**强化学习理论学习与代码实现**

# 蒙特卡洛方法

不同于前面的章节，这里不再假设具备环境的完整知识，蒙特卡洛方法仅仅需要经验——从与环境的实际或仿真交互中得到的状态、动作和奖励采样序列。从实际经验中学习是惊人的，因为它不需要事先了解环境的动态，但仍然可以获得最优行为，从仿真经验中学习同样有用。虽然模型是必需的，但是模型只需要生成样本转移，而不需要生成动态规划(DP)所需的所有可能转移的完整概率分布。令人惊讶的是，在许多情况下，根据期望的概率分布很容易产生抽样的经验，但无法得到显式形式的分布。

蒙特卡罗方法是一种基于平均采样回报的强化学习方法，为了确保能够得到定义良好的回报，这里我们仅为片段任务定义蒙特卡罗方法。也就是说，我们假设经验被划分为多个片段，并且无论选择什么动作，所有片段最终都会结束。只有在一个片段完成时，价值评估和策略才会发生变化。因此，蒙特卡罗方法是在片段与片段上递增的，而不是时间步与时间步(在线)。

为了处理非平稳性问题，我们采用了第三章为DP开发的广义策略迭代(GPI)思想，在DP中我们从MDP的知识中计算值函数，而在这里，我们从MDP的样本回报中学习值函数。值函数和相应的策略仍然以本质上相同的方式（GPI）交互以获得最优性。在DP章中，我们首先考虑预测问题(固定任意策略的v和q的计算)，然后考虑策略改进，最后考虑控制问题及其用GPI求解。从DP中提取的每一个概念都被扩展到蒙特卡罗的情况，此时只有样本经验是可用的。

## 学习目标

* 理解预测与控制之间的不同；
* 知道如何使用MC方法预测状态值和状态-动作值；
* 理解在策略初访MC控制算法；
* 理解离策略MC控制算法；
* 理解加权重要性采样；
* 理解MC算法相较于动态规划法的优点。

## 蒙特卡洛预测

首先考虑蒙特卡洛方法用于给定策略下状态值函数的学习，一个状态的值为期望的回报，即从该状态开始的期望累积未来折扣奖励。因此，根据经验估计状态值的一个显而易见的方法是，简单地对访问该状态后观察到的回报进行平均，当观察到更多的回报时，平均值应该收敛到期望值。这个思想是所有蒙特卡罗方法的基础。

假设希望估计vπ(s)，即为给定通过遵循π和经过s获得的一些列片段的情况下，策略π下状态s的值。一个片段中每次状态s的出现称为对s的一次访问（visit），当然s可能在一个片段中可以被访问多次，将一个片段中的第一次访问称为对s的初访（first-visit）。初访蒙特卡洛方法使用初访状态s时回报的平均值估计vπ(s)，而每次访问（every-visit）蒙特卡洛方法对所有访问s时的回报取平均值。这两种方法非常类似，但有稍微不同的理论性质。初访MC的研究最为广泛，可以追溯到20世纪40年代，也是我们在本章中重点研究的对象，每次访问MC可以更自然地扩展到函数逼近和资格迹。以下为初访MC的伪代码，每次访问MC的不同在于不需要检查在片段中St是否之前经历过。

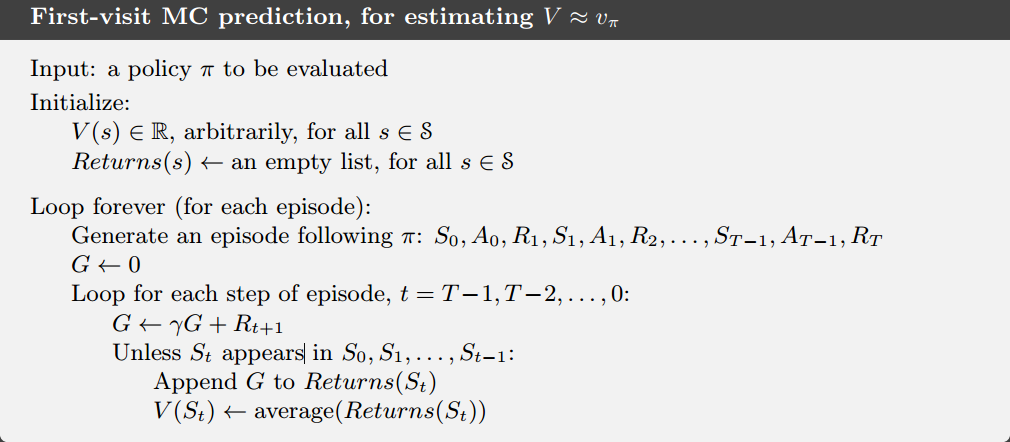


图1 初访MC算法

随着对s的访问（或者初访）趋于无穷，初访MC和每次访问MC都会收敛到vπ(s)，对于初访MC这是显而易见的，在这种情况下，每个回报都是有限方差下vπ(s)的独立同分布估计，根据大数定律，这些估计值的平均值序列收敛于其期望值，每一个平均值本身为无偏估计，误差的标准差依1/，其中n需要平均的回报数量。每次访问MC不是那么直截了当，但是其估计值也会二次收敛到vπ(s)。

## 蒙特卡洛动作值估计

如果无法获取模型，那么估计动作值（状态-动作对的值）比估计状态值有用的多。利用模型，仅使用状态值就足以确定策略，即通过一步前向观察，然后选择能带来最优的奖励和下一状态组合的动作。而在无模型的情况下，单凭状态值是不够的，还必须显示地估计每个动作的值，以便用于确定策略。因此蒙特卡洛方法的一个重要目标就是估计q\*，为此首先考虑动作值的策略评估问题。

动作值的策略评估问题就是估计qπ(s,a)，即从状态s开始，采取动作a，之后遵循策略π所能得到的期望回报。动作值估计的蒙特卡洛方法本质上与状态值估计相同，只是现在讨论的是对状态-动作对的访问，而不再是状态。如果状态s被访问并在该状态下执行动作s，则称在片段中对状态-动作对s,a进行了一次访问。每次访问MC方法将状态-动作对的值估计为对其所有访问之后回报的平均值，而初访MC中用于求平均的是每个片段中第一次访问的状态和采取的动作所获得的回报。和 之前一样，当访问每个状态-动作对的次数接近无穷时，这些方法以二次方式收敛到真实期望值。

