中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	21311274	姓名	林宇浩

一、 实验题目

用 Deep Q-learning Network(DQN)玩 CartPole-v1 游戏

二、 实验内容

1. 算法原理

当涉及到解决具有大型状态空间的马尔可夫决策过程问题时，传统的强化学习方法往往面临着维度灾难和样本效率低下的挑战。DQN 通过结合深度神经网络和 Q-learning 算法的优势，克服了这些问题。其利用深度神经网络来近似动作的价值函数 Q(s, a)，其中 s 表示状态，a 表示动作。通过训练神经网络，DQN 能够通过输入状态 s 来预测每个动作 a 的 Q 值，从而了解每个动作对于当前状态的优劣程度。为了提高训练的稳定性和样本效率，DQN 采用了两个重要的技术：经验回放和目标网络。

经验回放是一种重要的数据存储和训练方法。在每个时间步，DQN 将经验元组，其包括当前状态、采取的动作、获得的奖励、下一个状态，存储在一个经验回放缓冲区中。然后，从缓冲区中随机选择一批经验样本进行训练，这样可以打破时间上的相关性，减少样本间的相关性，提高训练的稳定性。

目标网络是为了解决 DQN 中的估计偏差问题。DQN 使用两个神经网络：主网络和目标网络。主网络用于选择动作，而目标网络用于计算目标 Q 值。为了减少目标 Q 值的变化和提高稳定性，目标网络的参数定期从主网络中复制，这样在训练过程中目标 Q 值的更新与当前 Q 值的变化之间有一定的延迟。

训练过程采用了 Q-learning 算法的基本原理，通过优化损失函数来使预测的 Q 值逐渐逼近目标 Q 值。DQN 使用均方差损失函数来衡量预测值与目标值之间的差异，并通过梯度下降优化算法来更新神经网络的参数。DQN 能够通过迭代学习到最优的动作价值函数，从而在复杂的状态空间中做出最优的动作选择。

2. 伪代码

初始化经验回放缓冲区 D
初始化主网络 Q 网络参数 θ 和目标网络 Q'网络参数 θ'
设置超参数
随机初始化状态 s

对于每个训练轮次：
重置环境，获取初始状态 s
对于每个时间步 t：
使用 ε -greedy 策略选择动作 a：
with 概率 ε 选择随机动作
否则，选择 a = argmax Q(s, a; θ) （根据主网络的 Q 值选择最优动作）
执行动作 a，观察下一个状态 s'和奖励 r
将经验元组(s, a, r, s')存储到经验回放缓冲区 D

从 D 中随机选择一批经验样本(s, a, r, s')进行训练：
计算目标 Q 值 target：
if s'是终止状态：
target = r
else:
target = r + γ * max Q'(s', a'; θ') （使用目标网络的 Q 值估计）
计算预测 Q 值 prediction：
prediction = Q(s, a; θ)
计算损失函数 loss：
loss = (target - prediction)^2
使用梯度下降法更新主网络参数 θ，使得 loss 最小化
每隔 C 步，更新目标网络参数 θ' = θ
降低 ε 的概率（ε -greedy 策略）或增加训练轮次，以便逐渐减少探索并增加利用

3. 关键代码展示（带注释）

```
import gym
import argparse
import numpy as np
import torch
import torch.nn.functional as F
from torch import nn, optim
import matplotlib.pyplot as plt


class QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        # TODO write another linear layer here with
        # inputsize "hidden_size" and outputsize "output_size"
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.Tensor(x)
        x = F.relu(self.fc1(x))
        # TODO: calculate output with layer fc2
        x = self.fc2(x)
        return x


class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = []
        self.capacity = capacity
```

