chenhoujiangsir的专栏

目录视图 摘要视图 RSS 订阅

个人资料

chenhoujiangsir

访问： 4884次

积分： 116

等级： BLOG 2

排名： 千里之外

原创： 6篇 转载： 1篇

译文： 0篇 评论： 0条

文章搜索

强化学习（一）

目录(?) [+]

前言

近几年，由于DeepMind成功地将强化学习（reinforcement learning）运用在A
得了超过人类的表现，使得强化学习成为目前机器学习研究的前沿方向之一
年就已经开始研究强化学习，1998年出版了强化学习介绍一书，并于2012年
考该书。

强化学习最早主要用于智能控制领域，比如机器人控制、电梯调度、电信通
容推荐[4]和语音交互领域都有相关的应用。2013年底DeepMind发表文章Play
Learning，首次成功地将深度学习运用到强化学习任务上，通过无监督学习
效果。而后DeepMind逐渐改进算法，使得DQN在Atari几乎一半的游戏中超

文章分类

dataset (1)
cnn (4)

文章存档

2016年10月 (2)
2016年09月 (1)
2016年08月 (1)
2016年06月 (1)
2016年04月 (1)
展开

阅读排行

决策树在Kaldi中如何使用 (1272)
GoogLeNet论文学习笔记 (1068)
强化学习（一） (955)
CNN卷积前后向推导 (646)
AlexNet论文学习笔记 (399)
收集的运动目标检测，阴 (352)
tensorflow源码安装 (120)

评论排行

收集的运动目标检测，阴 (0)
CNN卷积前后向推导 (0)
决策树在Kaldi中如何使用 (0)
AlexNet论文学习笔记 (0)

人车的出现，人们惊奇地发现**人工智能**即将颠覆我们的生活，甚至有人评论说传统的深度学习已经可以很好地感知理解了，强化学习可以利用这些感知生成策略，因而可以创造更高的机器智能。

下面是DeepMind使用DQN让机器学习玩Atari 2600游戏的视频。

什么是强化学习

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal[1].

强化学习研究的是智能体agent与环境之间交互的任务，也就是让agent像人类一样通过试错，不断地学习在不同的环境下做出最优的动作，而不是有监督地直接告诉agent在什么环境下应该做出什么动作。在这里我们需要引入回报（reward）这个概念，回报是执行一个动作或一系列动作后得到的奖励，比如在游戏超级玛丽中，向上跳可以获得一个金币，也就是回报值为1，而不跳时回报就是0。回报又分为立即回报和长期回报，立即回报指的是执行当前动作后能立刻获得的奖励，但很多时候我们执行一个动作后并不能立即得到回报，而是在游戏结束时才能返回一个回报值，这就是长期回报。强化学习唯一的准则就是学习通过一序列的最优动作，获得最大的长期回
战性的是，任一状态下做出的动作不仅影响当前状态的立即回报，而且也会影响到下一个状态，因此
个执行过程的回报。

因此，强化学习和监督学习的区别主要有以下两点[6]：

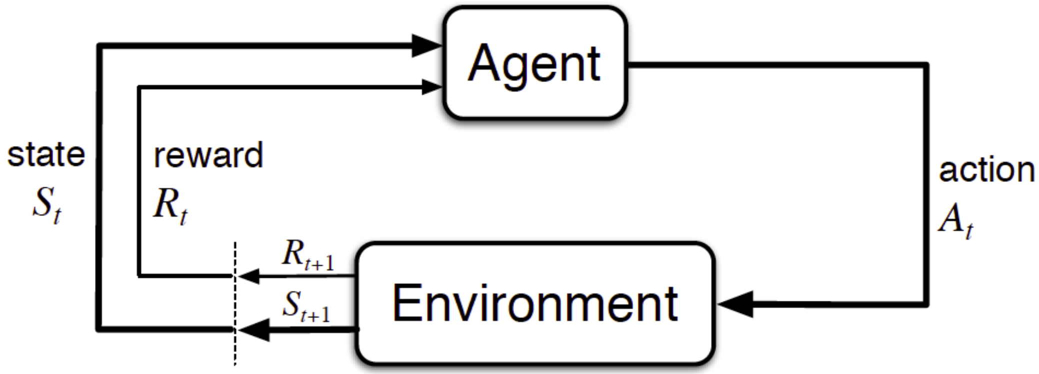
1. 强化学习是试错学习(Trail-and-error)，由于没有直接的指导信息，智能体要以不断与环境进行交错的方式来获得最佳策略。
2. 延迟回报，强化学习的指导信息很少，而且往往是在事后（最后一个状态）才给出的，这就导致了一个问题，就是获得正回报或者负回报以后，如何将回报分配给前面的状态。

问题描述与MDP

前面已经提到强化学习是尝试并发现回报最大动作的过程，下面就具体来描
一个之前完全没有接触过国际象棋的小白怎样和一个专业棋手对弈。刚开始
下，但假设双方每一轮下完后都会得到立即回报，比如吃子回报为1，被吃
始小白会输得很惨，但如果小白很聪明，随着不断地尝试小白不仅理解了下
什么动作可以吃更多的棋子。在这里我们将小白作为我们的智能体agent，棋
状态做出的动作，每个动作执行完后都会引起状态改变，如果状态的改变只

