5 - 免模型控制

Q-learning算法

状态价值函数的时序差分的目的是为了预测每个状态的价值。

动作价值函数

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) Q_\pi(s,a)$$

为了控制问题，我们只需要直接预测动作价值函数，然后再决策时选择动作价值Q之最大对用的动作即可。这样一来，策略和动作价值函数同时达到最优，相应的状态价值函数也是最优的，这就是 Q-learning 算法的思路。

Q-learning算法的更新公式

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

这个公式和时序差分中的状态价值函数的更新公式一样，只是在动作价值函数更新时直取极值。

Q表格

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
a_1	0	0	0	0	0	0	0	0	0
a_2	0	0	0	0	0	0	0	0	0
a_3	0	0	0	0	0	0	0	0	0
a_4	0	0	0	0	0	0	0	0	0

表 5.1: Q 表格

在Q-learning中，我们将所有的状态更新都记录于上述的Q表格之中。具体的做法是，我们会让机器人自行在网格中走动，走到一个状态，就把对应的 Q 值 更新一次，这个过程就叫做探索.

探索策略

回到正题，Q-learning 算法是采用了一个叫做 $\epsilon-greedy$ 的探索策略，$\epsilon-greedy$ 是指智能体在探索的过程中，会以 1- ϵ 的概率按照 Q 函数来执行动作，然后以剩下 ϵ 的概率随机动作。
当然，通常这个 ϵ 的值会设置的特别小，比如 0.1 ，毕竟“守旧”并不总是一件坏事，新的东西出现的概率总是特别小的，如果保持过度的好奇心即 ϵ 的值设得很大，就很有可能导致智能体既学不到新的东西也丢掉了已经学习到的东西，所谓“捡了芝麻丢了西瓜”。
而且一般来说也会随着学到的东西增多而更少，就好比科学知识体系几近完备的现代，能够探索到新的东西的概率是特别特别小的。因此通常在实践中，这个 ϵ 的值还会随着时步的增长而衰减，比如从 0.1 衰减到 0.01 。

Q-learning 算法[①]

初始化 Q 表 $Q(s,a)$ 为任意值，但其中 $Q(s_{terminal},)=0$，即终止状态对应的 Q 值为 0
for 回合数 $=1,M$ **do**
  重置环境，获得初始状态 s_1
  for 时步 $=1,T$ **do**
    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}
    更新策略：
    $Q(s_t,a_t)\leftarrow Q(s_t,a_t)+\alpha[r_t+\gamma\max_a Q(s_{t+1},a)-Q(s_t,a_t)]$
    更新状态 $s_{t+1}\leftarrow s_t$
  end for
end for

图 5.3 Q-learning 算法伪代码

Sarsa 算法

Sarsa 算法虽然在刚提出的时候被认为是 Q-learning 算法的改进，但在今天看来是非常类似。
Sarsa 算法中 Q 值得更新公式：

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

也就是说，Sarsa 算法是直接用下一个状态和动作对应的 Q 值来作为估计值的，而 Q-learning 算法则是用下一个状态对应的最大 Q 值。

Sarsa 算法[①]

初始化 Q 表 $Q(s,a)$ 为任意值，但其中 $Q(s_{terminal},)=0$，即终止状态对应的 Q 值为 0
for 回合数 $=1,M$ **do**
  重置环境，获得初始状态 s_1
  根据 $\varepsilon-greedy$ 策略采样初始动作 a_1
  for 时步 $=1,t$ **do**
    环境根据 a_t 反馈奖励 r_t 和下一个状态 s_{t+1}
    根据 $\varepsilon-greedy$ 策略 s_{t+1} 和采样动作 a_{t+1}
    更新策略：
    $Q(s_t,a_t)\leftarrow Q(s_t,a_t)+\alpha[r_t+\gamma Q(s_{t+1},a_{t+1})-Q(s_t,a_t)]$
    更新状态 $s_{t+1}\leftarrow s_t$
    更新动作 $a_{t+1}\leftarrow a_t$
  end for
end for

图 5.4 Sarsa 算法伪代码

同策略于异策略

尽管 Q-learning 算法和 Sarsa 算法仅在一行更新公式上有所区别，但这两种算法代表的是截然不同的两类算法。

我们注意到，Sarsa 算法在训练的过程中当前策略来生成数据样本，并在其基础上进行更新。换句话说，策略评估和策略改进过程是基于相同的策略完成的，这就是同策略算法。

