# **[强化学习]蒙特卡洛方法**

#### 作者：莫比

### 引言

蒙特卡洛方法并非是一个特定的算法，而是一类随机算法的统称，其基于思想是：用事件发生的“频率”来替代事件发生的“概率”。在机器学习中，这种方法可以用于模型是未知的情况中，它只需要从经验中去学习，这个经验包括样本序列的状态、动作和奖励。得到若干经验后，通过平均所有样本的回报来解决强化学习的任务。

### 1.策略评估

我们在给定策略的情况下，可以用蒙特卡洛方法来学习状态价值函数，也就是这个状态开始的期望回报——期望的累积未来折扣奖励，公式如下：

也就是说，我们想要估计 的值，即遵循策略 的情况下，状态 的价值，我们可以计算所有回合中首次访问状态 的平均回报， 以此作为 的估计值，这种方法就是首次访问MC方法； 与之对应的，另一种方法计算所有回合中每次访问状态 的平均回报，就是每次访问MC方法。

而去估计 的值，就是要去估计。因为在模型未知的情况下，状态值函数不能够直接决定策略，我们要通过状态动作值函数去决定。我们这两种评估方法去估计就是求所有episode访问所得到回报的均值。

首次访问MC算法流程：

输入：用来评估的策略

初始化：

对所有 ∈，任意

一个空的列表，对所有 ∈

一直循环（对每一个回合）：

使用生成一个回合：

对于回合中的每一步循环，：

除非 出现在 中：

将添加到列表中

### 2.策略控制

#### 2.1问题探讨

在我们得到值函数之后，下一步就是进行提升，去近似最优值函数和最优策略。

图1.策略控制的过程

策略提升的方法是针对当前的价值函数，即使策略贪婪。我们只需要对每个选择使动作价值函数最大的那个动作：

然后我们对于每个都取的贪婪，这样我们就可以得到：

也就是说每个都比更好，只要经过足够多的回合，就能收敛到最优的策略和价值函数。而想要这个过程收敛就必须满足下面两个假设：

（a）我们有无限个回合供策略评估使用。

（b）回合都是探索开端的方式。

#### 2.2解决第一个假设

要经过无限个episode显然是不可能的，这里我们最好的方法就是去拟合，拟合的主要方法有以下两种：

方法一：让每次策略评估都无限接近。使用一些方法和一些假设，并且经过足够多的步骤后， 就可以保证一定程度的收敛

方法二：在跳转到策略提升前，放弃尝试完成策略评估。评估的每一步，我们将价值函数向移动。一个极端的例子是价值迭代，就是每执行一步策略提升就要执行一步迭代策略评估。

#### 2.3解决第二个假设

解决第二个假设具体来讲有两种方法，我们称之为在策略(on-policy)方法和离策略（off-policy）。on-policy方法尝试去估计和提升我们用作决策的那个策略相同；而off-policy估计和提升的策略与用来生成数据的策略不同。

#### 2.3.1 on-policy策略

具体我们使用的是策略，这种算法其实就是权衡开发与探索。即在大多数时间选择有最大估计动作价值的动作，但仍有的概率选择随机的动作。

on-policy首次访问MC控制算法流程：

初始化:

任意 策略

对所有的 ，任意

对所有的， 空列表

一直循环：

遵循策略 ，生成一个回合：

对于这个回合中的每一步，：

除非 出现在 中：

将添加到列表中

对所有:

#### 2.3.2 off-policy策略

off-policy在策略估计和策略提升的时候使用两种策略，一个是行为策略 ：具有探索性的策略，专门用于产生episode积累经验；另一个则是目标策略 ：对行为策略产生的经验进行学习，使奖励最大化，成为最优策略 。

#### 2.3.2.1 off-policy策略中的重要性采样

重要性采样就是去估计随机变量在一个分布上的期望值，但是采样的样本来自另一个分布。 离策略上应用重要性采样的方法，是为了根据在目标策略和行为策略下得到发生的事件轨迹的概率比，将得到的概率对回报进行加权。 两个概率的比值称为重要性采样率。给定初始状态$\S\_t$，那么在策略下， 接下来的状态动作轨迹 发生的概率是

