7 - DQN算法

深度网络

为了解决 $Q-learning$ 中无法很好处理的高维问题，DQN 算法被推出来。在 $Q-learning$ 中我们是直接优化 Q 值的，但是再 DQN 我们只用神经网络近似 Q 函数。

$$y_i \begin{cases} r_i & \text{终止状态} \\ r_i + \gamma \max_{a'} Q(s_{i+1}, a'; \theta) & \text{非终止状态} \end{cases}$$

$$L(\theta) = (y_i - Q(s_i, a_i; \theta))^2$$

$$\theta_i \leftarrow \theta_i - \alpha \nabla_{\theta_i} L_i(\theta_i)$$

经验回放

问题：

1. 首先每次用单个样本去迭代网络参数很容易导致训练的不稳定，从而影响模型的收敛，在深度学习基础的章节中我们也讲过小批量梯度下降是目前比较成熟的方式。
2. 其次，每次迭代的样本都是从环境中实时交互得到的，这样的样本是有关联的，而梯度下降法是基于一个假设的，即训练集中的样本是独立同分布的。

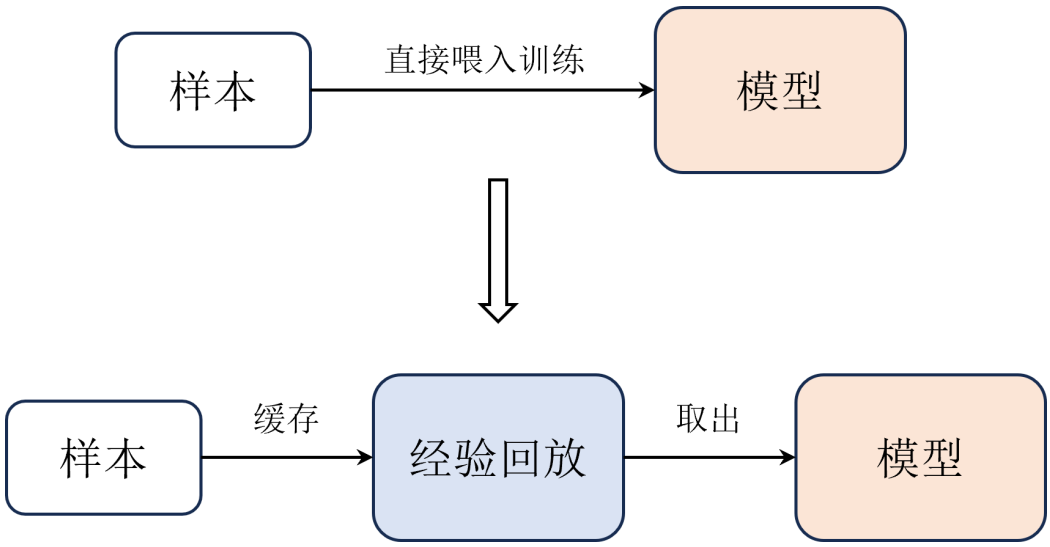

图 7-2 经验回放示例

如图 7-2 所示，Q-learning 算法训练的方式就是把每次通过与环境交互一次的样本直接喂入网络中训练。而在 DQN 中，我们会把每次与环境交互得到的样本都存储在一个经验回放中，然后每次从经验池中随机抽取一批样本来训练网络。

目标网络

在 DQN 算法中还有一个重要的技巧，即使用了一个每隔若干步才更新的目标网络。如图 7-3 所示，目标网络和当前网络结构都是相同的，都用于近似 Q 值，在实践中每隔若干步才把每步更新的当前网络参数复制给目标网络，这样做的好处是保证训练的稳定，避免 Q 值的估计发散。