相应地，像 Q-learning 算法这样从其他策略中获取样本然后利用它们来更新目标策略，我们称作异策略算法。也就是说，异策略算法基本上是从经验池或者历史数据中进行学习的。

实战：Q-learning算法

定义训练

```
for i_ep in range(train_eps): # 遍历每个回合
    # 重置环境，获取初始状态
    state = env.reset()  # 重置环境,即开始新的回合
    while True: # 对于比较复杂的游戏可以设置每回合最大的步长，例如while ep_step<100，即最大步长为
100。
        # 智能体根据策略采样动作
        action = agent.sample_action(state)  # 根据算法采样一个动作
        # 与环境进行一次交互，得到下一个状态和奖励
        next_state, reward, terminated, _ = env.step(action)
        # 智能体将样本记录到经验池中
        agent.memory.push(state, action, reward, next_state, terminated)
        # 智能体更新策略
        agent.update(state, action, reward, next_state, terminated)
        # 更新状态
        state = next_state
        # 如果终止则本回合结束
        if terminated:
            break
```

定义算法

这里定义强化学习中的几个要素，智能体、环境、经验池（经回放）。

采样动作

```
class Agent:
    def __init__():
        pass

    def sample_action(self, state):
        '''
        采样动作，训练时用
        '''
        self.sample_count += 1
        # epsilon是会递减的，这里选择指数递减
        self.epsilon = self.epsilon_end + (self.epsilon_start - self.epsilon_end) *
math.exp(- self.sample_count / self.epsilon_decay)
```

```
        # e-greedy 策略
        if np.random.uniform(0, 1) > self.epsilon:
            # 选择Q(s,a)最大对应的动作
            action = np.argmax(self.Q_table[str(state)])
        else:
                # 随机选择动作
            action = np.random.choice(self.n_actions)
        return action
```

在这里我们用了 $\epsilon - greedy$ 策略，其中 ϵ 会随着采样的步数指数衰减，感兴趣的读者也可以直接设置固定的 ϵ = 0.1 试试。
在 Q-learning 算法中还有一个重要的元素，即 Q 表，Q 表的作用是输入状态和动作，输出一个即可，这样一来我们可以用一个二维的数组来表示，比如 `Q_table[0][1] = 0.1` 可以表示 $Q(s_0, a_1)$=0.1 （注意 Python 中下标是从 0 开始）。

```
self.Q_table  = defaultdict(lambda: np.zeros(n_actions))
```

这样的好处是从数据结构上来说，默认字典是哈希表结构，二维数组是线性表结构，从哈希表拿出数据的速度会比线性表快。

预测动作

此外对于每个智能体在训练中和在测试中采取动作的方式一般是不一样的，因为在训练中需要增加额外的探索策略，而在测试中只需要输出 Q 值对应最大的动作即可，如下

```
class Agent:
        def __init__():
                pass
        def predict_action(self,state):
                '''
                预测或选择动作，测试时用
                '''
                action = np.argmax(self.Q_table[str(state)])
                return action
```

更新方法

Q-learning 的更新方法：

```
def update(self, state, action, reward, next_state, terminated):
    Q_predict = self.Q_table[str(state)][action]
    if terminated: # 终止状态
        Q_target = reward
    else:
        Q_target = reward + self.gamma * np.max(self.Q_table[str(next_state)])
    self.Q_table[str(state)][action] += self.lr * (Q_target - Q_predict)
```

其中`self.lr`是更新公式中的`alpha`（学习率）

定义环境

在本节中我们选择了一个叫做 CliffWalking-v0 的环境（中文名叫 “悬崖寻路” ），跟前面动态规划章节中举的机器人最短路径是类似的，只是要更加复杂一些。

如图 5.5 所示，整个环境中共有 48 个网格，其中黄色网格（标号为 36 ）为起点，绿色网格（标号为 47 ）为终点，红色的网格表示悬崖，智能体的目标是以最短的路径从起点到终点，并且避开悬崖。由于这个环境比较简单，我们一眼就能看出来最优的策略应当是从起点向上沿着 24 号网格直线走到 35 号网格最后到达终点，后面我们看看强化学习智能体能不能学出来。

