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	专业(方向)	学号	姓名
计科三班	计算机科学与技术	21307185	张礼贤

一、实验题目

- 用Deep Q-learning Network(DQN)玩CartPole-v1游戏，框架代码已经给出，只需要补充核心代码片段（'TODO'标记）
  - QNet 补充一个线性层
  - Choose_action 补充ε-greedy策略代码
  - Learn 补充Q值的计算，损失值计算，网络反向传播代码

二、实验内容

1.算法原理

Deep Q-Learning Network (DQN) 是一种强化学习算法，用于解决具有离散动作空间的马尔可夫决策过程（Markov Decision Process，MDP）问题。下面详细介绍DQN算法的原理和步骤：

1. 强化学习和Q-Learning简介：

   - 强化学习是一种机器学习方法，通过智能体与环境的交互学习最优策略来最大化累积奖励。
   - Q-Learning是一种基于值函数的强化学习方法，通过估计每个状态动作对的价值（Q值），并更新Q值函数来学习最优策略。

2. Q函数和Q值：

- 在强化学习中，Q函数表示状态和动作的映射关系，用于估计执行某个动作后能获得的累积奖励。
- Q值表示在给定状态下，采取某个动作的预期累积奖励。它可以用于选择最优动作和评估当前策略的好坏。

3. DQN的核心思想：

- DQN通过使用神经网络（深度网络）来逼近Q函数，将状态作为输入，输出每个动作的Q值。
- DQN使用经验回放和目标网络的技术来提高训练的稳定性和收敛性。

4. DQN算法步骤：

1. 初始化环境和网络：

   - 初始化马尔可夫决策过程（MDP）环境，如CartPole-v1游戏。
   - 初始化两个神经网络：评估网络（eval_net）和目标网络（target_net）。
   - 初始化回放缓冲区用于存储经验元组（状态、动作、奖励、下一个状态）。

2. 经验采样和存储：

   - 在每个时间步，根据当前策略（如ε-greedy）选择动作并与环境进行交互。
   - 将经验元组（状态、动作、奖励、下一个状态）存储到回放缓冲区中。

3. 网络训练：

   - 从回放缓冲区中随机采样一批经验元组（mini-batch）。
   - 使用评估网络（eval_net）计算当前状态的Q值。
   - 使用目标网络（target_net）计算下一个状态的最大Q值。
   - 根据贝尔曼方程更新Q值估计：Q(s,a) = r + γ * max(Q(s',a'))。
   - 使用均方误差（MSE）损失函数计算损失。
   - 使用反向传播算法更新评估网络的参数。

4. 目标网络更新：

   - 每隔一定的时间步，将评估网络的参数复制给目标网络。
   - 目标网络的参数保持固定一段时间，以提供稳定的目标值。

5. 控制策略改进：

- 根据当前的ε-greedy策略选择动作。
- 随着训练的进行，逐渐降低随机探索的概率ε。

6. 重复步骤b至e直到收敛或达到预设的训练轮数。

5. 算法优化和技巧：

- 经验回放（Experience Replay）：随机采样回放缓冲区中的经验样本，减少数据间的相关性，提高训练的稳定性。
- 目标网络（Target Network）：使用目标网络来计算目标Q值，减少目标值和估计值之间的相关性。
- ε-greedy策略：在训练初期采取随机动作，随着训练的进行逐渐降低随机探索的概率，使智能体更多地依赖已学习到的知识。
- 深度神经网络：使用深度网络来逼近Q函数，能够处理高维状态空间和复杂的决策任务。

总结：
DQN算法通过结合深度神经网络和Q-Learning的思想，可以在具有离散动作空间的MDP问题中学习最优策略。它使用经验回放和目标网络来提高训练的稳定性，同时通过ε-greedy策略探索环境并逐渐降低探索率。DQN算法在训练过程中通过更新评估网络的参数，不断优化Q值函数的估计，从而实现最优策略的学习。