唯一复杂的是，许多状态-动作对可能永远不会被访问。如果π是一个确定性策略，那么在遵循π时，将只观察来自每个状态的其中一个动作的返回。如果没有回报进行平均，其他动作的蒙特卡罗估计不会随着经验而改善。这是一个严重的问题，因为学习动作值的目的是帮助在每个状态中的可选动作中进行选择，为了比较这些可选方案，我们需要评估来自每个状态的所有动作的值，而不仅仅是当前偏好的动作。

这就是普遍存在的“maintaining exploration”问题，对于动作的策略评估问题，必须保证不断地进行探索，一种方式就是让每个状态-动作对都有一定的概率（大于0）被选中作为每个片段的开始节点，这样就能保证随着片段趋于无穷，所有的状态-动作对都能被访问无限次，我们将这个假设称为“exploring starts”。

这个“exploring starts”的假设有时是有用的，但在一般情况下却是不可靠的，特别是当直接与环境的实际交互中学习时，在那种情况下开始条件不太可能有太大帮助，因为真实情况下我们是不能去指定开始节点的。为了保证访问到所有的状态-动作对，最常用的方法是只考虑在每个状态下对所有动作选择概率都不为0的随机策略。在后面几节会讨论这种方法的两个重要变体，但是现在先保留“exploring starts”的假设并展示完整的蒙特卡洛控制方法。

## 蒙特卡洛控制

现在将考虑蒙特卡洛估计如何用于控制，也就是近似最优策略，总体思路是按照与DP章节相同的模式进行，即按照广义策略迭代（GPI）的思想。在GPI中，同时维护了近似策略和近似值函数，值函数反复更新以更逼近当前策略下的值函数，策略也利用当前值函数不断改进，如下图所示。这两种更新在某种程度上是相互矛盾的，因为它们都为另一种创建了一个移动的目标，但是它们一起又使得策略和值函数都接近于最优。

首先，考虑经典策略迭代的蒙特卡洛版本，在这个方法中，交替执行完整的策略评估和策略改进，这个过程以任意策略π0开始，而以最优策略和最优值函数结束：



其中E表示一次完成的策略评估，I表示一次完整的策略改进。策略评估完全按照前面的章节中描述的那样进行，经历许多片段后，近似值函数渐进地逼近真实值函数。现在，假设确实观察了无数个片段，而且这些片段是使用起始点探索（exploring starts）产生的。在这些假设下，蒙特卡洛法将对任意的πk精确地计算qπk。

策略改进是通过对当前值函数贪婪地制定策略实现的，这种情况下就获得了动作值函数，因此不需要模型来构建贪婪策略。对任意的动作值函数q，对应的贪婪策略是对任意s∈S，可以确定性地选择使得动作值最大的动作的策略：

 （1）

然后通过构建每一个πk+1作为qπk的贪婪策略，将策略改进理论应用于πk和πk+1，因为对于所有的s∈S，



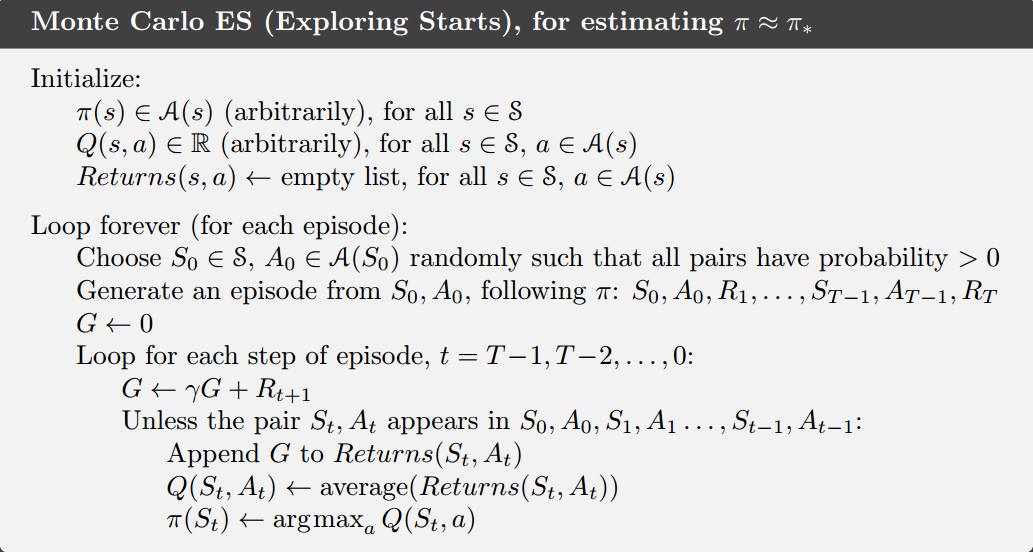
正如前面的章节中讨论的那样，策略改进理论保证了πk+1一律不劣于πk，或当两者都已经是最优策略时就一样好，这又保证了整个过程收敛到最优策略和最优值函数，通过这种方式，蒙特卡洛法可以在仅给定采样片段且没有环境动态知识的情况下，用于寻找最优策略。

为了简单地获得蒙特卡洛法的收敛性保证，这里我们做了两个不太可能实现的假设，一个是片段具有起始点探索，另一个是可以进行无限次策略评估。为了得到一个实用的算法，我们必须去掉这两个假设。我们把对第一个假设的考虑推迟到本章后面。

现在我们关注的假设是策略评估运行在无限次片段上，这个假设相对容易移除。实际上，同样的问题也会出现在经典的DP方法中，比如迭代策略评估，它也只渐进地收敛于真实值函数，对于DP和MC可以有两种方法解决这个问题。一种是坚持每一次策略评估中逼近qπk的观点，通过测量和假设来获得估计误差的大小和概率的界限，然后在每次策略评估期间采取足够的步数来确保这些界限足够小。这种方法可能完全令人满意，因为它在某种程度上保证了正确的收敛。然而，除了最小的问题之外，它还可能需要太多的片段才能在实践中发挥作用。

还有一种方法可以避免策略评估在名义上所需的无限个片段，在返回到策略改进之前不再试图完成策略评估，而是在每一评估步将值函数移向qπk，同时也不期望能真正接近目标，除非经历过很多步，当第一次介绍GPI时也利用了该思想，这种思想的一种极端方式就是值迭代，在每一步策略改进之间只进行一次迭代策略评估，而对于“in place”版本的值迭代则更加极端，对于单个状态，交替使用改进和评估步骤。

对于蒙特卡洛策略迭代，可以很自然地在逐个片段的基础上交替执行评估和改进，每一片段之后，使用观察到的回报评估策略，然后改进片段中所有访问到的状态上的策略。下面给出了一种完整的简单算法-蒙特卡洛ES（带有初始探索的蒙特卡洛）。