之前的状态和动作无关（即满足马尔可夫性），那么整个过程可以用马尔可夫决策过程（Markov Decision Processes）来描述，而Sutton在书中直接将满足马尔可夫性的强化学习任务定义为马尔可夫决策过程，并将状态和动作都是有限空间的MDP定义为有限马尔可夫决策过程（finite MDP）。

下面引入一些定义[1]：马尔可夫决策过程是一个agent与环境交互的过程，因此有一个离散的时间序列，$t=0,1,2,3,\ldots$，在每一个时刻t，agent都会接收一个用来表示环境的状态$S_t\in\mathbf{S}$，其中$\mathbf{S}$表示所有可能状态的集合，并且在状态的基础上选择一个动作$A_t\in\mathbf{A}(S_t)$，其中$\mathbf{A}(S_t)$表示在状态S_t时所有可能采取的动作的集合，在t时刻agent采取一个动作后都会收到一个回报值$R_{t+1}\in\mathbf{R}$，然后接收一个新状态S_{t+1}。下图为整个过程的示意图。



在任意时刻和状态下，agent都可以选择一个动作，选择的依据就是我们说的
而一个使得在任意时刻和状态下的长期回报都是最大的策略是我们最终需要
时刻的立即回报来表示：

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots = \sum_{k=t}^{\infty}$$

但实际上我们一般会用下面更通用的公式来代替：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T = \sum$$

其中$\gamma \in [0,1]$称为回报折扣因子，表明了未来的回报相对于当前回报的重要程度。$\gamma = 0$时，相当于只考虑立即回报不考虑长期回报，$\gamma = 1$时，将长期回报和立即回报看得同等重要。$T \in [1, \infty]$表示完成一次实验过程的总步数，$T = \infty$和$\gamma = 1$不能同时满足，否则长期回报将无法收敛。特别地，我们将一次有限步数的实验称作一个单独的episodes，也就是经过有限步数后最终会接收一个终止状态，这一类的任务也叫做episodic tasks。下面讨论的强化学习任务都是有限MDP的episodic tasks。

马尔可夫决策过程

一个有限马尔可夫决策过程由一个四元组构成 $M = (\mathbf{S}, \mathbf{A}, \mathbf{P}, \mathbf{R})$[6]。如上所述，$\mathbf{S}$表示状态集空间，空间，$\mathbf{P}$表示状态转移概率矩阵，$\mathbf{R}$表示期望回报值。

在MDP中给定任何一个状态$s \in \mathbf{S}$和动作$a \in \mathbf{A}$，都会以某个概率转移到下一个状态s'，这个概率为$p(s' \mid s, a) = \mathbf{Pr}\{S_{t+1} = s' \mid S_t = s, A_t = a\} \in \mathbf{P}$，并获得下一个回报的期望值为$r(s, a, s') = \mathbf{E}\left[R_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s'\right] \in \mathbf{R}$。

值函数及贝尔曼公式

增强学习的最终结果是找到一个环境到动作的映射—即策略$\pi(a \mid s)$。如果一
就会掉入眼前陷阱。比如说有一个岔路口，往左回报是100，往右回报是10
往左，但往左走的下一次回报只有10，而往右走的下一次回报有200，可以看
增强学习又往往有具有延迟回报的特点，在很多情况下的动作并不会产生立
的确会导致后续回报的产生，因此立即回报并不能说明策略的好坏。在几乎
来表示给定策略下期望的未来回报，并将值函数作为评估学习效果的指标。

值函数有多种定义，目前常见的是将值函数直接定义为未来回报的期望：

$$v_\pi(s) = \mathbf{E}_\pi\left[G_t \mid S_t = s\right] = \mathbf{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+\cdots\right.$$

上面表示的是在某个策略π下，当环境处于状态s时未来回报的期望，因此又叫做状态值函数(state-value function for policy)，只跟当前状态有关。同样，我们也可以定义动作值函数(action-value function for policy)，如下：

$$q_\pi(s,a) = \mathbf{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] = \mathbf{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \tag{2.2}$$

动作值函数表示在某个策略π下，当环境处于状态s时采取动作a的未来回报的期望。可以看到动作值函数与状态值函数唯一的不同是动作值函数不仅指定了一个初始状态，而且也指定了初始动作，而状态值函数的初始动作是根据策略产生的。由于在MDP中，给定状态s，agent根据策略选择动作a，下个时刻将以概率$p(s^{'} \mid s,a)$转移到状态$s^{'}$，因此值函数又可以改写成如下形式：

$$\begin{aligned}
v_\pi(s) &= \mathbf{E}_\pi\left[G_t \mid S_t = s\right] \\
&= \mathbf{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1} \mid S_t = s\right] \\
&= \mathbf{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2} \mid S_t = s\right] \\
&= \sum_a \pi(a \mid s)\cdot\mathbf{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2} \mid S_t = s, A_t\right] \\
&= \sum_a \pi(a \mid s)\sum_{s^{'}} p(s^{'} \mid s,a)\left[r(s,a,s^{'}) + \gamma\mathbf{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2} \mid S_{t+1} = s^{'}\right]\right] \\
&= \sum_a \pi(a \mid s)\sum_{s^{'}} p(s^{'} \mid s,a)\left[r(s,a,s^{'}) + \gamma v_\pi(s^{'})\right]
\end{aligned}$$