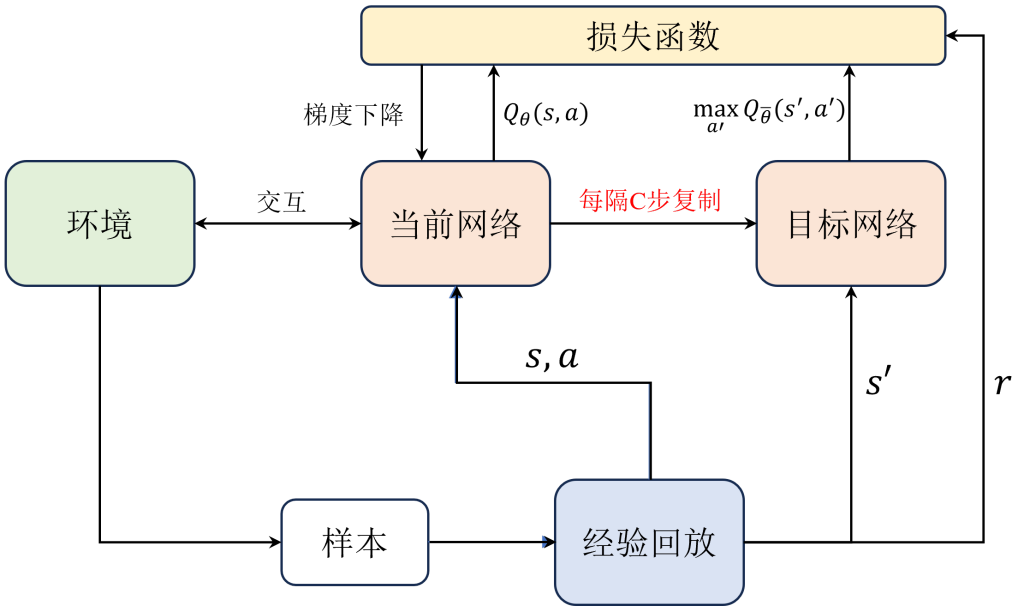

图 7-3 目标网络示例

同时在计算损失函数的时候，使用的是目标网络来计算 Q 的期望值，如式 (7.3) 所示。

$$Q_{\mathbb{E}} = [r_t + \gamma \max_{a'} Q_{\bar{\theta}}(s', a')]$$

实战：DQN算法

伪代码

DQN 算法

初始化当前网络参数 θ
复制参数到目标网络 $\hat{\theta} \leftarrow \theta$
初始化经验回放 D
for 回合数 $m = 1, 2, \cdots, M$ **do**
  重置环境，获得初始状态 s_0
  for 时步 $t = 1, 2, \cdots, T$ **do**
    交互采样：
    根据 $\varepsilon - greedy$ 策略采样动作 a_t
    环境根据 a_t 反馈奖励 r_t 和下一个状态 s_{t+1}
    存储样本 (s_t, a_t, r_t, s_{t+1}) 到经验回放 D 中
    更新环境状态 $s_{t+1} \leftarrow s_t$
    模型更新：
    从 D 中随机采样一个批量的样本
    计算 Q 的期望值，即 $y_i = r_t + \gamma \max_{a_{i+1}} Q(s_{i+1}, a; \hat{\theta})$
    计算损失 $L(\theta) = (y_i - Q(s_i, a_i; \theta))^2$，并关于参数 θ 做随机梯度下降
    每 C 步复制参数到目标网络 $\hat{\theta} \leftarrow \theta$
  end for
end for

图 7-4 DQN 算法伪代码

与 Q-learning 算法不同的是，这里由于用的是神经网络，因此会多一个计算损失函数并进行反向传播的步骤

定义模型

```
class MLP(nn.Module):
    def __init__(self, input_dim,output_dim,hidden_dim=128):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim,hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

经验回放

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity # 经验回放的容量
        self.buffer = [] # 用列表存放样本
        self.position = 0 # 样本下标，便于覆盖旧样本

    def push(self, state, action, reward, next_state, done):
        '''
        缓存样本
        '''
        if len(self.buffer) < self.capacity: # 如果样本数小于容量
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        '''
        取出样本，即采样
        '''
        batch = random.sample(self.buffer, batch_size) # 随机采出小批量转移
        state, action, reward, next_state, done =  zip(*batch) # 解压成状态，动作等
        return state, action, reward, next_state, done

    def __len__(self):
        '''
        返回当前样本数
        '''
        return len(self.buffer)
```