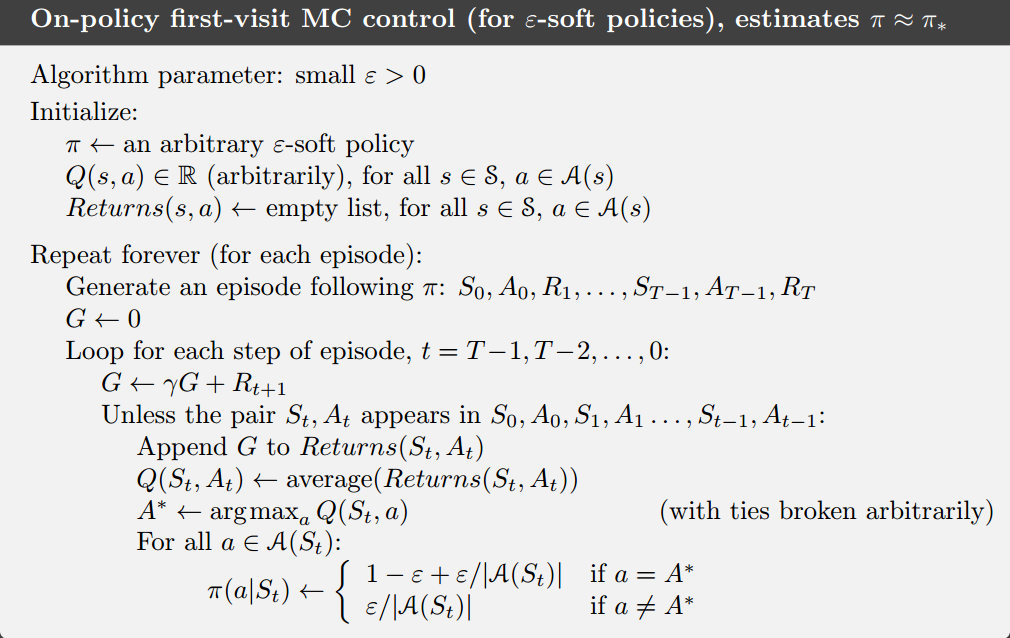


## 不带初始探索的蒙特卡洛控制

如何才能避免像初始探索这种不切实际的假设呢？确保所有动作都被选择无限次的唯一通用方法是让代理持续选择，有两种方法可以保证这一点。也就是所谓的在策略（on-policy）方法和离策略（off-policy）方法。在策略方法尝试评估或改进用于决策的策略，而离策略用于评估或改进与生成数据的策略不同的策略。上述蒙特卡罗ES是一种基于在策略的方法，在本节中，我们将展示如何在不使用不切实际的初始探索假设的情况下，设计在策略的蒙特卡罗控制方法。

在在策略控制方法中，策略通常是不确定的（soft），即对于所有的状态s∈S和所有的动作a∈A，都有π(a|s)>0，但会逐渐向确定性最优策略靠拢。我们在本节中介绍的在策略方法使用ε-贪婪策略，这意味着大多数情况下，选择能最大化估计动作值的动作，但是以概率ε随机选择动作。也就是说，所有的非贪婪动作给定最小的选择概率，剩下的概率分配给贪婪动作。ε-soft策略的定义为：在某个ε>0，对于所有的状态和动作均满足的策略，ε-贪婪策略就是ε-soft策略的一种，在ε-soft策略中，ε-贪婪策略是最接近贪婪的。

在策略蒙特卡洛控制的思想仍然是GPI，同蒙特卡洛ES一样，使用初访MC方法评估当前策略的动作值函数，然而由于缺乏初始探索假设，就不能简单地通过对当前值函数进行贪婪优化来改进策略，否则无法进一步探索非贪婪的动作。幸运的是，GPI并不要求整个过程中遵循的策略是贪婪的，而只需要它能逐渐逼近贪婪策略。在在策略方法中，仅仅改为遵循ε-贪婪策略。对于任意ε-soft策略π，根据qπ生成的任意ε-贪婪策略保证优于或等于π。下面给出了完整算法。



根据策略改进定理，对于一个ε-soft策略π，根据qπ生成的任意ε-贪婪策略都是对其的一个改进，假设π’为ε-贪婪策略。策略改进定力成立，因为对于任意的s∈S：

 （2）

（由于期望累加和是用和为1的非负权重进行的加权平均，所以一定小于等于最大值）



因此根据策略改进定理，π’≥π，即对于所有s∈S，都有vπ’(s)≥vπ(s)。下面证明该式的等号成立的条件是：当且仅当π’和π最优的ε-soft策略，即它们都比其他的ε-soft策略更优或相同。

考虑一个类似于原先环境的新环境，区别在于现在将策略是ε-soft这一要求“移入”环境中。新环境与原先环境具有相同的动作和状态空间，并按如下方式执行动作。如果在状态s时执行动作a，那么新环境将以1-ε的概率与原先环境表现完全相同，以ε概率重新随机等概率选择动作，然后具有和原先环境采取这一新的随机动作一样的表现。在新环境中的最优策略的表现与原先环境中最优的ε-soft策略的表现是相同的。令和为新环境的最优值函数，则当且仅当vπ=时，策略π是最优的ε-soft策略。根据的定义其可以是下式的唯一解：



当等式成立且ε-soft策略无法再改进时，根据式（2）有



然而，除了使用vπ代替以外，这个等式和上一个完全相同。由于是唯一解，因此一定有vπ=。

## 基于重要度采样的离策略预测

所有的学习控制方法都面临一个困境：它们希望学习到可以使随后的行为是最优的动作值函数，但是为了探索所有的动作（以找到最优动作），需要采取非最优的行为。如何在遵循探索性策略的同时学习最优策略呢？前面小节中介绍的在策略方法实际上是一种妥协—它并不学习最优层面的动作值，而是学习一个近似最优且仍在探索的策略的动作值。一种更为直接的方法是使用两个策略，一个是用来学习并最终成为最优策略，另一个更具有探索性，用于生成行为。用来学习的策略称为目标策略（target policy），用于生成行为的策略称为行为策略（behavior policy）。在这种情况下，认为学习所用的数据是“偏离”目标策略的，因此整个过程被称为离策略学习（off-policy learning）。

本节中通过讨论预测问题来开始对离策略的学习，在预测问题中目标策略和行为策略都是固定的。假设希望估计vπ或qπ，但是只有遵循另一个策略b（b≠π）所产生的片段，这种情况下，π是目标策略，b是行为策略，两个策略均固定且已知。