图 5.5 CliffWalking-v0 环境示意图

此外，我们做强化学习算法的时候更重要的还是要对环境本身的状态、动作和奖励了解，以便指导我们优化算法。在这个环境中，状态比较简单，就是当前智能体所处的网格位置或者说编号，动作就是上右下左（分别是 0,1,2,3 ，顺序可能有点奇怪，但官方环境是这么设置的），奖励的话分几种情况，一个是每走一个白色网格（包括起点）是会给一个 -1 的奖励，到达终点的时候得到的奖励为 0，走到边沿、悬崖或者终点的时候本回合游戏结束，这些设置在官方源码中都能找到①。这里之所以每走一个网格会给一个负的奖励，是因为我们的目标是最短路径，换句话说每走一步都是有代价的，所以需要设置一个负的奖励或者说惩罚，设置正的奖励会容易误导智能体在训练过程中一直走网格，设置 0 的话也是一样的，会让智能体找不到目标。奖励就相当于我们给智能体设置的目标，因此如何合理地设置奖励其实也是一项复杂的工程，具体后面我们会再展开。

这里使用`OpenAI Gym`环境开发：

```
env = gym.make('CliffWalking-v0')
```

在 Gym 中我们可以通过以下方式获取环境的状态数和动作数：

```
n_states = env.observation_space.n # 状态数
n_actions = env.action_space.n # 动作数
print(f"状态数：{n_states}， 动作数：{n_actions}")
```

结果：

```
状态数：48， 动作数：4
```

设置参数

智能体、环境和训练的代码都写好之后，就是设置参数了，由于 Q-learning 算法的超参数（需要人工调整的参数）比较少。
其中 γ（折扣因子）比较固定，设置在 0.9 到 0.999 之间，一般设置成 0.99 即可。
而学习率 $alpha$ 在本章节中设置的比较大，为 0.1，实际更复杂的环境和算法中学习率是小于 0.01，因为太大很容易发生过拟和的问题，只是本节的环境和算法都比较简单，为了收敛得更快点所以设置得比较大。
此外由于我们探索策略中的 ϵ 是会随着采样步数衰减的，在实践过程中既不能让它衰减得太快也不能让它衰减得太慢，因此需要合理设置如下参数：

```
self.epsilon_start = 0.95 #  e-greedy策略中epsilon的初始值
self.epsilon_end = 0.01 #  e-greedy策略中epsilon的最终值
self.epsilon_decay = 200 #  e-greedy策略中epsilon的衰减率
```

开始训练

准备工作做好之后，就可以开始训练了，得到的训练曲线如图 5.6 所示，曲线横坐标表示回合数（episode），纵坐标表示每回合获得的总奖励，可以看出曲线其实从大约 50 个回合的时候就开始收敛了，也就是我们的智能体学到了一个最优策略。