定义智能体

```
class Agent:
    def __init__(self):
        # 定义当前网络
        self.policy_net = MLP(state_dim,action_dim).to(device)
        # 定义目标网络
```

```
        self.target_net = MLP(state_dim,action_dim).to(device)
        # 将当前网络参数复制到目标网络中
        self.target_net.load_state_dict(self.policy_net.state_dict())
        # 定义优化器
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=learning_rate)
        # 经验回放
        self.memory = ReplayBuffer(buffer_size)
        self.sample_count = 0  # 记录采样步数

    def sample_action(self,state):
        '''
        采样动作，主要用于训练
        '''
        self.sample_count += 1
        # epsilon 随着采样步数衰减
        self.epsilon = self.epsilon_end + (self.epsilon_start - self.epsilon_end) *
math.exp(-1. * self.sample_count / self.epsilon_decay)
        if random.random() > self.epsilon:
            with torch.no_grad(): # 不使用梯度
                state = torch.tensor(np.array(state), device=self.device,
dtype=torch.float32).unsqueeze(dim=0)
                q_values = self.policy_net(state)
                action = q_values.max(1)[1].item() # choose action corresponding to the
maximum q value
        else:
            action = random.randrange(self.action_dim)

    def predict_action(self,state):
        '''
        预测动作，主要用于测试
        '''
        with torch.no_grad():
            state = torch.tensor(np.array(state), device=self.device,
dtype=torch.float32).unsqueeze(dim=0)
            q_values = self.policy_net(state)
            action = q_values.max(1)[1].item() # choose action corresponding to the maximum
q value
        return action

    def update(self, share_agent=None):
            # 当经验回放中样本数小于更新的批大小时，不更新算法
            if len(self.memory) < self.batch_size:
            # when transitions in memory do not meet a batch, not update
            return
            # 从经验回放中采样
            state_batch, action_batch, reward_batch, next_state_batch, done_batch =
self.memory.sample( self.batch_size)
            # 转换成张量（便于GPU计算）
            state_batch = torch.tensor(np.array(state_batch), device=self.device,
dtype=torch.float)
            action_batch = torch.tensor(action_batch, device=self.device).unsqueeze(1)
            reward_batch = torch.tensor(reward_batch, device=self.device,
dtype=torch.float).unsqueeze(1)
            next_state_batch = torch.tensor(np.array(next_state_batch), device=self.device,
dtype=torch.float)
```

```
            done_batch = torch.tensor(np.float32(done_batch),
device=self.device).unsqueeze(1)
            # 计算 Q 的实际值
            q_value_batch = self.policy_net(state_batch).gather(dim=1, index=action_batch) #
shape(batchsize,1),requires_grad=True
            # 计算 Q 的估计值，即 r+\gamma Q_max
            next_max_q_value_batch = self.target_net(next_state_batch).max(1)
[0].detach().unsqueeze(1)
            expected_q_value_batch = reward_batch + self.gamma * next_max_q_value_batch* (1-
done_batch)
            # 计算损失
            loss = nn.MSELoss()(q_value_batch, expected_q_value_batch)
            # 梯度清零，避免在下一次反向传播时重复累加梯度而出现错误。
            self.optimizer.zero_grad()
            # 反向传播
            loss.backward()
            # clip避免梯度爆炸
            for param in self.policy_net.parameters():
                    param.grad.data.clamp_(-1, 1)
                # 更新优化器
                self.optimizer.step()
                # 每C(target_update)步更新目标网络
                if self.sample_count % self.target_update == 0:
                        self.target_net.load_state_dict(self.policy_net.state_dict())
```