几乎所有的离策略方法都采用了重要度采样（importance sampling），这是一种给定来自其他分布的样本条件下，估计某种分布的期望值的通用方法。我们将重要度采用应用于离策略学习，对报酬根据其轨迹在目标策略和行为策略中出现的相对概率进行加权，这个相对概率称为重要度采样比（importance-sampling ratio）。给定起始状态St，后续的状态-动作轨迹At，St+1，At+1，...，ST在策略π发生的概率为



因此在目标策略和行为策略轨迹下的相对概率（重要度采样比）是

 （3）

尽管轨迹概率取决于MDP的转移概率，且MDP的转移转移概率通常是未知的，但是它在分子和分母中是完全相同的，因此可以被约分。最终重要度采样比只与两个策略和样本序列有关，而与MDP无关。

回想一下，我们希望评估目标策略下的期望回报，但是只有行为策略下的回报Gt。这些从行为策略中得到的回报的期望是不准确的，因此不能用它们的平均来得到vπ。这里就要用到重要度采样了，使用比例系数ρt:T-1可以调整回报使其具有正确的期望值：

 （4）

下面给出通过观察到的一批遵循策略b的多个片段并将其回报平均来预测vπ(s)的蒙特卡洛算法。为了预测vπ(s)，只需要根据重要性采样比来调整回报值并对结果进行平均即可：

 （5）

通过这样一种简单平均实现的重要度采样被称为普通重要性采样（ordinary importance sampling）。

另一种重要的方法是加权重要度采样（weighted importance sampling），它采用加权平均，定义如下：

 （6）

如果分母为0，式（6）的值也为0。为了理解两种重要性采样方法的区别，讨论它们在初访方法下，从状态s开始观察单一一个片段回报后的估计值。在加权平均的估计中，比例系数ρt:T(t)-1在分子和分母中被约分，所以估计值等于观察到的回报，而与重要度采样比无关（假设比例系数不为0）。考虑到这个回报是唯一的观测结果，所以这是一个合理的估计，但其期望是vb(s)而不是vπ(s)，在统计学意义上这个估计是有偏的。相反，普通重要度采样的初访版本得到的结果在期望上总是vπ(s)（无偏的），但是其值可能变得很极端。假设比例系数为10，这意味着，观察到的轨迹遵循目标策略的可能性是遵循行为策略的10倍，在这种情况下，普通重要度采样的估计值会是观察到的回报值的10倍，也就是说，尽管该片段的轨迹应该可以非常有效地反映目标策略，但是其估计值却会偏离观察到的回报值很远。

在数学上，两种重要度采样方法在初访下的差异可以用偏差和方差来表示。普通重要度采样是无偏的，而加权重要度采样是有偏的（尽管偏差监近收敛于0）。另一方面，普通重要度采样的方差一般是无界的，因为重要度采样比的方差是无界的，而在加权估计中任何单个回报的最大权重为1。实际上，如果假设回报值有界，那么即时重要度采样比的方差是无穷的，加权重要度采样的方差也会收敛到0。在实际中，加权估计具有很小的方差，因而在应用中首选。尽管如此，也不会完全丢弃普通重要度采样，因为它更容易扩展到基于值函数的近似方法。

普通和加权重要度采样的每次访问方法都是有偏的，但是偏差随着采样数量的增加渐近于0。在实际中，每次访问方法更受欢迎，因为其不需要记录访问过哪些状态，且易于扩展到近似方法。

## 增量式实现

蒙特卡洛预测方法可以通过逐个片段地进行增量实现，对于离策略蒙特卡洛方法，需要分别讨论使用普通重要度采样和加权重要度采样的方法。

对于普通重要度采样，回报先通过重要度采样比ρt:T(t)-1进行缩放然后再简单地取平均（式5）。

对于加权重要度采样，假设有一个回报序列G1,G2,...Gn-1，它们从相同的状态开始，且每一个回报都对应一个随机权重Wi（如Wi=ρti:T(ti)-1）。我们希望得到如下的估计，并在获得了额外的回报值Gn时保持更新。

 （7）

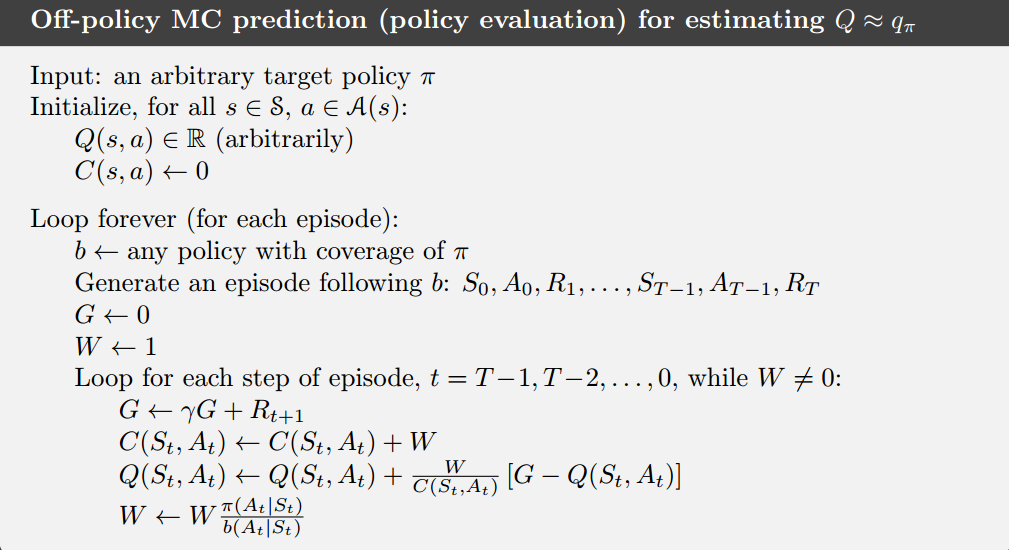
为了追踪Vn的变化，必须为每一个状态维护前n个回报对应权重的累加和Cn。Vn的更新规则为：

 （8）

和



其中（V1是任意的，不需要特别指定）。下面给出了完整的蒙特卡洛策略评估的逐片段增量算法。该算法表面上针对于离策略的情况，采用加权重要度采样，但是同样适用于在策略的情况，只需要选择相同的目标策略和行为策略即可（这种情况下π=b，W始终为1）。近似值Q收敛于qπ（对于所有遇到的“状态-动作”对），而动作则根据b进行选择，这个b可能是一个不同的策略。