也就是说在策略π下当前状态的值函数可以通过下一个状态的值函数来迭代

（Bellman equation for v_π）。

同样，动作值函数也可以写成相似的形式：

$$
\begin{aligned}
q_\pi(s,a) &= \mathbf{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\
&= \mathbf{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s, A_t = a\right] \\
&= \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma v_\pi(s')\right]
\end{aligned}
\tag{2.4}
$$

$v_\pi(s)$也可以用$q_\pi(s,a)$来表示：

$$
v_\pi(s) = \sum_a \pi(a \mid s) q_\pi(s,a) \tag{2.5}
$$

下面是迭代计算$v_\pi(s)$和$q_\pi(s,a)$的图解[1]，可以与上述公式对照理解。

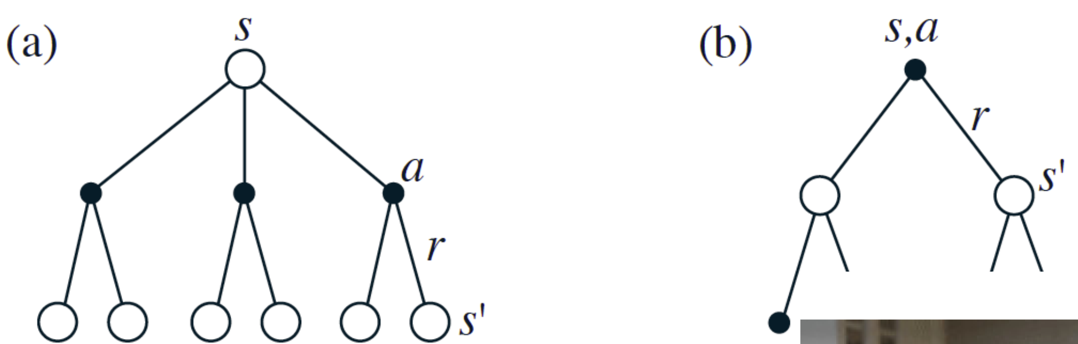


Figure 3.4: Backup diagrams for (a) v_π

最优值函数及贝尔曼最优公式

上面所说的值函数都是未来回报的期望值，而我们需要得到的最优策略必然
最大的，也就是说我们的优化目标可以表示为：

$$\pi_* = \arg\max_{\pi} v_\pi(s) \tag{2.6}$$

当然最优策略可能不止一个，但这些最优策略都有一个共同的特点，就是它们共享同样的状态值函数，这个状态值函数叫做最优状态值函数（optimal state-value function），用v_*来表示。对于所有的$s \in \mathbf{S}$，

$$v_*(s) = \max_{\pi} v_\pi(s) \tag{2.7}$$

最优策略同样也共享相同的动作值函数（optimal action-value function），用q_*来表示。对于所有的s
$a \in \mathbf{A}(s)$，

$$q_*(s,a) = \max_{\pi} q_\pi(s,a)$$

回顾一下上面动作值函数的改写公式(2.4)，$q_\pi(s,a) = \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma v_\pi(s')\right]$，由于动
示的是给定初始动作，后面的动作遵循策略π，因此最优动作值函数后面的动作应当遵循最优策略π_*，不难得到下面的公式。

$$q_*(s,a) = \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma v\right.$$

至此，最优值函数的形式已经给出了，现在我们继续回顾一下公式(2.5)的意
必然存在$v_\pi(s) \le \max q_\pi(s,a)$。但对于最优策略来说，

$$\begin{aligned} v_*(s) &= \max_{\mathbf{a}} q_*(s,a) \\ &= \max_{\mathbf{a}} \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma\right. \end{aligned}$$

$$q_*(s,a) = \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma \max_{a'} q_*(s',a')\right] \tag{2.11}$$

与状态值函数的贝尔曼公式一样，最优状态值函数和最优动作值函数也可以表示成递归的形式，因此公式(2.10)和公式(2.11)又分别叫做状态值函数和动作值函数的贝尔曼最优公式（Bellman optimality equation）。因为没有$\pi(a \mid s)$，不需要根据策略生成动作，因此贝尔曼最优公式完全独立于策略，但如果我们已知v_*或q_*，都可以很容易地得到最优策略。

如果我们已知v_*，而且在每一步都有多个动作可以选择，可以想到最优策略的$v_*(s)$必然是满足贝尔
的，因此至少有一个动作会满足公式中的最大化条件。任何一个采用上述动作并能够以非零概率转移
的策略都是最优策略。我们可以把当前动作的选择看成是一个单步搜索（one-step search）的问题，在
单步搜索结果最大的动作即最优动作，而每个状态下都采取最优动作的策略即最优策略。如果我们已
需要在每一步都选择使得$q_*(s,a)$最大的动作，就可以得到一个最优策略。

贝尔曼公式与贝尔曼最优公式是MDP求解的基础，下面主要介绍几种MDP求解的方法。

动态规划方法

动态规划（dynamic programming）指的是能够用来解决给定环境模型，计算
划算法存在两个问题，一是需要依赖一个非常好的环境状态转移模型，二是
几乎不会直接用动态规划求解MDP，但动态规划理论还是非常重要的，因为
础上，摆脱模型依赖并尽可能地减少计算量。