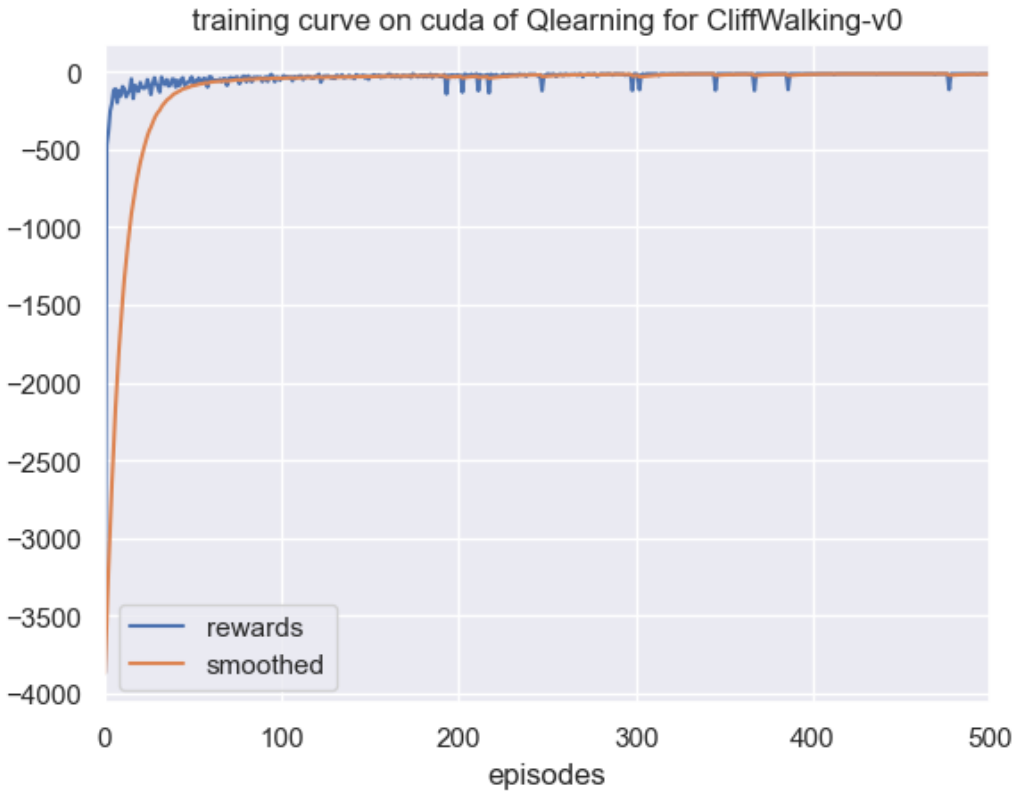

图 5.6 CliffWalking-v0 环境 Q-learning 算法训练曲线

500回合的训练结果：

```
回合：300/500，奖励：-13.00
回合：310/500，奖励：-13.00
回合：320/500，奖励：-14.00
回合：330/500，奖励：-14.00
回合：340/500，奖励：-13.00
回合：350/500，奖励：-13.00
回合：360/500，奖励：-13.00
回合：370/500，奖励：-13.00
```

```
回合：380/500，奖励：-13.00
回合：390/500，奖励：-13.00
回合：400/500，奖励：-13.00
回合：410/500，奖励：-13.00
回合：420/500，奖励：-13.00
回合：430/500，奖励：-13.00
回合：440/500，奖励：-13.00
回合：450/500，奖励：-13.00
回合：460/500，奖励：-13.00
回合：470/500，奖励：-17.00
回合：480/500，奖励：-13.00
回合：490/500，奖励：-13.00
回合：500/500，奖励：-13.00
完成训练！
```

