Московский государственный технический университет им. Н.Э. Баумана

Кафедра «Системы обработки информации и управления»

Лабораторная работа №6

по дисциплине

«Обучение на основе DQN»

Выполнила:

студентка группы ИУ5И-24М

Лю Бовэнь

Москва — 2024 г.

Цель лабораторной работы

Ознакомление с базовыми методами обучения с подкреплением на основе глубоких Q-сетей.

Задание

- На основе рассмотренных на лекции примеров реализуйте алгоритм DQN.
- В качестве среды можно использовать классические среды (в этом случае используется полносвязная архитектура нейронной сети).
- В качестве среды можно использовать игры Atari (в этом случае используется сверточная архитектура нейронной сети).
- В случае реализации среды на основе сверточной архитектуры нейронной сети +1 балл за экзамен.

Основной раздел кода

```
import gym

import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

import random

from collections import deque

import matplotlib.pyplot as plt



# 设置设备

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



# 定义 Q 网络

class QNetwork(nn.Module):

    def __init__(self, state_size, action_size):

        super(QNetwork, self).__init__()

        self.fc1 = nn.Linear(state_size, 64)

        self.fc2 = nn.Linear(64, 64)

        self.fc3 = nn.Linear(64, action_size)



    def forward(self, x):

        x = torch.relu(self.fc1(x))

        x = torch.relu(self.fc2(x))

        x = self.fc3(x)

        return x
```

```
# 经验回放缓冲区
class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def add(self, experience):
        self.memory.append(experience)

    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().to(device)
        actions = torch.from_numpy(np.vstack([e[1] for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e[2] for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e[3] for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e[4] for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

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

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

# DQN 智能体
class DQNAgent:
    def __init__(self, state_size, action_size, 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.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 and len(self.memory) > self.memory.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)

# 训练 DQN 智能体
def dqn(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, _ = 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 % 100 == 0:

        print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores_window):.2f}')



    if np.mean(scores_window) >= 195.0:

        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


# 创建环境和智能体
env = gym.make('CartPole-v1')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n

agent = DQNAgent(state_size=state_size,
                 action_size=action_size,
                 buffer_size=int(1e5),
                 batch_size=64,
                 gamma=0.99,
                 lr=5e-4,
                 tau=1e-3,
                 update_every=4)

# 训练智能体
scores = dqn()

# 绘制分数
plt.figure()
plt.plot(np.arange(len(scores)), scores)
plt.xlabel('Episode')
plt.ylabel('Score')
plt.show()
```

Результат

```
Episode 100     Average Score: 17.59
Episode 200     Average Score: 13.73
Episode 300     Average Score: 29.28
Episode 400     Average Score: 55.18
Episode 500     Average Score: 154.61
Episode 600     Average Score: 167.40
Episode 700     Average Score: 172.06
Episode 800     Average Score: 168.20
Episode 900     Average Score: 165.70
Episode 1000    Average Score: 165.24
```

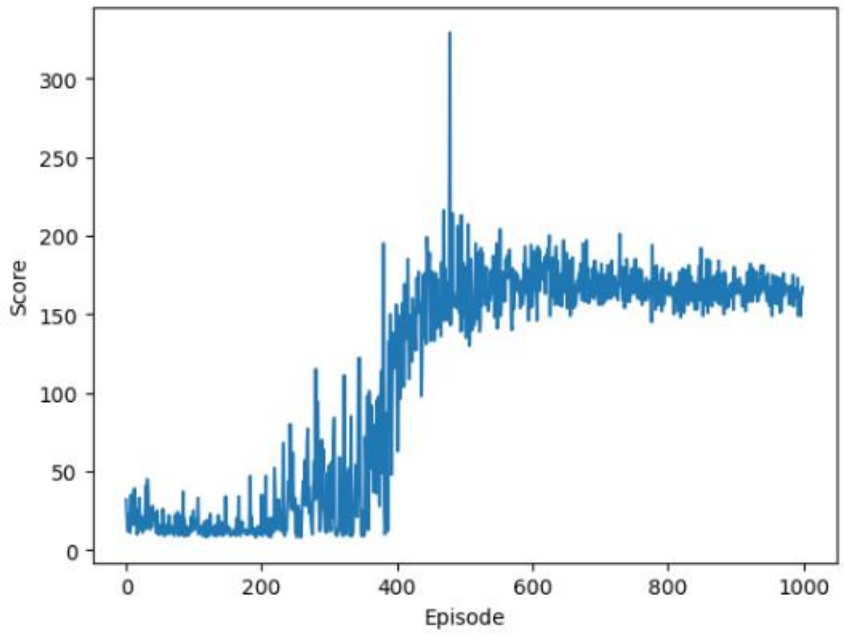

Рис 1. Результат.