策略估计

首先，我们考虑一下如果已知策略π，如何来计算v_π。这个问题被称作DP迭

先举一个例子，一个岔路口有向左和向右两个方向，向左回报为10，向右回报为100，我们没有任何先验知识，但我们需要估计站在路口的值函数，也就是估计当前状态的值函数，该如何来估计呢？首先我们将值函数初始化为0，然后进行大量的尝试，每次都以0.5的概率选择方向左，并获得回报10，以0.5的概率选择方向右，获得回报100。那么只要能将这两个方向都至少遍历一遍，就可以得到该状态的值函数$v_{随机策略} = \frac{1}{N}\sum_{i=0}^{N} 0.5 \cdot R_i$，其中$N$为实验的总次数。

同样，我们也是采用相似的方法迭代来进行策略估计的。首先将所有的$v_\pi(s)$都初始化为0（或者任意值，但终止状态必须为0），然后采用如下公式更新所有状态s的值函数。

$$
\begin{aligned}
v_{k+1}(s) &= \mathbf{E}_\pi\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s\right] \\
&= \sum_a \pi(a \mid s) \sum_{s'} p(s' \mid s, a)\left[r(s, a, s') + \gamma v_k(s')\right]
\end{aligned}
$$

其中$v_{k+1}(s)$表示在当前策略下第$k+1$次迭代状态s的值函数，$v_k(s')$表示在当前策略下第k次迭代状态s的值函数，该公式就是用上一次迭代计算得到的值函数来更新本次迭代的值函数。在具体操作时，又有两种

- 将第k次迭代计算得到的所有状态值函数$[v_k(s_1), v_k(s_2), v_k(s_3), \ldots]$保存在一个数组中，第$k+1$次迭代的$v_{k+1}(s)$使用第$k$次的$v_k(s')$进行更新，更新后的值保存在另一个数组中。
- 仅用一个数组来保存各状态的值函数，每次更新后就将原来的值覆盖。可能使用的是第$k+1$次更新后的$v_{k+1}(s')$，这样可以及时地利用更新

下面为整个策略估计的算法过程：

Input π, the policy to be evaluated
Initialize an array $v(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $temp \leftarrow v(s)$
    $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
    $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $v \approx v_\pi$

策略改进

策略估计是为了计算当前策略下各状态的值函数，那得到值函数又有什么用呢？首先我们可以用来比

好坏，如果状态值函数是已知的，那么就可以根据公式(2.4)计算动作值函数，如果一个策略π的所有动

大于另一个策略π'，那么可以认为策略π比策略π'更好。其次，最主要的用处是可以用来进行策略改进（policy improvement）。

仍然是上面岔路口的例子，但是假设无论向左还是向右，下一个路口都是唯识，因此采用了一个随机策略，然后我们可以计算得到随机策略下的状态值了。具体的做法就是前面提到的单步搜索，向左时当前动作的回报为10，因个路口的值函数，而向右为100+γv，因此策略会更新为向右，而不再是随机意到，单步搜索计算的值正是动作值函数。

根据上面的例子，我们可以总结一下策略改进的方法：遍历所有的状态和所的更新，即对所有$s \in \mathbf{S}$，

$$
\begin{aligned}
\pi'(s) &= \arg\max_{\mathbf{a}} q_\pi(s,a) \\
&= \arg\max_{\mathbf{a}} \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma v_\pi(s')\right]
\end{aligned}
\tag{3.2}
$$

现在我们已经知道如何计算当前策略的状态值函数，也知道可以根据动作值函数来更新策略，那下面就来讲讲如何从零开始求解最优策略。

策略迭代

一旦策略π通过策略改进得到一个更好的策略π'，那么我们就可以通过策略估计算法，计算策略π'的
并用公式(3.2)进行策略改进得到一个比策略π'更好的策略π''。如下图所示，经过无数次的策略估计禾
后，我们终将会收敛于最优策略π_*。这种通过不断迭代地去改进策略的方法叫做策略迭代（policy ite

$$
\pi_0 \xrightarrow{\mathrm{E}} v_{\pi_0} \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} v_{\pi_1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} \cdots \xrightarrow{\mathrm{I}} \pi_* \xrightarrow{\mathrm{E}} v_*,
$$

下面为整个策略迭代的算法过程：

1. Initialization
   $v(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\quad \Delta \leftarrow 0$
   $\quad$ For each $s \in \mathcal{S}$:
   $\quad\quad temp \leftarrow v(s)$
   $\quad\quad v(s) \leftarrow \sum_{s'} p(s'|s,\pi(s))\Big[r(s,\pi(s),s') + \gamma v(s')\Big]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad temp \leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s,a)\Big[r(s,a,$
   $\quad$ If $temp \neq \pi(s)$, then *policy-stable* $\leftarrow$
   If *policy-stable*, then stop and return v a