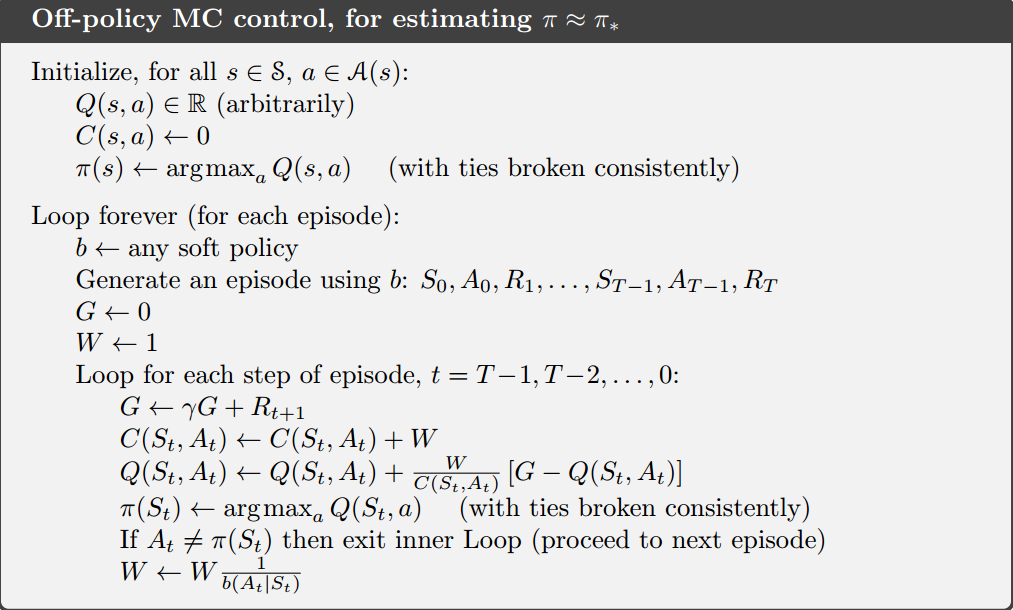


## 离策略蒙特卡洛控制

下面讨论第二类学习控制方法—离策略方法。在策略方法与其的区别在于，在策略方法在使用某个策略进行控制的同时也对其进行价值评估。在离策略方法中，这两个过程是分离的。用于生成行为数据的策略称为行为策略，行为策略可能与实际上待评估和改进的策略无关，而被评估和改进的策略称为目标策略。这样分离的好处在于当行为策略能对所有可能的动作继续进行采样时，目标策略可以是确定的（例如贪婪）。

蒙特卡洛控制方法使用了前两节介绍的一种技术。它们遵循行为策略同时学习和改进目标策略，这些方法要求行为策略对目标策略所有可能选择的动作都有非0的概率被选择。为了探索所有的可能性，就需要行为策略是软性的（也就是说它在所有状态下选择所有动作的概率都非0）。

下面给出了基于GPI和加权重要度采样的离策略蒙特卡洛控制方法，用于估计π\*和q\*。目标策略π≈π\*是对应Q得到的贪婪策略，其中Q是对qπ的估计。行为策略b可以是任意的策略，但为了保证π能收敛到最优策略，对每一状态-动作对都需要取得无穷多次回报，这可以通过选择ε-soft的b来保证。即使动作是根据不同的soft策略b选择的，且这个策略可能在片段间甚至是片段内发生变化，策略π仍能在所有经历的状态收敛到最优。



## 代码练习

### 蒙特卡洛预测

# -\*- coding: utf-8 -\*-

"""

Created on Thu Oct 24 19:22:21 2019

@author: hba

"""

import gym

import matplotlib

import numpy as np

import sys

from collections import defaultdict

if "../" not in sys.path:

sys.path.append("../")

from Lib.envs.blackjack import BlackjackEnv

from Lib import plotting

matplotlib.style.use('ggplot')

env = BlackjackEnv()

def mc\_prediction(policy, env, num\_episodes, discount\_factor=1.0):

"""

蒙特卡洛预测算法：使用采样计算给定策略的值函数.

参数:

策略:将观察映射成动作概率的函数.

env: OpenAI gym环境.

num\_episodes: 采样的迭代次数.

discount\_factor: Gamma折扣因子.

返回值:

值函数V.

"""

# 跟踪每个状态的次数和回报以计算平均，可以使用一个数组来保存所有

# 返回(如书中所示)，但这是内存效率低下的.

returns\_sum = defaultdict(float)

returns\_count = defaultdict(float)

# The final value function

V = defaultdict(float)

# Implement this!

for i\_episode in range(1,num\_episodes+1):

#打印当前代，用于调试

if i\_episode%1000==0:

print("\rEpisode {}/{}.".format(i\_episode,num\_episodes),end="")

sys.stdout.flush()

#产生一代样本数据

#一代是由tuple(state,action,reward)组成的array数列

episode=[]

#重置得到初始状态

state=env.reset()

for t in range(100):

action=policy(state)

next\_state,reward,done,\_=env.step(action)

episode.append((state,action,reward))

if done :

break

state=next\_state

#寻找当前代已经访问过的所有状态

#将每个状态转换成tuple以用作字典（通过set将重复的元素去除）

states\_in\_episode=set([tuple(x[0])for x in episode])

for state in states\_in\_episode:

#在当前代state第一次出现的位置

first\_occurence\_idx=next(i for i,x in enumerate(episode) if x[0]==state)

#对从第一次出现开始的所有奖励进行求和

G=sum([x[2]\*(discount\_factor\*\*i) for i,x in enumerate(episode[first\_occurence\_idx:])])

#计算该状态在所有采样代数上的平均回报

returns\_sum[state]+=G

returns\_count[state]+=1.0

V[state]=returns\_sum[state]/returns\_count[state]

return V

def sample\_policy(observation):

"""

采样策略：如果玩家得分>20,那么停牌，否则叫牌.