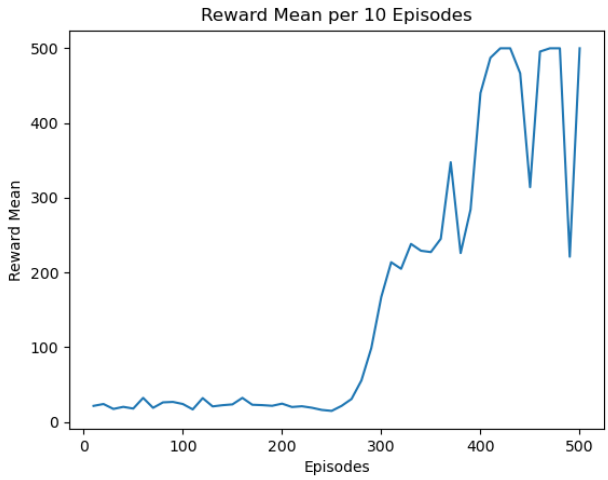
```
    def len(self):
        return len(self.buffer)

    def push(self, *transition):
        if len(self.buffer) == self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, n):
        index = np.random.choice(len(self.buffer), n)
        batch = [self.buffer[i] for i in index]
        return zip(*batch)

    def clean(self):
        self.buffer.clear()


class DQN:
    def __init__(self, env, input_size, hidden_size, output_size):
        self.env = env
        self.eval_net = QNet(input_size, hidden_size, output_size)
        self.target_net = QNet(input_size, hidden_size, output_size)
        self.optim = optim.Adam(self.eval_net.parameters(), lr=args.lr)
        self.eps = args.eps
        self.buffer = ReplayBuffer(args.capacity)
        self.loss_fn = nn.MSELoss()
        self.learn_step = 0

    def choose_action(self, obs):
        # epsilon-greedy
        if np.random.uniform() <= self.eps:
            # TODO: choose an action in [0, self.env.action_space.n) randomly
            action = np.random.randint(0, self.env.action_space.n)  # 以随机方式
选择一个动作，范围在[0, self.env.action_space.n)之间
        else:  # 如果随机数大于 epsilon
            # TODO: get an action with "eval_net" according to observation
"obs"
            # 根据观测值 "obs" 使用 "eval_net" 获取一个动作
            obs = np.expand_dims(obs, 0)
            obs = torch.Tensor(obs)
            q_values = self.eval_net(obs)  # 使用神经网络模型 eval_net 来计算所有可能
动作的 Q 值
            action = torch.argmax(q_values).item()  # 选择具有最高 Q 值的动作
        return action

    def store_transition(self, *transition):
        self.buffer.push(*transition)

    def learn(self):
        if self.eps > args.eps_min:
            self.eps *= args.eps_decay

        if self.learn_step % args.update_target == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())
        self.learn_step += 1

        obs, actions, rewards, next_obs, dones =
self.buffer.sample(args.batch_size)
        actions = torch.LongTensor(actions)  # LongTensor to use gather latter
        dones = torch.FloatTensor(dones)
        rewards = torch.FloatTensor(rewards)

        # TODO: calculate q_eval with eval_net and q_next with target_net
        # 使用 eval_net 计算 q_eval 和用 target_net 计算 q_next
        obs = torch.Tensor(obs)
        q_eval = self.eval_net(obs).gather(1, actions.unsqueeze(1))
        next_obs = torch.Tensor(next_obs)
        q_next = self.target_net(next_obs).detach().max(1)[0]
```

```
        # TODO: q_target = r + gamma * (1-dones) * q_next
        q_target = rewards + args.gamma * (1 - dones) * q_next

        # TODO: calculate loss between "q_eval" and "q_target" with loss_fn
        # 使用 loss_fn 计算"q_eval"和"q_target"之间的损失
        loss = self.loss_fn(q_eval, q_target.unsqueeze(1))

        # TODO: optimize the network with self.optim
        # 使用 self.optim 优化网络
        self.optim.zero_grad()
        loss.backward()
        self.optim.step()


def main():
    env = gym.make(args.env, render_mode='human')
    o_dim = env.observation_space.shape[0]
    a_dim = env.action_space.n
    agent = DQN(env, o_dim, args.hidden, a_dim)

    reward_history = []  # 存储平均值
    reward_mean = 0  # 近 10 局 reward 均值

    for i_episode in range(args.n_episodes):
        obs = env.reset()[0]
        episode_reward = 0
        done = False
        step_cnt = 0
        while not done and step_cnt < 600:
            step_cnt += 1
            env.render()
            action = agent.choose_action(obs)
            next_obs, reward, done, info, _ = env.step(action)
            agent.store_transition(obs, action, reward, next_obs, done)
            episode_reward += reward
            obs = next_obs
            if agent.buffer.len() >= args.capacity:
                agent.learn()
        # print(f"Episode: {i_episode}, Reward: {episode_reward}")

        reward_mean += episode_reward
        if (i_episode + 1) % 10 == 0:
            reward_mean = reward_mean / 10  # 计算近 10 局 reward 均值
            reward_history.append(reward_mean)
            print(f"Episode: {i_episode + 1}, Reward: {reward_mean}")  # 输出近
10 局 reward 均值
            reward_mean = 0

    # 绘制 500 局游戏的近 10 局 reward 均值曲线图
    plt.plot(range(10, args.n_episodes + 1, 10), reward_history)
    plt.xlabel('Episodes')
    plt.ylabel('Reward Mean')
    plt.title('Reward Mean per 10 Episodes')
    plt.show()


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--env", default="CartPole-v1", type=str)
    parser.add_argument("--lr", default=1e-3, type=float)
    parser.add_argument("--hidden", default=64, type=int)
    parser.add_argument("--n_episodes", default=500, type=int)
    parser.add_argument("--gamma", default=0.99, type=float)
    parser.add_argument("--log_freq", default=100, type=int)
    parser.add_argument("--capacity", default=5000, type=int)
    parser.add_argument("--eps", default=1.0, type=float)
    parser.add_argument("--eps_min", default=0.05, type=float)
```