我们发现收敛值约在 -13 左右波动，波动的原因是因为此时还存在 0.01 的概率做随机探索。

如图 5.7 所示，我们测试了 10 个回合，发现每回合获得的奖励都是 -10 左右，说明我们学到的策略是比较稳定的。

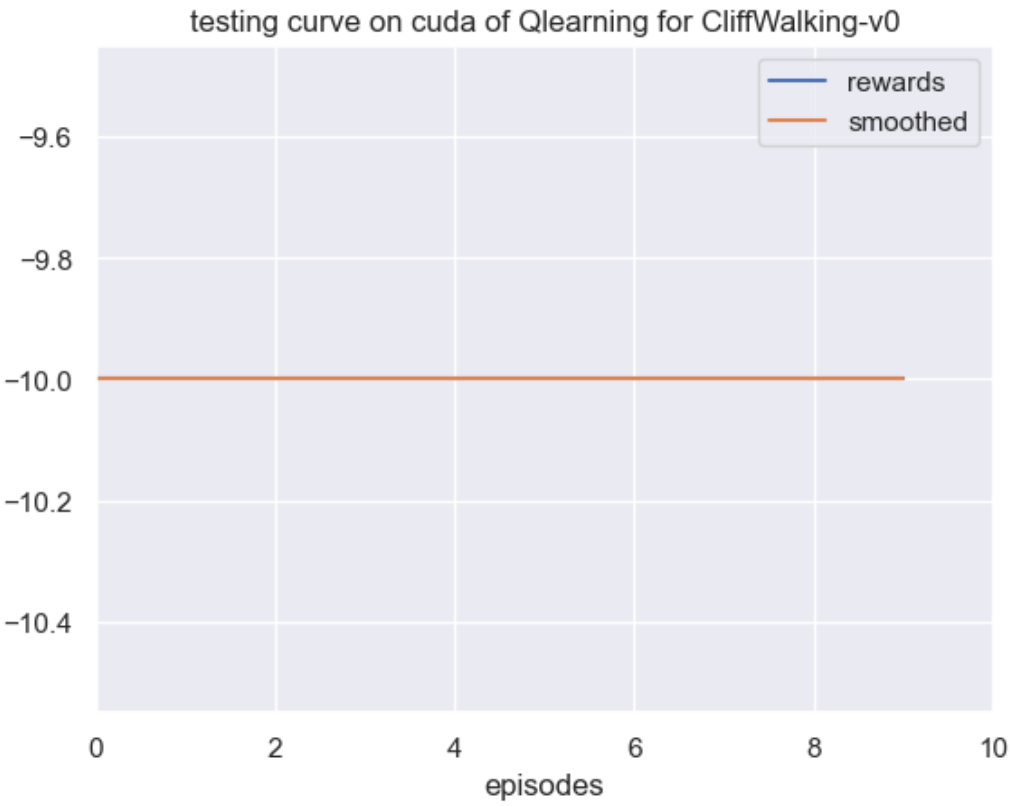

图 5.7 CliffWalking-v0 环境 Q-learning 算法测试曲线

结果分析

那么问题来了,为什么学到的策略每回合的奖励是 -10 呢？回顾一下我们在前面介绍环境的时候讲到，我们一眼就能看出来最优的策略应当是从起点向上沿着 24 号网格直线走到 35 号网格最后到达终点，而这中间要走多少个网格呢？读者们可以数一下，不包括终点（走到终点得到的奖励是 0）的话正好就是 13 步，每一步会得到 -1 的奖励，总共加起来正好也是 -10 ，这说明智能体学到的策略很有可能就是最优的。具体我们还需要把智能体在测试的时候每回合每步的动作打印出来验证一下，打印结果如下：

```
测试的动作列表：[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

可以看到智能体学到的策略是先往上（即动作 0），然后一直往右（即动作 1）走十二格，最后再往下（即动作 2），这其实就是我们肉眼就能看出来的最优策略！

消融实验

为了进一步探究 ϵ 是随着采样步数衰减更好些，还是恒定不变更好，我们做了一个消融（ Ablation ）实验，即将 ϵ 设置为恒定的 0.1，如下：