值迭代

策略迭代算法需要不断地进行策略估计和策略改进，每次策略估计和改进都
算法的计算量非常大，效率非常低。同时可以看到策略迭代的依据是贝尔曼

会不会加速求解过程呢？事实上是可以的，下面的值迭代（value iteration）算法就是利用贝尔曼最优公式来提高求解效率的一种算法。

我们还是需要先迭代估计状态值函数，但不必每次迭代都进行策略改进。根据贝尔曼最优公式，可以直接用上一次迭代的最大动作值函数对当前迭代的状态值函数进行更新，如下所示：

$$\begin{aligned} v_{k+1}(s) &= \max_{\mathbf{a}} q_k(s,a) \\ &= \max_{\mathbf{a}} \sum_{s'} p(s' \mid s,a)\left[r(s,a,s') + \gamma v_k(s')\right] \end{aligned} \tag{3.3}$$

值迭代算法的好处就是省去了每次迭代时的策略改进过程，并且由于每次迭代得到的$v_{k+1}(s)$都要≥
的$v_{k+1}(s)$，也就是说相同迭代次数下，策略迭代得到的策略肯定没有值迭代得到的策略好，因此能
收敛。直到值函数收敛到最优值函数后，再通过最优值函数来计算得到最优策略，下面是值迭代算

Initialize array v arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathcal{S}$:
$temp \leftarrow v(s)$
$v(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s')$
$\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such tha
$\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\Big[r(s,a,s')$

一般来说值迭代和策略迭代都需要经过无数次迭代才能精确收敛到最优策略

Δ来作为迭代中止条件，即当所有的$v_\pi(s)$变化量小于Δ时，我们就近似的认为获得了最优策略。值迭代和策略迭代都可以用来求解最优策略，但是都需要依赖一个现有的环境模型，而对环境进行精确建模往往是非常困难的，所以导致了动态规划方法在MDP求解时几乎不可用，当然如果状态转移是确定性的（$p(s' \mid s,a)=1$），那就另当别论了。

蒙特卡罗方法

下面我们要讲的是蒙特卡罗方法（Monte Carlo Methods）。与动态规划不同，蒙特卡罗方法不需要知道环境的完整
模型，仅仅需要经验就可以获得最优策略，这些经验可以通过与环境在线或模拟交互的方式获得。在
不需要任何环境的先验知识，模拟交互虽然需要知道环境状态的转移，但与动态规划不同的是这里不
的转移概率。

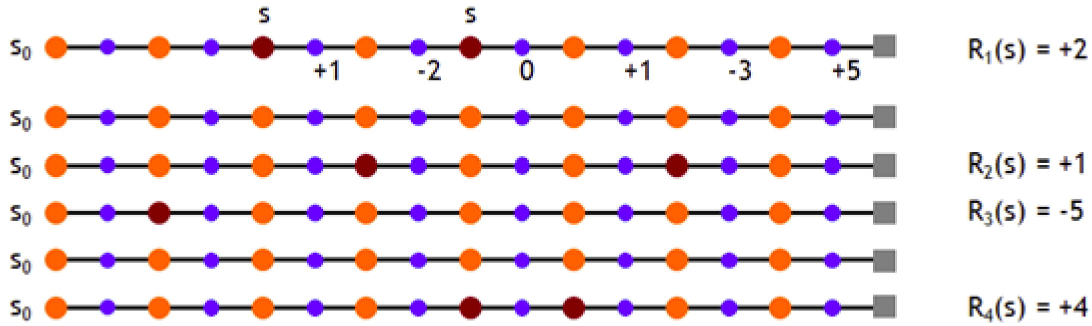
蒙特卡罗方法也称统计模拟方法，基本思想是通过对大量的重复随机事件进行统计，估计随机事件的概率分布或期
望。一个典型的例子是利用蒙特卡罗方法计算圆周率。假设我们知道圆的面积公式为$S=\pi r^2$，那计算
式自然就是$\pi=\frac{S}{r^2}$，因此如果我们知道圆面积和圆半径，那么就可以求到圆周率。那么如何计算一个圆的面积
呢？给定一个圆，我们可以画出这个圆的外切正方形，那么这个外切正方形的面积为$S_{正方形}=4r^2$，现在我们往
正方形区域随机投点，并统计点落在圆内的概率p，那么圆面积可以这么计算：$S_{圆}=p \cdot S_{正方形}$，因此
$\pi=4 \cdot p$。可以想到，如果投点次数越多，p估计越精确，π的结果也就越接

蒙特卡罗策略估计

我们现在来考虑一下如何利用蒙特卡罗方法估计给定策略下的状态值函数。
是，现在我们估计的是未来回报的期望，而不是概率，但基本思想是一样的
先需要根据给定策略生成大量的经验数据，然后从中统计从状态s开始的未来
估计的状态值函数。这种利用蒙特卡罗方法进行策略估计的算法又叫做蒙特
Evaluation）。

蒙特卡罗策略估计在具体实现时又分为first-visit MC methods和every-visit MC methods。由于在一个episode中，状态s可能会出现多次，first-visit MC methods就是只统计第一次到达该状态的未来回报，而every-visit MC methods是所有达到该状态的未来回报都会统计累加起来。下面我们举例说明first-visit MC methods的估计方法[6]。

现在我们假设有如下一些样本（下图每一行都是在当前策略下的一个独立的episode），紫色实心点为状态s，取折扣因子γ=1，即直接计算累积回报。



第一个episode中到达过两次状态s，我们只计算第一次的未来回报$R_1(s)=1-2+0+1-3+5=2$。假设我们已经用相同的方法计算得到$R_2(s)=1$，$R_3(s)=-5$，$R_4(s)=4$。那么当前策略下状态s的值函数

$$v_\pi(s)=\mathbf{E}\left[R(s)\right]=\frac{1}{N}\sum_{i=1}^{N}\left[R_i(s)\right]=\frac{1}{4}\ (2+1-$$