"""

score, dealer\_score, usable\_ace = observation

return 0 if score >= 20 else 1

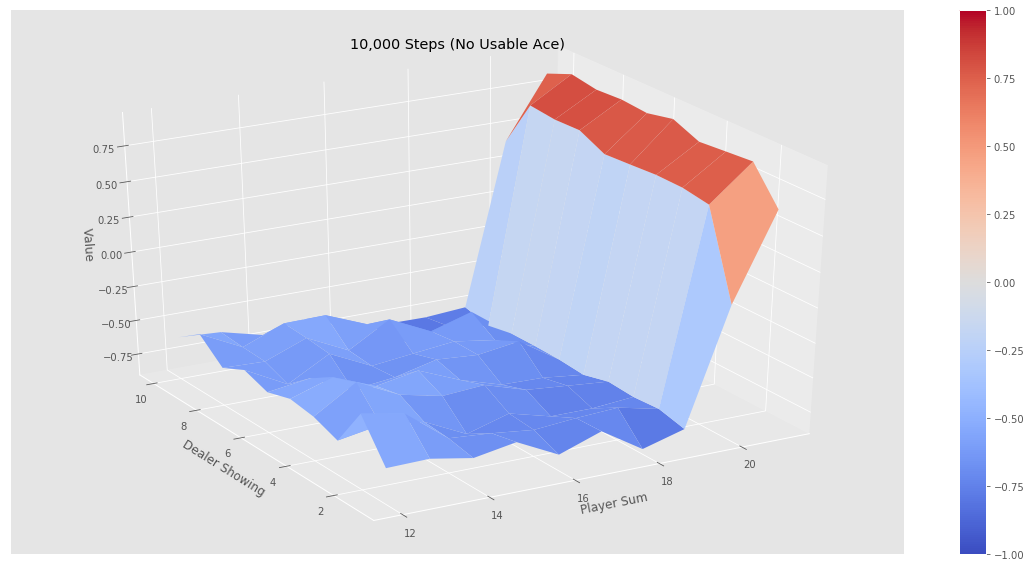
V\_10k = mc\_prediction(sample\_policy, env, num\_episodes=10000)

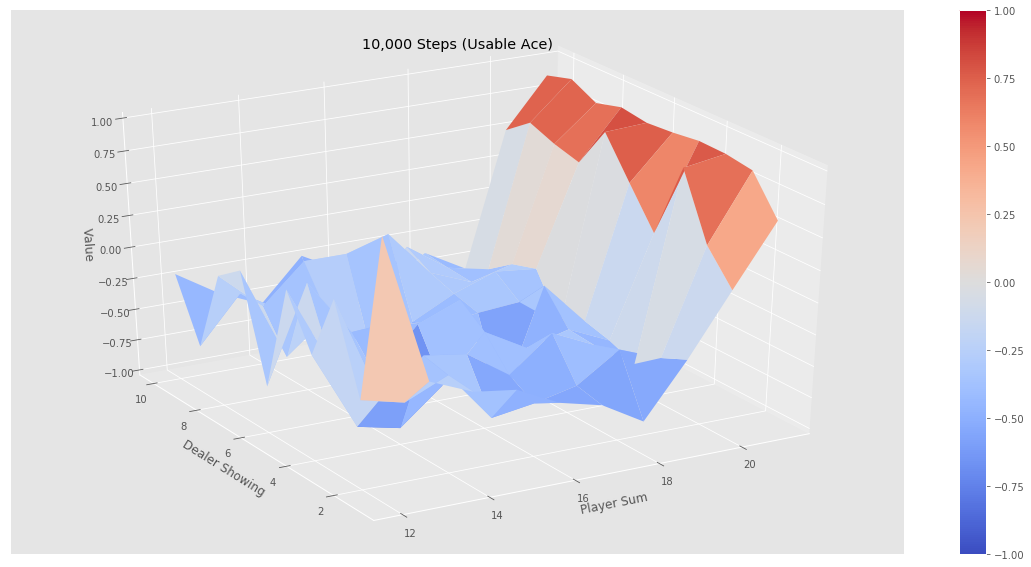
plotting.plot\_value\_function(V\_10k, title="10,000 Steps")

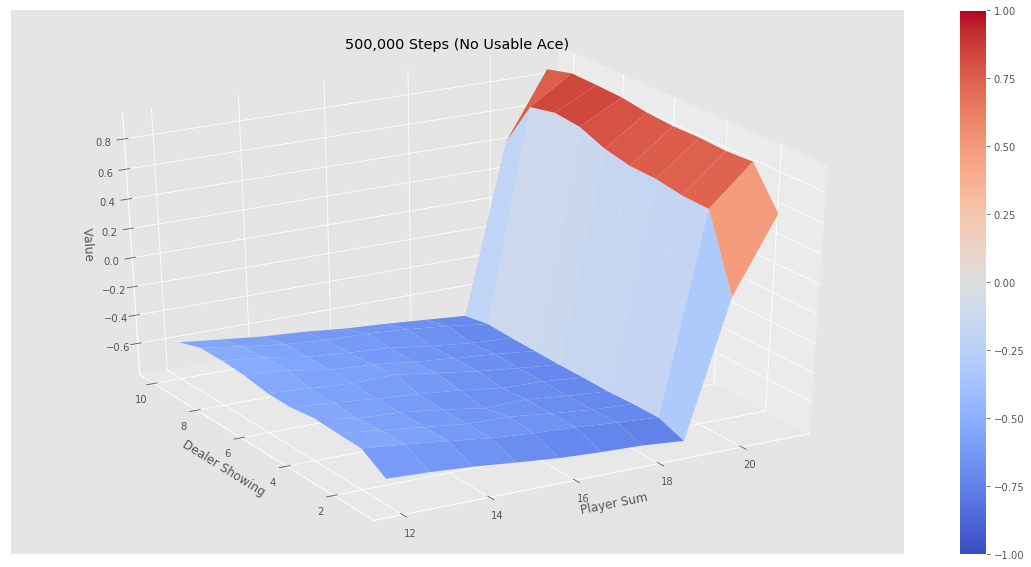
V\_500k = mc\_prediction(sample\_policy, env, num\_episodes=500000)

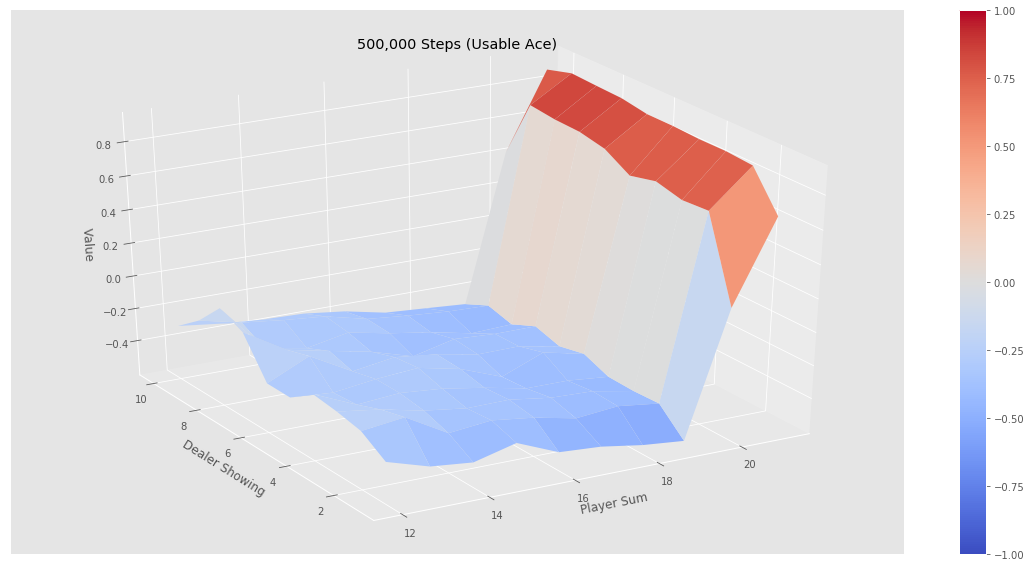
plotting.plot\_value\_function(V\_500k, title="500,000 Steps")

