Week13 实验报告

一、实验原理

DQN 算法是 Q-learning 算法的扩展，它用神经网络拟合 Q，克服了表格型强化学习在动作和状态数量、维度很大时的缺陷。

目标函数与损失函数

DQN 要优化的目标函数为

$$Q^*(s,a)=\max_{\pi}\mathbb{E}\left[r_t+\gamma r_{t+1}+\gamma^2 r_{t+2}+\ldots|s_t=s,\,a_t=a,\,\pi\right],$$

即通过改进策略最大化 Q 值。

在神经网络的训练中，DQN 的损失函数则是在 Time Difference 中的 TD-error 的 MSE:

$$L_i(\theta_i)=\mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)}\left[\left(r+\gamma\max_{a'}Q(s',a';\theta_i^-)-Q(s,a;\theta_i)\right)^2\right]$$

动作的选择:ε-Greedy

如果在策略提升中，一直以贪心算法选择动作，则会导致一些状态-动作对(s,a)永远不会出现，我们无法对这些动作对的 Q 值进行估计，因此以ε-Greedy 的策略来选择动作:

$$\pi(a|s)=\begin{cases}\epsilon/|\mathcal{A}|+1-\epsilon & \text{如果}a=\arg\max_{a'}Q(s,a')\\ \epsilon/|\mathcal{A}| & \text{其他动作}\end{cases}$$

经验回放

DQN 算法需要维护一个回放缓冲区，存放采样过的所有样本，并在训练时从缓冲区中随机采样，这样做的好处如下:

1. 使得样本满足独立性假设，在 MDP 中每一个样本都与前一个样本高度相关，非独立的数据对训练的影响非常大，采用经验回放可以打破样本之间的相关性，有利于神经网络的训练。
2. 每一个样本可以被使用多次，适合神经网络的梯度学习。

网络的构建

由于传统的 Q-learning 算法中，

$$Q(s_t,a_t)\leftarrow Q(s_t,a_t)+\alpha[R_t+\gamma\max_{a}Q(s_{t+1},a)-Q(s_t,a_t)]$$

涉及两个 Q 网络：当前 Q 网络和 TD-target 的 Q 网络，因此 DQN 训练时选择以

TD-target 的 Q 网络为目标网络，当前网络为训练网络，用当前 Q 训练来逼近 TD-target,并每隔一段时间用当前 Q 逼近目标 Q。

算法流程

初始化当前网络，并将相同的参数赋值到目标网络
初始化经验回放池
采样 num_episodes 条序列，对于每次采样：
初始化状态 S 为初始状态
对于每个时间步:
用 ε -Greedy 来选择 action
采样 (s_t, a_t, r_t, s_{t+1})，并放入经验回放池
若回放池中样本数目足够，则随机选取 batch_size 个数据，进行网络训练、梯度更新
每隔一段时间用当前网络更新一次目标网络

DQN 的改进：DoubleDQN

在 TD-target 的计算中，对未来的 Q 值估算可以写成这种形式:

$$Q_{\omega^-}\left(s', \arg\max_{a'} Q_{\omega^-}(s', a')\right)$$

可以拆分为两步:
1. 先根据目标网络，选择 Q 值最大对应的动作
2. 再根据目标网络计算(s',a')对应的 Q 值

当对某些动作的估算有正误差时，这个误差就会在目标网络中不断累积，导致目标 Q 值被过高估计。

因此，DoubleDQN 做出的改进为，用当前网络选取动作，目标网络计算 Q 值:

$$r + \gamma Q_{\omega^-}\left(s', \arg\max_{a'} Q_{\omega}(s', a')\right)$$

这样即使其中一套神经网络的某个动作存在较严重的过高估计问题，由于另一套神经网络的存在，这个动作最终使用的 Q 值不会存在很大的过高估计问题。

二、关键代码展示

Q 网络：输入为状态维度，输出动作维度

```python
class QNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, inputs):
        inputs=F.relu(self.fc1(inputs))
        return self.fc2(inputs)
```