2.关键代码及注释

```
import gym
import argparse
import numpy as np
import torch
import torch.nn.functional as F
from torch import nn, optim
import wandb
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) #从输入层到隐藏层
        self.fc2 = nn.Linear(hidden_size, output_size)  #从隐藏层到输出层

    def forward(self, x):
        x = torch.Tensor(x) #转化为tensor张量
        x = F.relu(self.fc1(x)) #激活函数
        x = self.fc2(x) #激活后的数据通过第二个线性层进行线性变换
        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] #根据选中的索引，从缓存器中
获取对应的训练数据，形成一个批次
```

```
        return zip(*batch)  #将列表 batch 中的每个元素按照索引转置，相当于将每个训练数据的相同位置的元素组成一个元组

    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:
            # 随机选择 [0, self.env.action_space.n) 之间的一个动作
            action = np.random.randint(0, self.env.action_space.n)
        else:
            # 根据观察值 "obs" 使用 "eval_net" 获取一个动作
            obs_tensor = torch.FloatTensor(obs)
            with torch.no_grad():
                q_values = self.eval_net(obs_tensor)
            action = q_values.argmax().item()
        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)  # 使用 LongTensor 来进行
gather 操作
        dones = torch.FloatTensor(dones)
        rewards = torch.FloatTensor(rewards)

        q_eval = self.eval_net(np.array(obs)).gather(1,
actions.unsqueeze(1)).squeeze(1)
        # 使用 eval_net 对观察值 obs 进行前向传播，得到相应的 Q 值
        # np.array(obs) 将观察值转换为 NumPy 数组
        # gather(1, actions.unsqueeze(1)) 选择与 actions 对应的 Q 值
        # squeeze(1) 去除不必要的维度

        q_next = self.target_net(np.array(next_obs)).max(dim=1)[0]
        # 使用 target_net 对下一个观察值 next_obs 进行前向传播，得到最大的 Q
值
        # np.array(next_obs) 将下一个观察值转换为 NumPy 数组
        # torch.max(..., dim=1)[0] 选择沿着第 1 维度（动作）的最大 Q 值

        q_target = rewards + args.gamma * (1 - dones) * q_next
        # 根据贝尔曼方程计算目标 Q 值
        # Q_target = rewards + gamma * (1 - dones) * max(Q_next)

        loss = self.loss_fn(q_eval, q_target)
        # 计算 q_eval 和 q_target 之间的损失，使用损失函数 loss_fn

        self.optim.zero_grad()
        # 清除优化器中的梯度信息

        loss.backward()
        # 反向传播计算梯度

        self.optim.step()
        # 更新网络参数


def main():
    env = gym.make(args.env)

    o_dim = env.observation_space.shape[0]
    a_dim = env.action_space.n
    agent = DQN(env, o_dim, args.hidden, a_dim)
```