结果如下：









### 蒙特卡洛ε-贪婪控制

# -\*- coding: utf-8 -\*-

"""

Created on Fri Oct 25 11:09:51 2019

@author: hba

"""

import gym

import matplotlib

import numpy as np

import sys

from collections import defaultdict

if "../" not in sys.path:

sys.path.append("../")

from Lib.envs.blackjack import BlackjackEnv

from Lib import plotting

matplotlib.style.use('ggplot')

env = BlackjackEnv()

def make\_epsilon\_greedy\_policy(Q,epsilon,nA):

"""

基于给定的Q值函数和epsilon创建epsilon贪婪策略

参数：

Q:将状态映射成动作值函数的字典。每个值都是一个长度为nA的numpy数组。

epsilon：选择随机动作的概率，为0与1之间的浮点数。

nA：环境中的动作数。

返回值：

返回一个函数，该函数的输入为观察即状态，并以numpy数组（长度为nA）的形式返回每个动作的概率。

"""

def policy\_fn(obseration):

A=np.ones(nA,dtype=float)\*epsilon/nA

best\_action=np.argmax(Q[obseration])

A[best\_action]+=(1.0-epsilon)

return A

return policy\_fn

def mc\_control\_epsilon\_greedy(env,num\_episodes,discount\_factor=1.0,epsilon=0.1):

"""

使用epsilon贪婪策略进行蒙特卡洛控制，寻找最优的epsilon贪婪策略。

参数：

env：OpenAI gym环境。

num\_eposides:采样代数。

discount\_factor:Gamma折扣因子。

epsilon:选择随机动作的概率，为0与1之间的浮点数。

返回值：

返回一个元组(Q,policy).Q是将状态映射成动作值函数的字典，policy是一个函数，以obseration为输入，返回动作的概率

"""

# 跟踪每个状态的次数和回报以计算平均，可以使用一个数组来保存所有

# 返回(如书中所示)，但这是内存效率低下的.

returns\_sum = defaultdict(float)

returns\_count = defaultdict(float)

#最终的动作值函数

#为嵌套字典，将state映射为(action -> action-value)

Q=defaultdict(lambda:np.zeros(env.action\_space.n))

#遵循的策略

policy=make\_epsilon\_greedy\_policy(Q,epsilon,env.action\_space.n)

for i\_episode in range(1,num\_episodes+ 1):

#打印当前代，用于调试

if i\_episode%1000==0:

print("\rEpisode {}/{}.".format(i\_episode,num\_episodes),end="")

sys.stdout.flush()

#产生一代样本数据

#一代是由tuple(state,action,reward)组成的array数列

episode=[]

#重置得到初始状态

state=env.reset()

for t in range(100):

probs=policy(state)

action=np.random.choice(np.arange(len(probs)),p=probs)

next\_state,reward,done,\_=env.step(action)

episode.append((state,action,reward))

if done:

break

state=next\_state

#寻找当前代中所有已访问的状态-动作对

#将每个状态转换成tuple以用作字典（通过set将重复的元素去除）

sa\_in\_episode=set([(tuple(x[0]),x[1])for x in episode])

for state,action in sa\_in\_episode:

sa\_pair=(state,action)

#在当前代中寻找状态-动作对(state,action)第一次出现的位置

first\_occurence\_idx=next(i for i,x in enumerate(episode) if x[0]==state and x[1]==action)

#对从第一次出现开始的所有奖励进行求和

G=sum([x[2]\*(discount\_factor\*\*i) for i,x in enumerate(episode[first\_occurence\_idx:])])

#计算该状态在所有采样代数上的平均回报

returns\_sum[sa\_pair]+=G

returns\_count[sa\_pair]+=1.0

Q[state][action]=returns\_sum[sa\_pair]/returns\_count[sa\_pair]

#通过修改Q字典，可以隐式地改进策略

return Q,policy

Q, policy = mc\_control\_epsilon\_greedy(env, num\_episodes=500000, epsilon=0.1)

# For plotting: Create value function from action-value function

# by picking the best action at each state

V = defaultdict(float)

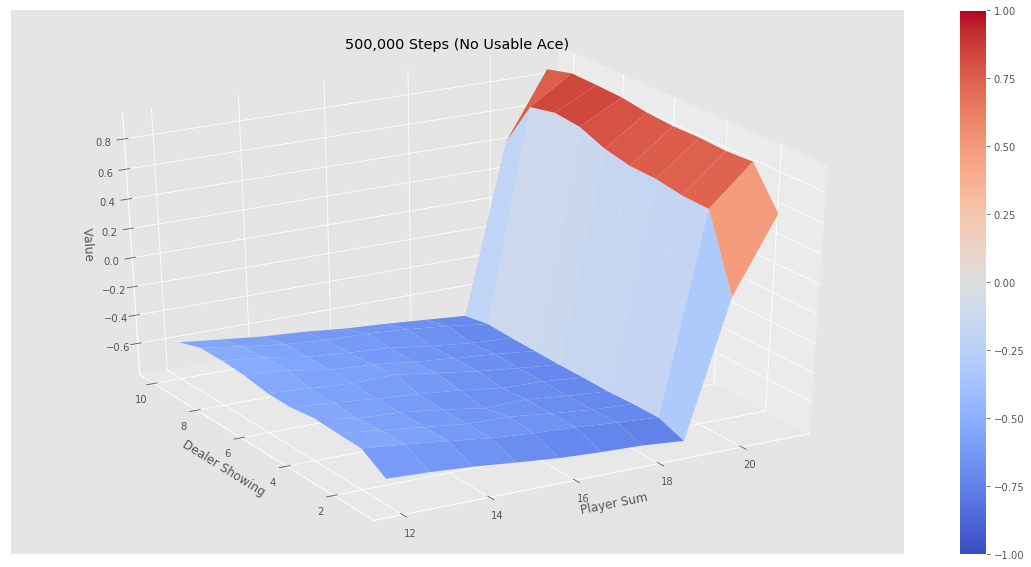
for state, actions in Q.items():

