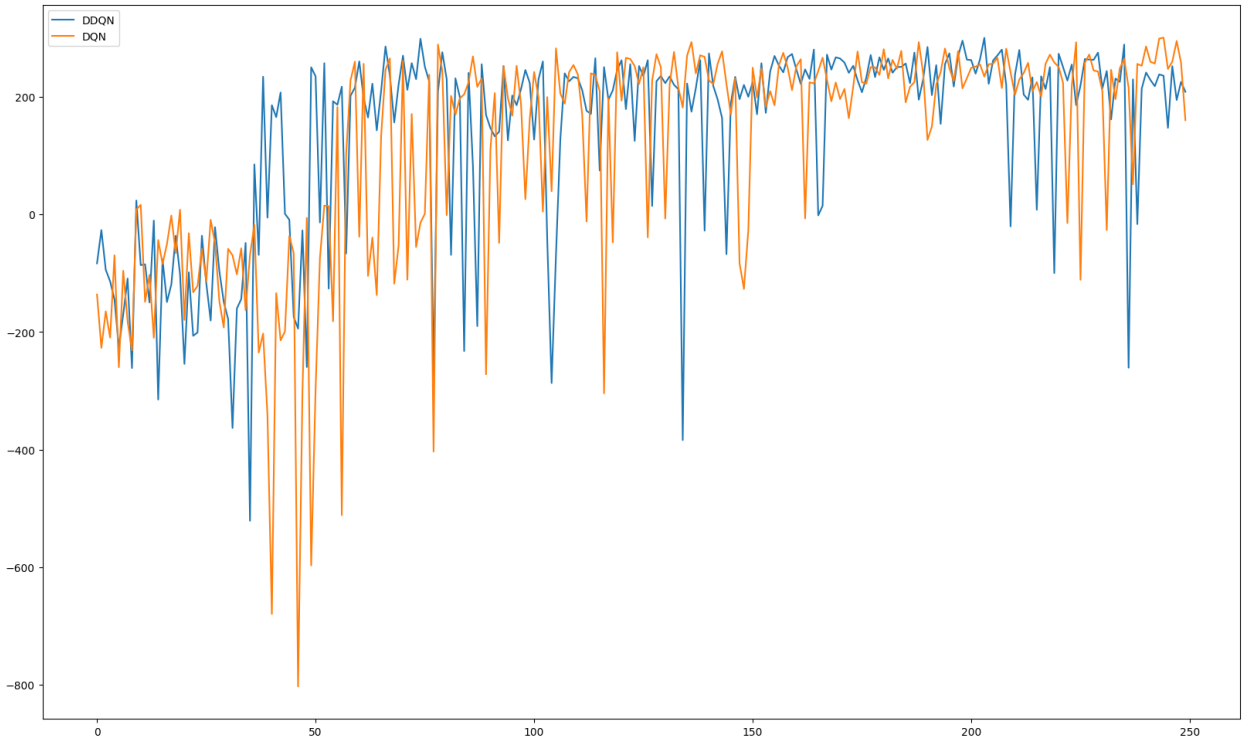
- **Over fitting** در DQN، به دلیل استفاده از $\max_{a'} Q(s',a')$ برای به‌روزرسانی ،مقادیر-Q احتمال Over fittingوجود دارد که می‌تواند منجر به پایداری کمتر در یادگیری شود.