定义环境

这里我们使用 OpenAI Gym 平台的 Cart Pole 环境作为本次算法的训练环境。

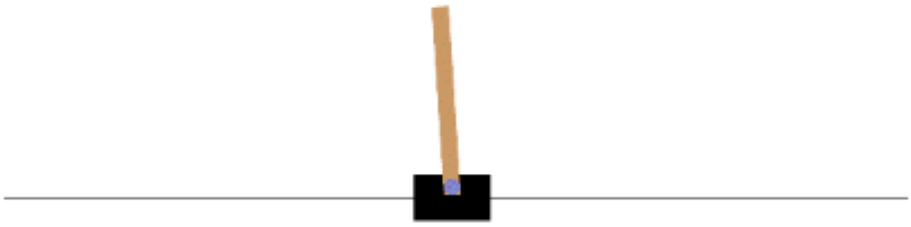

图 7-5 Cart-Pole 游戏

设置参数

```
self.epsilon_start = 0.95  # epsilon 起始值
self.epsilon_end = 0.01  # epsilon 终止值
self.epsilon_decay = 500  # epsilon 衰减率
self.gamma = 0.95  # 折扣因子
self.lr = 0.0001  # 学习率
self.buffer_size = 100000  # 经验回放容量
self.batch_size = 64  # 批大小
self.target_update = 4  # 目标网络更新频率
```