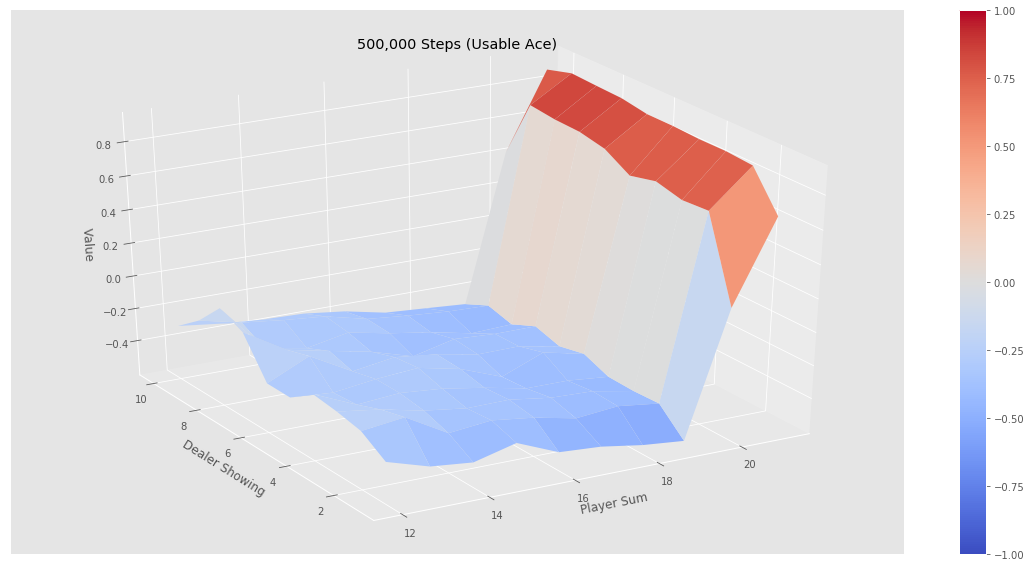
action\_value = np.max(actions)

V[state] = action\_value

plotting.plot\_value\_function(V, title="最优值函数")

结果为：





### 采样加权重要度采样的离策略蒙特卡洛控制

# -\*- coding: utf-8 -\*-

"""

Created on Fri Oct 25 15:52:35 2019

@author: hba

"""

import gym

import matplotlib

import numpy as np

import sys

from collections import defaultdict

if "../" not in sys.path:

sys.path.append("../")

from Lib.envs.blackjack import BlackjackEnv

from Lib import plotting

matplotlib.style.use('ggplot')

env = BlackjackEnv()

def create\_random\_policy(nA):

"""

创建随机策略函数.

参数:

nA: 环境中动作的数量.

返回值:

返回一个函数，该函数以观察为输入，返回一个动作概率向量.

"""

A = np.ones(nA, dtype=float) / nA

def policy\_fn(observation):

return A

return policy\_fn

def create\_greedy\_policy(Q):

"""

基于Q值创建贪婪策略.

参数:

Q: 映射状态到动作值的字典

返回值:

返回一个函数，该函数以观察为输入，返回一个动作概率向量.

"""

def policy\_fn(state):

A = np.zeros\_like(Q[state], dtype=float)

best\_action = np.argmax(Q[state])

A[best\_action] = 1.0

return A

return policy\_fn

def mc\_control\_importance\_sampling(env, num\_episodes, behavior\_policy, discount\_factor=1.0):

"""

使用加权重要度采样进行蒙特卡洛离策略控制，寻找最优贪婪策略.

参数:

env: OpenAI gym环境.

num\_episodes: 采样代数.

behavior\_policy: 生成各代样本数据时所遵循的行为策略，为给定观察下返回每个动作概率向量的函数.

discount\_factor: Gamma折扣因子.

返回值:

返回一个元组(Q, policy).

Q是映射状态到动作值的字典.

policy是一个以观察为输入，返回一个动作概率向量的函数，该策略为最优贪婪策略.

"""

#最终的动作值函数

#映射状态到动作值函数的字典

Q=defaultdict(lambda:np.zeros(env.action\_space.n))

#加权重要性抽样公式的分母累积（贯穿于所有的代）

C=defaultdict(lambda:np.zeros(env.action\_space.n))

#希望学习的贪婪策略即目标策略

target\_policy=create\_greedy\_policy(Q)

for i\_episode in range(1,num\_episodes+1):

#打印当前代，用于调试

if i\_episode%1000==0:

print("\rEpisode {}/{}.".format(i\_episode,num\_episodes),end="")

sys.stdout.flush()

#生成一代样本数据

#每一代是由元组（state,action,reward）组成的array数组

episode=[]

state=env.reset()

for t in range(100):

#从行为策略采样动作

probs=behavior\_policy(state)

action=np.random.choice(np.arange(len(probs)),p=probs)

next\_state,reward,done,\_=env.step(action)

episode.append((state,action,reward))

if done:

break

state=next\_state

#折扣回报总和

G=0.0

#重要度采样比（回报的权重）

W=1.0

#反向遍历当前代中所有的时间步(反向的目的是采样增量式求平均)

for t in range(len(episode))[::-1]:

state,action,reward=episode[t]

#更新t时间步后的总奖励

G=discount\_factor\*G+reward

#更新加权重要性抽样公式的分母

C[state][action]+=W

#使用增量更新式更新动作值函数，同时也会改善引用Q的目标策略

Q[state][action]+=(W/C[state][action])\*(G-Q[state][action])

#如果行为策略所采取的动作

if action!=np.argmax(target\_policy(state)):

break

W=W\*1./behavior\_policy(state)[action]

return Q,target\_policy

random\_policy = create\_random\_policy(env.action\_space.n)

Q, policy = mc\_control\_importance\_sampling(env, num\_episodes=500000, behavior\_policy=random\_policy)

# 输出：根据动作值函数，通过为每个状态选择最大的动作得到值函数

V = defaultdict(float)

for state, action\_values in Q.items():

action\_value = np.max(action\_values)

V[state] = action\_value

plotting.plot\_value\_function(V, title="最优值函数")

结果如下：