DDQN ۲.۳.۱

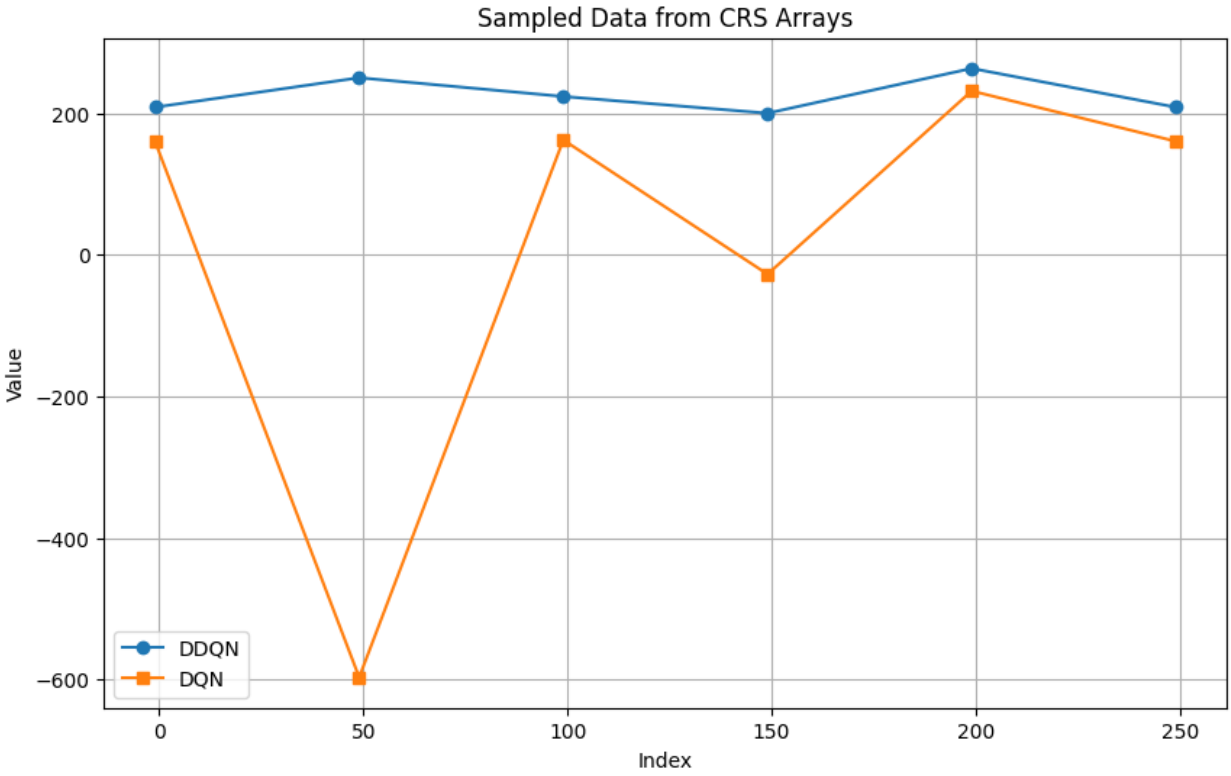
- **معماری:** معماری شبکه عصبی در DDQN مشابه DQN است، با این تفاوت که در DDQN دو شبکه Q به‌صورت جداگانه به‌روزرسانی می‌شوند: یک شبکه Q اصلی و یک شبکه Q هدف.
- **به‌روزرسانی :مقادیر-Q** در DDQN، به‌روزرسانی مقادیر-Q به گونه‌ای اصلاح شده است که از Over fittingجلوگیری شود. این کار با استفاده از شبکه Q اصلی برای انتخاب Actions و شبکه Q هدف برای محاسبه مقادیر-Q انجام می‌شود.
- **معادله به‌روزرسانی:** معادله به‌روزرسانی مقادیر-Q در DDQN به صورت زیر است:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q'(s', \arg\max_{a'} Q(s',a')) - Q(s,a) \right)$$

که در آن $Q(s',a')$ مقدار Q پیش‌بینی شده توسط شبکه Q اصلی است و $Q'(s', \arg\max_{a'} Q(s',a'))$ مقدار Q پیش‌بینی شده توسط شبکه هدف با استفاده از اقدام انتخاب شده توسط شبکه Q اصلی است.



شکل ۳: مقایسه DQN و DDQN



شکل ۴: مقایسه DQN و DDQN

همانطور که انتظار می‌رفت مدل DDQN نتیجه بهتر و پایدارتری کسب کرده است که این مسئله ناشی از تغییر روند آموزش شبکه عصبی می‌باشد. پایداری مدل DDQN در فرایند همگرایی و نتیجه کلی Agent تاثیر مهمی داشته است.

مراجع

[۱] Xu. Sophia "Final project for data science، course. ۲۰۲۴. Accessed: ۰۵-۰۷-۲۰۲۴.