```
# 将初始值和最终值设置为一样，这样 epsilon 就不会衰减
self.epsilon_start = 0.1 #  e-greedy策略中epsilon的初始值
self.epsilon_end = 0.1 #  e-greedy策略中epsilon的最终值
self.epsilon_decay = 200 #  e-greedy策略中epsilon的衰减率
```

重新训练和测试的结果：

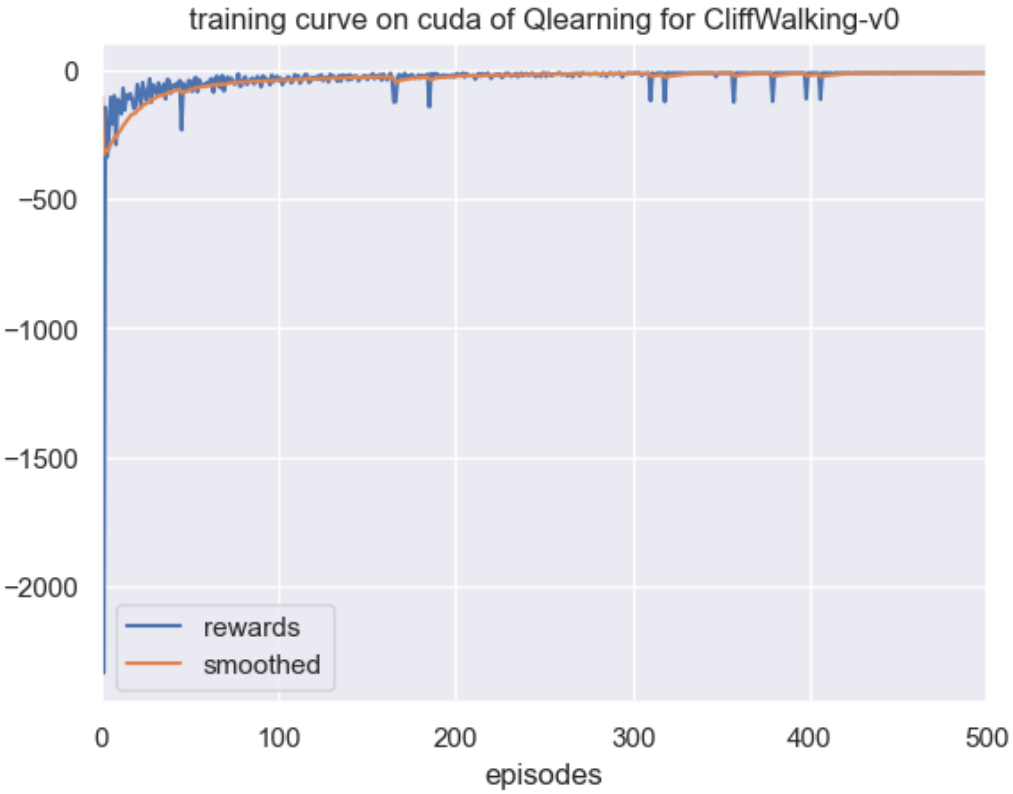

图 5.8 Q-learning 算法消融实验训练曲线

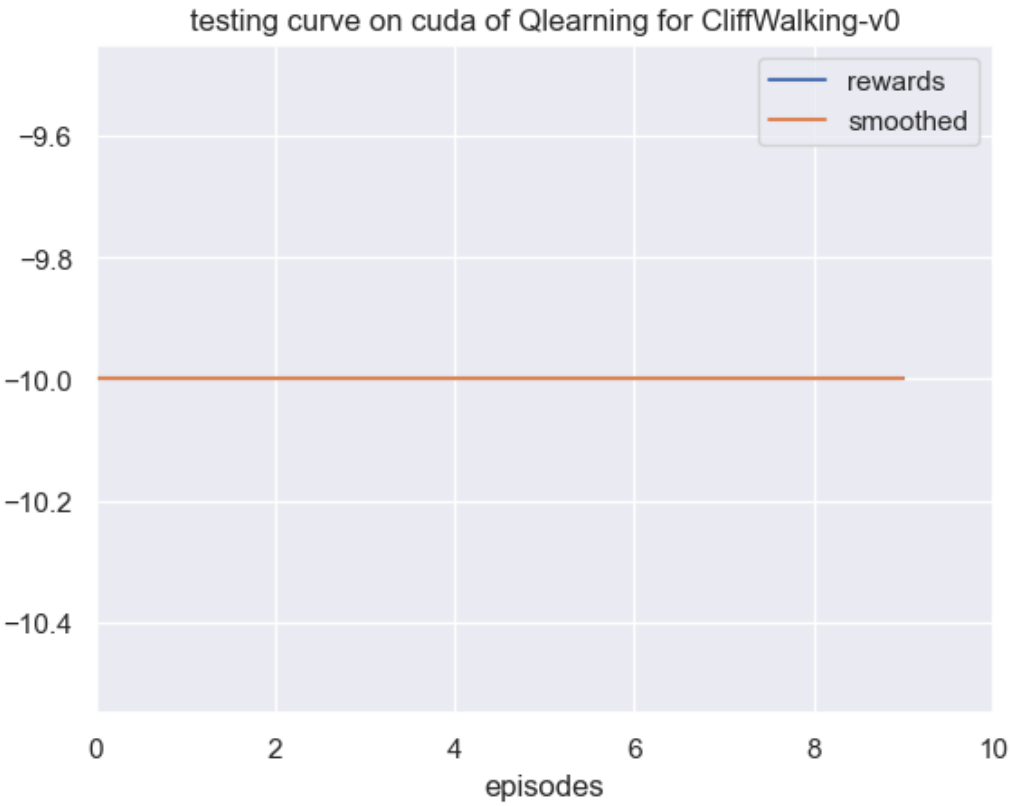

图 5.9 Q-learning 算法消融实验训练曲线

练习题

什么是 Q 值的过估计？有什么缓解的方法吗？

Q-learning 直接拿最大的未来动作价值的$\gamma \max_a Q(s_{t+1}, a)$来估计的，而在状态价值函数更新中相当于使用对应的平均值来估计的，这会使状态价值函数中的估计值更不准确。

可以使用Sarsa算法，避免直接使用最大的未来动作价值。

on-policy 与 off-policy 之间的区别是什么？

on-policy是目标策略和行为策略是同一个策略，而off-policy是将目标函数和行为函数分开。

为什么需要探索策略？

寻找最优策略