其中代表状态转移概率函数。因此，在目标策略和行为策略下的重要性采样率为：

从上式可以看出重要性采样率最终仅仅依赖于两个策略和序列，而与MDP无关。接下来就是off-policy评估策略的公式：

（1）原始重要性采样

（2）加权重要性采样

其中:代表所有状态 s 在某个 episode 中第一次被 visit 的时刻的集合

:从时刻t到的回报

：属于状态s的回报

：代表相应的重要性采样率

#### 2.3.2.2增量式求均值

我们除了直接求均值的方式，还可以增量式求均值。假设我们得到了一系列回报,对于off-policy来说，因为我们利用了重要性采样，所以多了一个权重的因素，设每个回报的权重为

于是有

#### 2.3.2.3 off-policy策略算法

综合前面的重要性采样和增量式求均值，我们就可以得到off-policy算法。这里我们的行为策略用的是，目标策略是贪婪算法。

off-policy首次访问MC控制算法流程：

初始化:对所有：

（随机值）

一直循环（对每一个回合）：

任何覆盖的策略

使用策略生成一个回合：

对回合的每一步循环，，：

如果则退出内循环（进行下一个回合）

### 3.实例 21点

#### 3.1游戏规则

21点的游戏规则是这样的：游戏里有一个玩家（player）和一个庄家（dealer），每个回合的结果可能是玩家获胜、庄家获胜或打成平手。回合开始时，玩家和庄家各有两张牌，玩家可以看到玩家的两张牌和庄家的其中一张牌。接着，玩家可以选择是不是要更多的牌。如果选择要更多的牌（称为“hit”），玩家可以再得到一张牌，并统计玩家手上所有牌的点数之和。其中牌面A代表1点或11点。如果点数和大于21，则称玩家输掉这一回合，庄家获胜；如果点数和小等于21，那么玩家可以再次决定是否要更多的牌，直到玩家不再要更多的牌。如果玩家在总点数小等于21的情况下不要更多的牌，那么这时候玩家手上的总点数就是最终玩家的点数。接下来，庄家展示其没有显示的那张牌，并且在其点数小于17的情况下抽取更多的牌。如果庄家在抽取的过程中总点数超过21，则庄家输掉这一回合，玩家获胜；如果最终庄家的总点数小于等于21，则比较玩家的总点数和庄家的总点数。如果玩家的总点数大于庄家的总点数，则玩家获胜；如果玩家和庄家的总点数相同，则为平局；如果玩家的总点数小于庄家的总点数，则庄家获胜。

#### 3.2代码实现

这里我使用的是off—policy策略，其实对于蒙特卡罗方法，最主要的就是解决策略评估和策略控制。下面我将给出实验代码：

def obs2state(obs):  
 # 将观测信息转换为状态信息  
 return (obs[0], obs[1], int(obs[2]))  
#策略评估  
def evaluate(env, target\_policy, behavior\_policy, episode\_num=500000):  
 #初始化  
 q = np.zeros\_like(target\_policy)#q（s,a）  
 c = np.zeros\_like(target\_policy)#前 n 个回报下每个状态的累积权值  
 for \_ in range(episode\_num):  
 state\_actions = []#状态动作键值对  
 observation = env.reset()#获取观测值  
 #使用行为策略生成一个回合  
 while True:  
 state = obs2state(observation)  
 action = np.random.choice(env. action\_space.n, p=behavior\_policy[state])  
 state\_actions.append((state, action))  
 obs, reward, done, \_ = env.step(action)  
 if done:  
 break  
 g = 0 # 回报  
 rho = 1. # 重要性采样比率  
 for state, action in reversed(state\_actions):  
 g = gamma\*g+reward #G←γG+Rt+1  
 c[state][action] += rho #C(St,At)←C(St,At)+W  
 q[state][action] += (rho / c[state][action]\*(g - q[state][action]))#Q(St,At)←Q(St,At)+W/C(St,At)[G−Q(St,At)]  
 rho \*= (target\_policy[state][action]/ behavior\_policy[state][action])  
 #W←W\*π(At|St)/b(At|St)  
 if rho == 0:  
 break  
 return q  
  