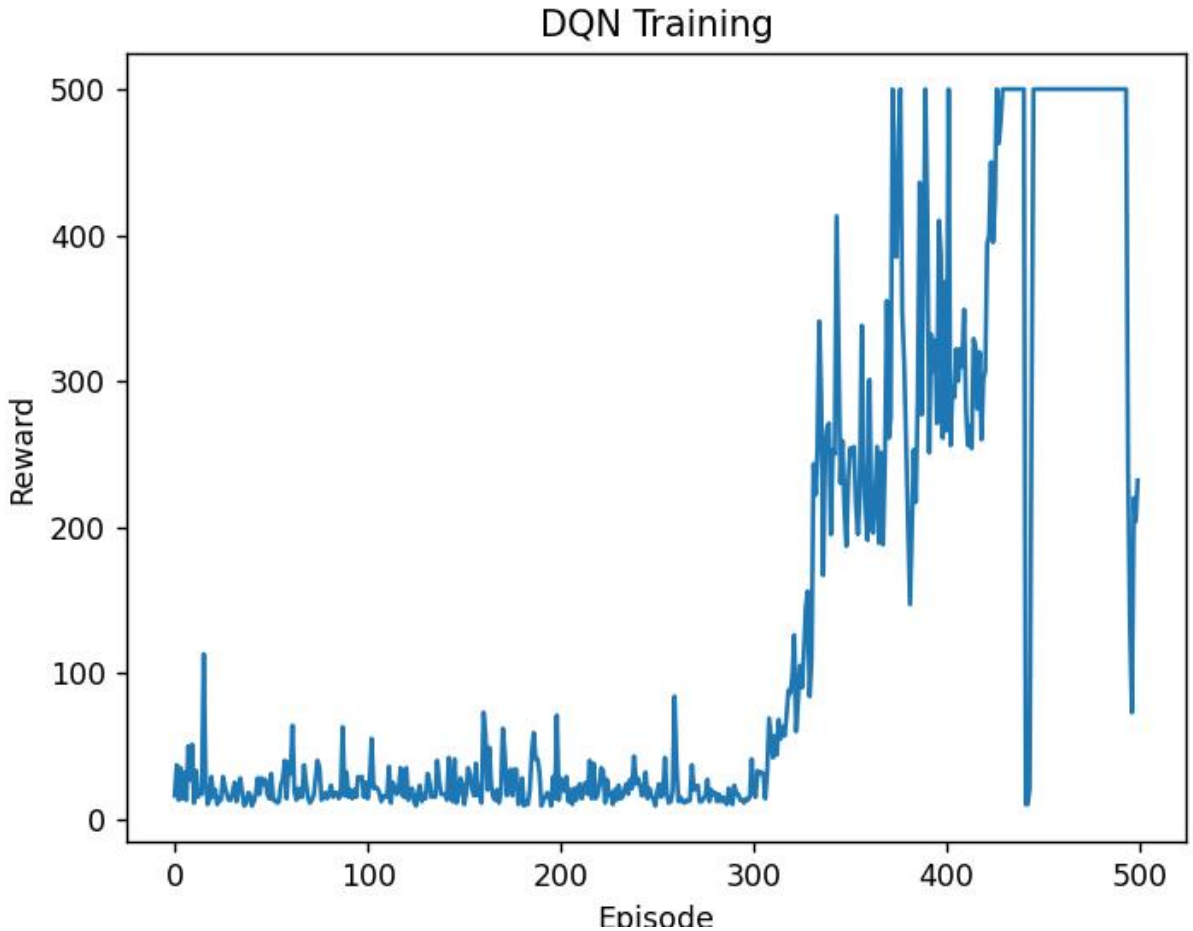
```
    rewards = []  # 记录每一轮次的奖励值

    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 < 500:
            step_cnt += 1
            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()
        rewards.append(episode_reward)
        print(f"Episode: {i_episode}, Reward: {episode_reward}")


    # 数据可视化
    plt.plot(rewards)
    plt.xlabel("Episode")
    plt.ylabel("Reward")
    plt.title("DQN Training")
    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=64, 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()
```