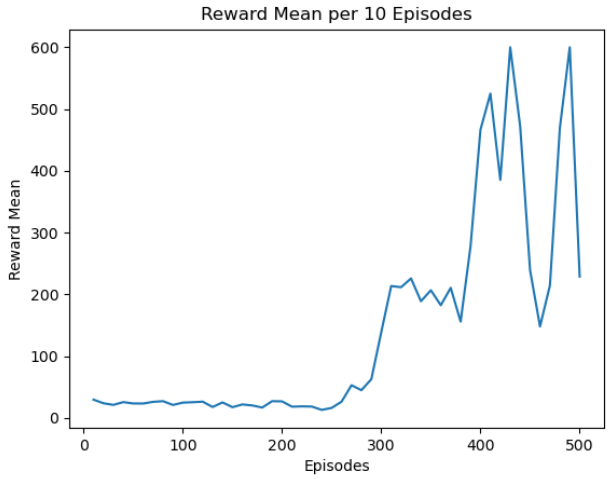
```
    parser.add_argument("--batch_size", default=128, type=int)
    parser.add_argument("--eps_decay", default=0.999, type=float)
    parser.add_argument("--update_target", default=100, type=int)
    args = parser.parse_args()
    main()
```

三、 实验结果及分析

500 局游戏的近 10 局 reward 均值曲线图



将 step_cnt 上限设为 600 后

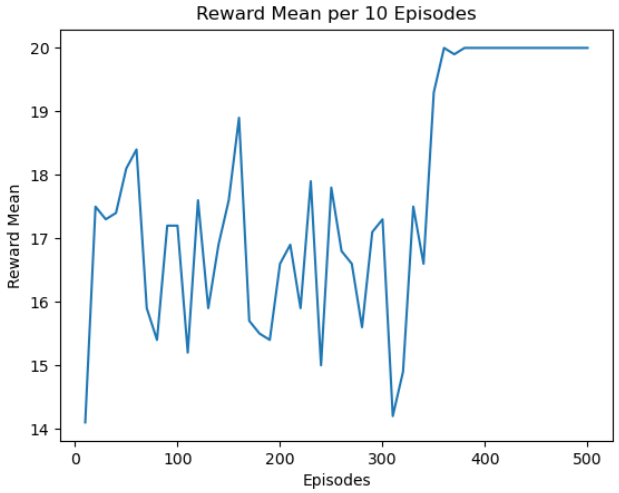


可以看到近 10 局 reward 均值达到 500 后并没有一直保持最大值 500，而是后续还有下降和回升。设成 600 时同样。

分析原因可能是因为当训练开始时，智能体的策略是比较随机的，并且它对环境的了解非常有限。随着训练的进行，智能体逐渐学习到更好的策略，并且其在环境中获得的回报也会相应增加。当智能体在训练的早期阶段达到一个较高的回报时，这很可能是因为它在某些情况下获得了较好的结果，或者仅仅是由于随机性导致的偶然事件。然而，这并不能保证智能体将一直保持这种高水平的回报。随着训练的继续，智能体会探索更多的状态和动作，并试图

寻找更优的策略。这可能会导致其尝试一些不同的动作选择，有时可能导致在短期内获得较低的回报。但是，通过学习和优化过程，智能体有望找到更好的策略，从而提高长期回报的期望值。所以，虽然智能体在训练的早期阶段可能会达到较高的回报，但在训练的过程中，由于策略的优化和环境的变化，回报可能会出现波动或下降。最终，训练成功后，智能体有望找到更优的策略，并达到更高的长期回报。

将 step_cnt 上限设为 20 后



将上限改小后，学习内容变得更简单，以帮助我们试验上述分析。可以看到，当近 10 局 reward 均值刚达到 20 后又有一个小的下降，之后回升后一直保持最大值 20，这很可能就是达到了最终训练效果，之后不会再出现回落。所以整个训练过程中，reward 值应该是先达到较高的回报后发生波动，最后稳定在最高的回报不会再回落。

四、 参考资料

课件 ppt