经验回放池：用 deque 来维护

```python
class ReplayBuffer:
    def __init__(self, buffer_size):
        self.buffer=collections.deque(maxlen=buffer_size)

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

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))
```

```python
def sample(self, batch_size):
    mysample=random.sample(self.buffer, batch_size)
    '''
    mysample:[(s,a,r,s',done)],然后对它按列分组存入
    '''
    states,actions,rewards,next_states,dones=zip(*mysample)#对mysample里面的按列分组
    states=torch.tensor(states,dtype=torch.float)   #二维张量[batch_size,state_dim]
    actions=torch.tensor(actions,dtype=torch.int64).view(-1,1) #标量，要转化成[batch_size,1]
    rewards=torch.tensor(rewards,dtype=torch.float).view(-1,1)
    next_states=torch.tensor(next_states,dtype=torch.float)
    dones=torch.tensor(dones,dtype=torch.float).view(-1,1)

    return states,actions,rewards,next_states,dones
def clean(self):
    self.buffer.clear()
def size(self):
    return len(self.buffer)
```

选择动作：

```python
def make_action(self, observation, test):
    """
    Return predicted action of your agent
    Input:observation
    Return:action
    """
    action=np.random.randint(self.action_dim)
    if test:
        state=torch.tensor([observation],dtype=torch.float).to(self.device)#[batch_size,state_dim]
        action=self.train_net(state).max(1)[1].item()
    else:
        if np.random.random()<self.epsilon:
            return action
        else:
            state=torch.tensor([observation],dtype=torch.float).to(self.device)
            action=self.train_net(state).max(1)[1].item()
    return action
```

得到的(s_t, a_t, r_t, s_{t+1})进行更新，且根据 DQN 的类型区分对 MAX_next_Q 的计算

```python
def run(self,states,actions,rewards,next_states,dones):
    """
    Implement the interaction between agent and environment here
    """
    states=states.to(self.device)
    actions=actions.to(self.device)
    rewards=rewards.to(self.device)
    next_states=next_states.to(self.device)
    dones=dones.to(self.device)

    q_values=self.train_net(states).gather(1,actions)
    max_next_q=0
    if self.dqn_type=='DoubleDQN':
        best_actions=self.train_net(next_states).argmax(dim=1).view(-1,1)
        max_next_q=self.target_net(next_states).gather(1,best_actions)
    else:
        max_next_q=self.target_net(next_states).max(1)[0].view(-1,1)
    TD_targets=rewards+self.gamma*max_next_q*(1-dones)

    loss=F.mse_loss(q_values,TD_targets)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    self.count+=1
    if self.count%self.target_update==0:
        self.target_net.load_state_dict(self.train_net.state_dict())
```

DQN 的训练，并记录每次 return:

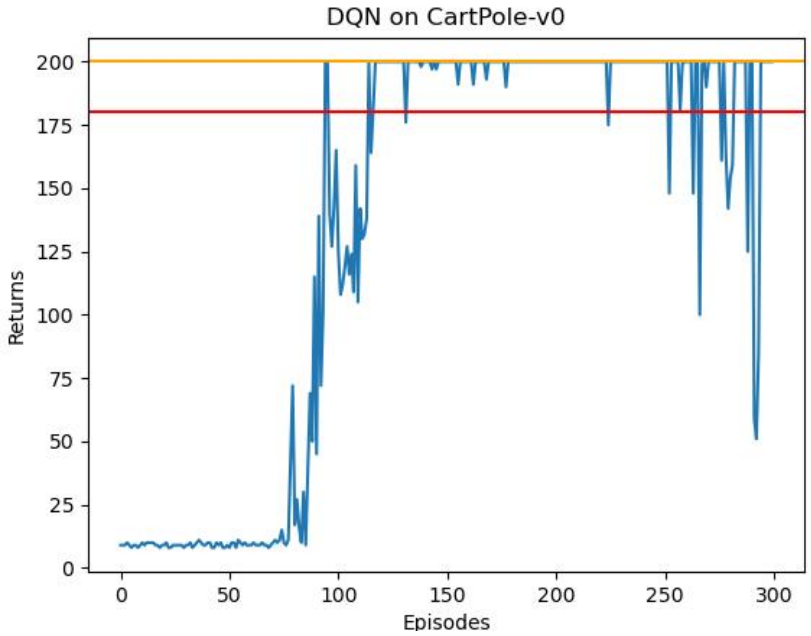
```python
def train(self):
    """
    Implement your training algorithm here
    """
    return_list=[]
    for i in range(self.num_episodes):
        state=self.init_game_setting()
        episode_return=0
        done=False
        while not done:
            action=self.make_action(state,self.test)
            next_state,reward,terminated,trancated,_=self.env.step(action)
            done=terminated or trancated
            self.buffer.push(state,action,reward,next_state,done)
            state=next_state
            episode_return+=reward
            if self.buffer.size()>=self.min_size:
                states,actions,rewards,next_states,dones=self.buffer.sample(self.batch_size)
                self.run(states,actions,rewards,next_states,dones)
        return_list.append(episode_return)
        self.scheduler.step(episode_return)
        print(f'Episode {i}: Return {episode_return},lr {self.scheduler.get_last_lr()}')
    return return_list
```