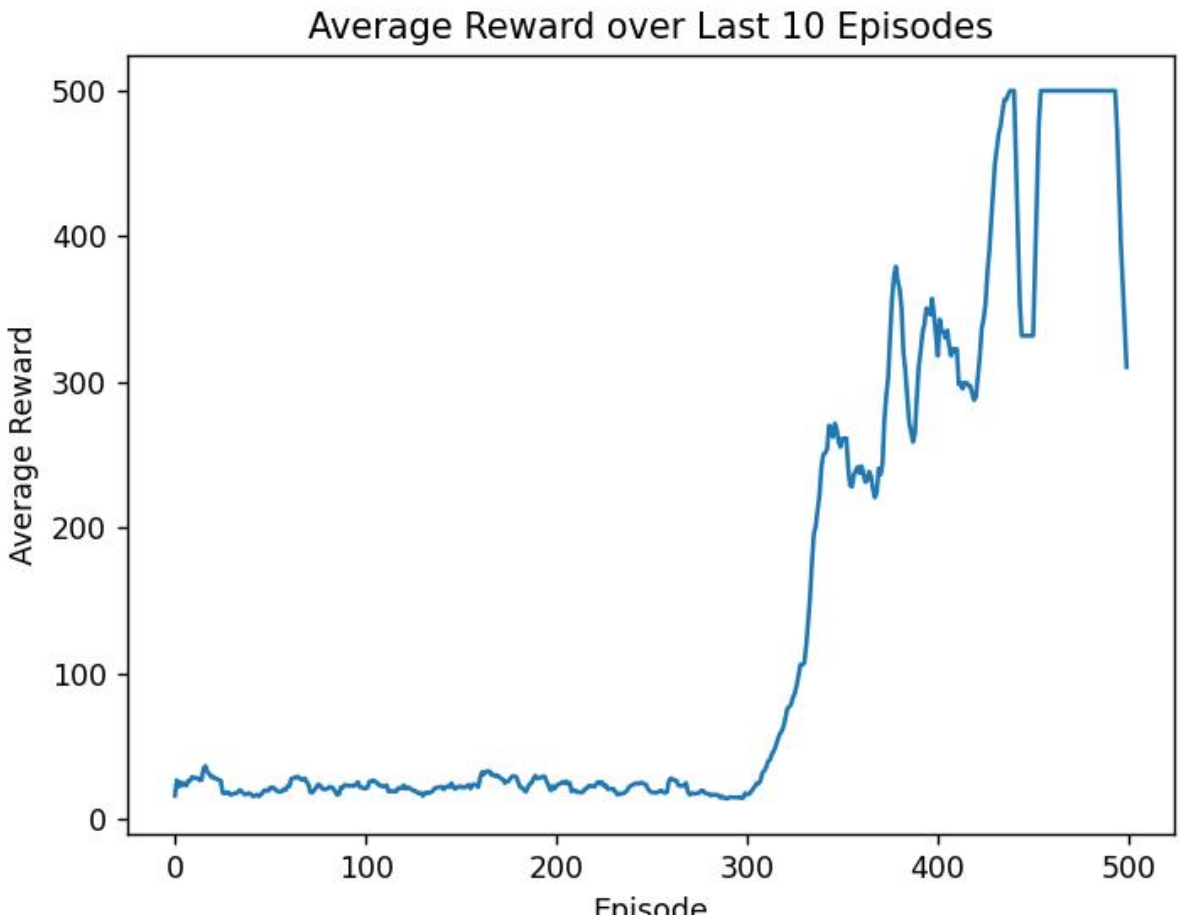
三、实验结果与分析

实验结果展示

- 500轮reward：



- 近10轮reward均值图



实验结果分析

- 通过实验结果可以看出在500轮的训练中，随着训练的进行，reward不断增加，但是在达到500后会出现一定的抖动情况，分析如下：

  1. 随机性：DQN算法中使用epsilon-greedy策略来选择动作，epsilon的初始值为1.0，随着训练的进行逐渐衰减。当epsilon较高时，算法更倾向于进行随机探索，这可能导致在reward较高的情况下仍然选择随机动作，从而导致reward的抖动。

  2. 环境的随机性：某些环境具有随机性，即使采取相同的动作，也可能导致不同的reward结果。这可能是由于环境的内在随机性导致的，例如CartPole-v1游戏中的杆子可能会在不同的初始状态下开始，从而导致不同的轨迹和reward。

  3. 算法的收敛性：DQN算法是一种基于经验回放的近似最优控制算法，它通过不断迭代优化神经网络的参数来逼近最优策略。然而，算法的收敛性并不保证每个episode都能达到最优结果，因此在训练过程中可能会出现reward的抖动。

- 因此近十轮的reward曲线图会出现一定的波动，但是在某一段时期达到了预期的效果