#策略控制  
def off\_policy(env,target\_policy,behavior\_policy,espisode\_num=500):  
 q=np.zeros\_like(target\_policy)  
 c=np.zeros\_like(target\_policy)  
 for i in range(espisode\_num):  
 state\_action=[]  
 obs=env.reset()  
 while True:  
 state = obs2state(obs)  
 action = np.random.choice(env.action\_space.n,p=behavior\_policy[state])  
 state\_action.append((state,action))  
 observation, reward, done, \_ = env.step(action)  
 if done:  
 break  
 #完成了一个episode  
 g=0 # 回报  
 rho=1#重要性采样比率  
 for state,action in reversed(state\_action):  
 g = gamma\*g+reward #G←γG+Rt+1  
 c[state][action]+=rho#C(St,At)←C(St,At)+W  
 q[state][action]+=(rho / c[state][action]\*(g - q[state][action]))#Q(St,At)←Q(St,At)+W/C(St,At)[G−Q(St,At)]  
 #策略提升 π(St)←argmaxaQ(St,a)  
 a =q[state].argmax()  
 target\_policy[state]=0  
 target\_policy[state][a]=1  
 if a!=action:  
 break  
 rho /= behavior\_policy[state][action]  
 return target\_policy,q

#### 3.3实验结果

图2.一个episode

其中一个episode如图，最开始玩家获得[1,4]的牌，庄家显示了5，策略决定要牌（动作为1），这时候玩家的牌就为[1,4,9]，奖励为0，策略继续决定要牌，结果下一回合的观测得到玩家的牌总和为23点，超过21点，所以游戏结束，奖励-1，庄家获胜。

最优策略图如下：

图3.最优策略图

with ace：使用了ace，即a当1；without ace：没使用ace，即a当11，可以看出在最优策略图中，without ace：大部分情况都选择不加牌，with ace：在玩家总和小于18时，大概率都是选择继续加牌。

### 4总结

（a）蒙特卡洛方法是一个用于估计价值函数和发现最优策略的学习方法。与DP不同的是，我们不需要对环境的完全了解。蒙特卡洛方法只需要状态、动作和与环境实际或模拟交互的奖励的经验样本序列。

（b）蒙特卡洛方法对每个状态 - 动作对的回报进行采样和平均。

（c）策略评估通过平均回报估计状态动作值函数。策略提升使用贪婪策略

（d）on-policy和off-policy：on-policy方法尝试去估计和提升我们用作决策的那个策略相同；而off-policy估计和提升的策略与用来生成数据的策略不同。

### 5参考文献

[1]Reinforcement Learning

* [](https://imgchr.com/i/dOkr60)

[行者AI（成都潜在人工智能科技有限公司，xingzhe.ai）](https://xingzhe.ai)致力于使用人工智能和机器学习技术提高游戏和文娱行业的生产力，并持续改善行业的用户体验。我们有内容安全团队、游戏机器人团队、数据平台团队、智能音乐团队和自动化测试团队。 > >如果您对世界拥有强烈的好奇心，不畏惧挑战性问题；能够容忍摸索过程中的各种不确定性、并且坚持下去；能够寻找创新的方式来应对挑战，并同时拥有事无巨细的责任心以确保解决方案的有效执行。那么请将您的个人简历、相关的工作成果及您具体感兴趣的职位提交给我们。

我们欢迎拥抱挑战、并具有创新思维的人才加入我们的团队。请联系：*hr@xingzhe.ai*

如果您有任何关于内容安全、游戏机器人、数据平台、智能音乐和自动化测试方面的需求，我们也非常荣幸能为您服务。可以联系：*contact@xingzhe.ai*