同样，如果生成的episode数量越多，$v_\pi(s)$的估计就越接近真实值，下面是

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

(a) Generate an episode using π

(b) For each state s appearing in the episode:

$G \leftarrow$ return following the first occurrence of s

Append G to $Returns(s)$

$V(s) \leftarrow \text{average}(Returns(s))$

注意这里使用大写的V表示状态值函数的估计，Sutton的理由是状态值函数一旦初始化，就会立即变成值了，因为G会随着生成的episode不同而不断变化。可以认为每次G都为$v_\pi(s)$的一个独立同分布估计 当数据量非常大时$V(s)$将最终收敛于这个分布的均值。

动作值函数的蒙特卡罗估计

由于我们没有完整的环境状态转移模型，因此即使我们得到当前策略的值函
进。既然我们可以估计得到状态值函数，那么肯定也可以用相同的方法直接
数的蒙特卡罗估计（Monte Carlo Estimation of Action Values）。

估计方法跟蒙特卡罗策略估计差不多，只不过我们需要找到所有的状态动作
计每一个状态动作对的未来回报的平均值，即$q_\pi(s,a)$的估计值。得到了$q_\pi($
略改进了。

蒙特卡罗控制

蒙特卡罗控制（Monte Carlo Control）首要的问题就是如何估计最优策略。跟之前动态规划一样，这里也可以采用策略迭代和策略改进交替进行的方式，经过大量的迭代后收敛到最优策略。但蒙特卡罗方法有一个最大的问题，即我们需要产生无数的episode才能保证收敛到最优结果。无数的episode和大量的迭代导致计算量巨大，效率非常低。Sutton在书[1]中提到两种解决方法，其中一种方法是采用episode-by-episode的方式进行优化。

episode-by-episode的思想与动态规划中值迭代的in-place版本非常相似。在动态规划的值迭代中，我们每次迭代都直接覆盖更新值函数，因此能及时地利用到更新后的值函数，从而能加快收敛。episode-by-episode则是生成一个episode，然后根据这个episode进行动作值函数的更新，同时更新策略，并利用更新后的策略
续的episode。

下面是exploring starts的蒙特卡罗控制（Monte Carlo ES，exploring starts指的是从一个随机的开始状态
个episode）算法的完整过程：

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$\quad Q(s,a) \leftarrow$ arbitrary
$\quad \pi(s) \leftarrow$ arbitrary
$\quad Returns(s,a) \leftarrow$ empty list

Repeat forever:
$\quad$(a) Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs h
$\qquad$Generate an episode starting from S_0, A_0, foll
$\quad$(b) For each pair s, a appearing in the episode:
$\qquad G \leftarrow$ return following the first occurrence o
$\qquad$Append G to $Returns(s,a)$
$\qquad Q(s,a) \leftarrow \text{average}(Returns(s,a))$
$\quad$(c) For each s in the episode:
$\qquad \pi(s) \leftarrow \arg\max_a Q(s,a)$

至于为何要使用exploring starts，这与episode-by-episode在线生成episode的更新策略有关。还是上面的岔路口的例子，我们先随机指定一个策略，比如指定向左，那么使用该策略生成一个episode时必然也是向左，那么也就只能更新向左的动作值函数了，而无法更新向右的动作值函数。由于动作值函数是随机初始化的，如果向右的动作值函数初始值小于更新后的向左的动作值函数，那么下一次生成episode时仍然是向左，并且可以想象可能永远不会选择向右。但其实向右才是最优动作，因此上述更新的策略永远不可能是最优策略。但随机选择开始状态和动作，可以避免某些动作的值函数不会更新的问题，因此可以保证能获得最优策略。

当然也可以采用其他方法避免使用exploring starts，下面要介绍的on-policy方法和off-policy方法就是其中的两种方法。

On-Policy蒙特卡罗控制

前面的Monte Carlo ES算法使用exploring starts是为了保证所有可能的动作值函数都能得到更新，从而保证能获得最优策略。如果策略本身就可以在任何状态下都采取所有可能的动作，而不是贪婪地只选择动作值函数
那问题不就迎刃而解了吗。下面要讨论策略是非确定性的，也就是对于所有的状态s和该状态下所有可
都有$\pi(a \mid s) > 0$，并且用$\epsilon - soft$策略生成episode。由于我们评估和改进的策略与生成episode的策略是相同的，因此叫做on-policy方法。

在$\epsilon - soft$策略中，大多数时候策略会选择动作值函数最大的动作（或者换
数最大的动作，ϵ是一个非常小的正数），但也会以概率ϵ从其他动作中随机

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$Q(s,a) \leftarrow$ arbitrary

$Returns(s,a) \leftarrow$ empty list

$\pi \leftarrow$ an arbitrary ε-soft policy

Repeat forever:

(a) Generate an episode using π

(b) For each pair s, a appearing in the episode:

$G \leftarrow$ return following the first occurrence of s, a

Append G to $Returns(s,a)$

$Q(s,a) \leftarrow \text{average}(Returns(s,a))$

(c) For each s in the episode:

$a^* \leftarrow \arg\max_a Q(s,a)$

For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

Off-Policy蒙特卡罗控制

在off-policy方法中，生成episode的策略与评估和改进的策略并非同一个策略
策略（behavior policy），而评估和改进的策略叫估计策略（estimation policy
略是$\epsilon - soft$策略，但估计策略是确定性的。下面只给出算法流程，具体推

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$N(s,a) \leftarrow 0$; Numerator and
$D(s,a) \leftarrow 0$; Denominator of $Q(s,a)$
$\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
(a) Select a policy μ and use it to generate an episode:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
(b) $\tau \leftarrow$ latest time at which $A_\tau \neq \pi(S_\tau)$
(c) For each pair s, a appearing in the episode at time τ or later:
$t \leftarrow$ the time of first occurrence of s, a such that $t \geq \tau$
$W \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\mu(A_k|S_k)}$
$N(s,a) \leftarrow N(s,a) + WG_t$
$D(s,a) \leftarrow D(s,a) + W$
$Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
(d) For each $s \in \mathcal{S}$:
$\pi(s) \leftarrow \arg\max_a Q(s,a)$

时间差分学习

时间差分学习（temporal-dierence (TD) learning）结合了动态规划和蒙特卡罗
需要环境模型，与动态规划一样更新估计值时只依赖于下一个状态可用的估
episode。

TD预测

TD预测（TD prediction）又叫TD策略估计，就是从给定的一系列经验数据中估计出当前策略的状态值函数v_π。回顾一下蒙特卡罗控制，我们是先自举一个episode，然后根据历史episode和当前最新的episode计算从状态s开始未来回报的均值，作为当前状态值函数的更新值。对上面更新方式稍做修改，我们可以用一种滑动平均的方法来更新，即只用当前episode的未来回报与状态值函数的差值来更新。一个简单的every-visit MC方法的更新公式就如下所示：

$$V(S_t) = (1-\alpha)V(S_t) + \alpha G_t = V(S_t) + \alpha\left[G_t - V(S_t)\right] \qquad (4\text{-}1)$$

$V(S_t)$表示第t个时刻为状态S_t的状态值函数，G_t表示从状态S_t开始到episode结束时的总回报，α是一个常数步长参数（梯度下降算法中叫学习率），这个公式叫做$constant-\alpha$ MC。在这个公式中，G_t是需要等到
结束才能得到的，因此只有在自举完整的episode后才能进行更新。下面要说的TD算法就很好地解决
只需要等到下一个时刻转移到下一个状态和获得回报值。下面是一种最简单的TD算法，叫做TD(0)。

$$V(S_t) = V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

我们这里只是用$R_{t+1} + \gamma V(S_{t+1})$来估计$constant-\alpha$ MC中未来回报的真实值。与蒙特卡罗控制一
能确保收敛到最优状态值函数，当然前提也是需要大量的经验数据。至于TD(0)与蒙特卡罗控制哪个算法收敛更快，这个问题并没有准确的答案，不过Sutton在书中指出，在一些随机任务上TD(0)比$constant-\alpha$ M
快。TD(0)算法在每个时刻都要进行一次更新，更高效的方法是在训练时使用batch updating的方式，即一个batch进行一次更新。

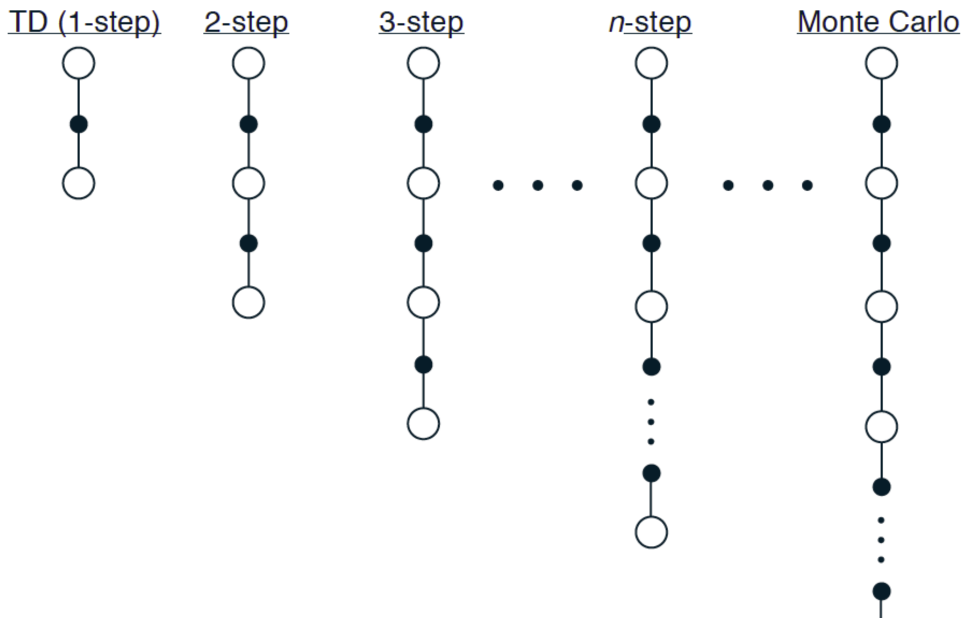
显然，TD learning相比MC有以下优点[7]：

- 由于TD预测使用差值进行更新，加上步进参数α的存在，TD learning
- TD learning可以用于在线训练，因为不需要等到整个episode结束才更
- TD learning应用更广，可以用于非有限步数的情况。