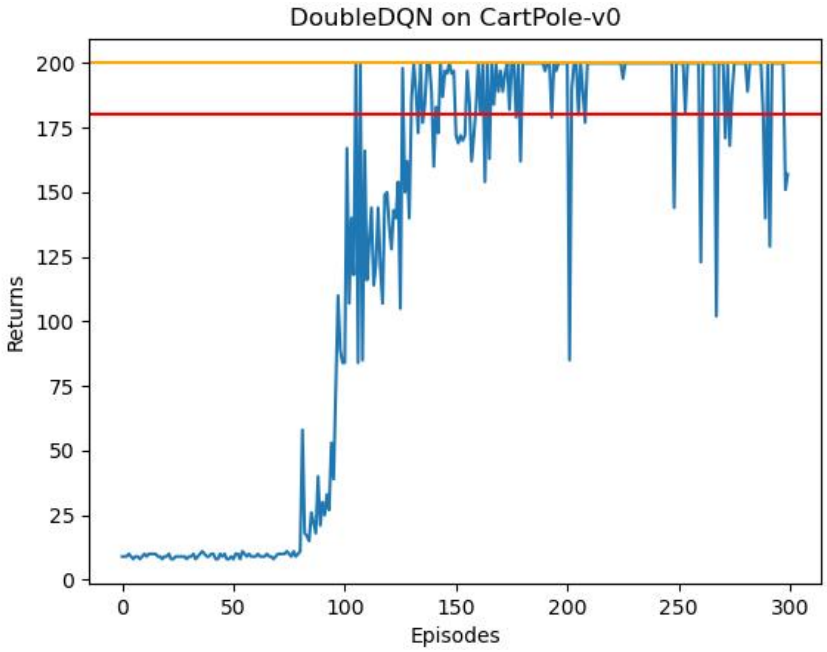
三、实验结果展示

本次实验在 OPENAI 的 Carpole-V0 环境下进行，其单个 Trajectory 的 Return 满分为 200 分，各个参数的配置如下：

```
python main.py `
--env_name "CartPole-v0" `
--seed 11037 `
--hidden_size 128 `
--lr 5e-3 `
--gamma 0.98 `
--epsilon 0.01 `
--target_update 10 `
--batch_size 64 `
--buffer_size 10000 `
--dqn_type "DQN" `
--num_episodes 300 `
--min_size 500 `
--test False `
--use_cuda True `
```

分别以 DQN 和 DoubleDQN 两种方法进行了实验，它们的 Epoch-Return 图像如下:





可以观察到这两种 DQN 在第 100 轮的时候能达到了满分的成绩，DQN 在训练的后期会存在较大的波动(return 小于 100)，而 DoubleDQN 虽存在波动，但波动幅度较小。

四、创新点&优化

1. 对比了 DQN 与 DoubleDQN 算法在的实验结果，分析了两种 DQN 算法的优缺点
2. 不断调整 DQN 的训练参数，使得它们在 150 轮左右能多次收敛到 200 分的满分结果

五、参考资料

1. https://hrl.boyuai.com/ 动手学强化学习
2. Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).