成果：

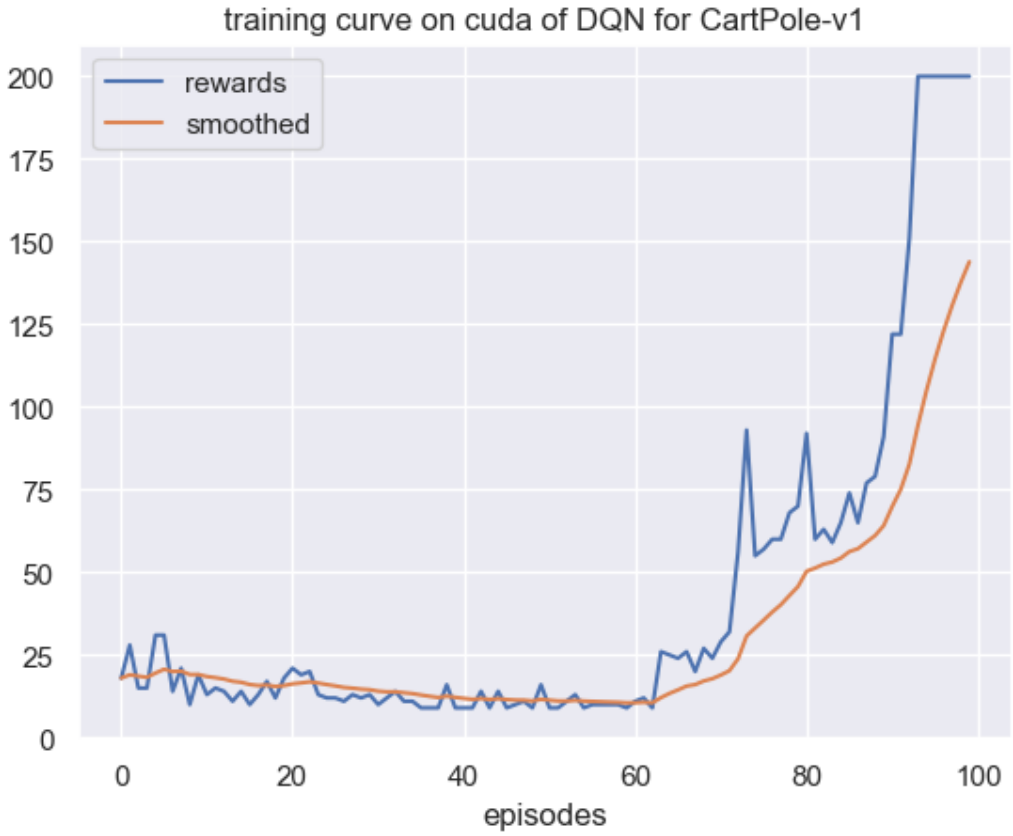

图 7-6 CartPole-v1 环境 DQN 算法训练曲线

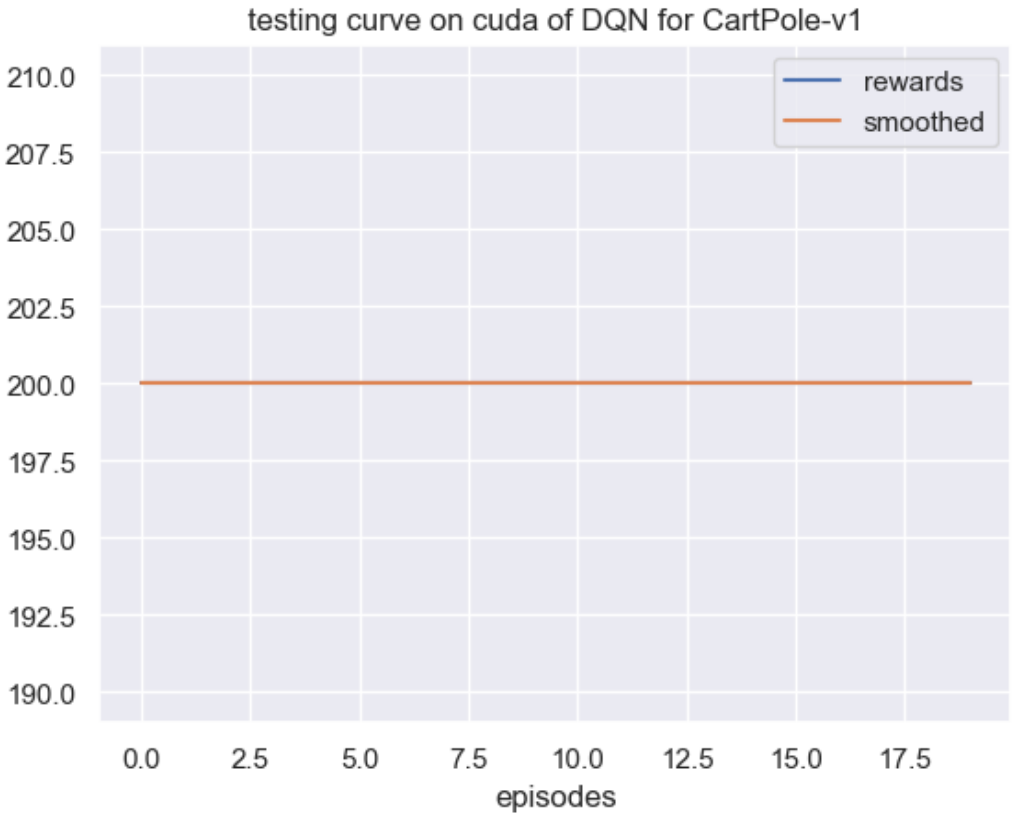

图 7-7 CartPole-v1 环境 DQN 算法测试曲线

练习题

相比于Q-learning 算法，DQN 算法做了哪些改进？

- 增加了经验回放
- 引入 ϵ - greedy 策略
- 使用深度学习

为什么要在 DQN 算法中引入 ϵ-greedy 策略?

寻找最优的探索策略

DQN 算法为什么要多加一个目标网络?

- 提高稳定性
- 减少相关性
- 防止目标值的过度估计

经验回放的作用是什么?

- 提高稳定性
- 减少相关性