但也存在一些缺点，比如TD learning对初始值比较敏感，以及收敛结果是有

TD(λ)

在介绍TD(λ)之前，我们先介绍一下n-Step TD预测。前面介绍的TD(0)算法在当前状态的基础上往后执行一步就可以进行更新，并且在更新时使用了贝尔曼公式对当前状态的未来回报进行估计，那我们是不是也可以往后执行n步之后再更新，这样用贝尔曼公式估计的未来回报是不是会更加精确呢？实际上，当n等于整个episode的总步数时，n-Step TD预测就完全成了MC估计了。



对于1-step来说，未来回报的值等于第一个回报值加上下一个状态值函数折扌

$$G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$$

2-step比1-step多执行一步，其未来回报值为：

$$G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$$

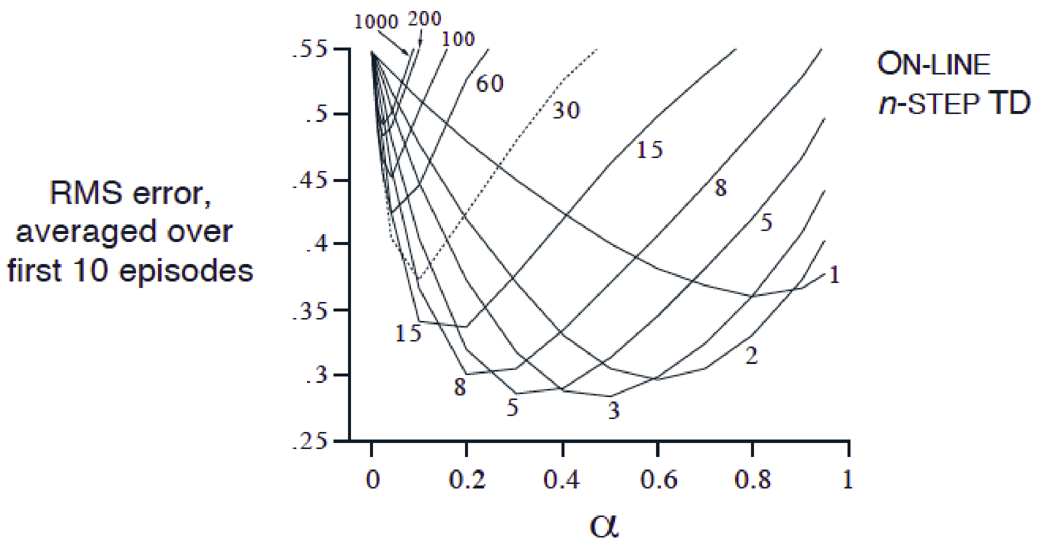
那么n-step的未来回报值为：

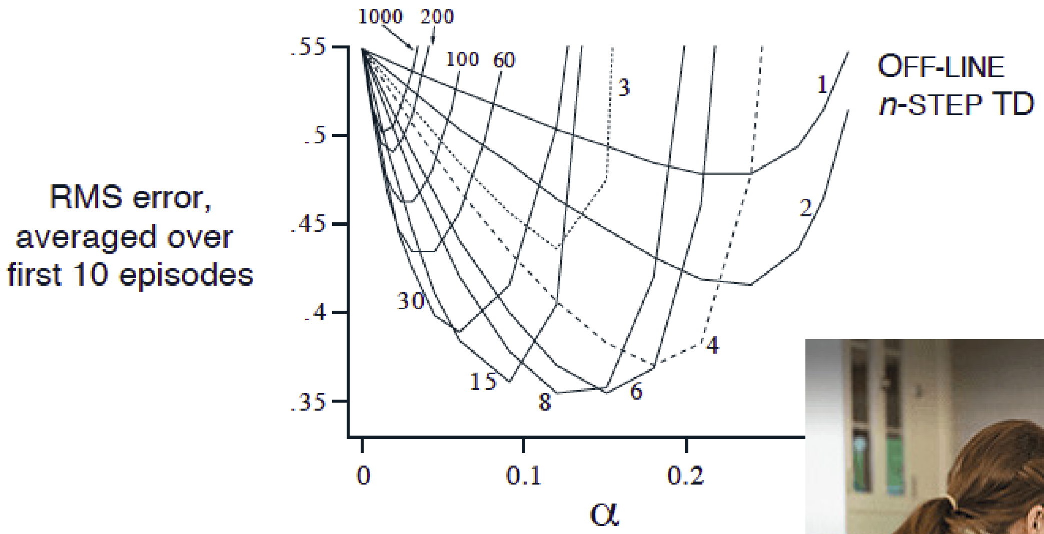
$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) + \ldots + \gamma^n V(S_{t+n})$$

在公式(4-1)中我们用$G_t^{(n)}$替代G_t，最后n-Step TD预测的更新公式为：

$$V(S_t) = V(S_t) + \alpha \left[G_t^{(n)} - V(S_t) \right] \tag{4-3}$$

n-Step TD预测一定程度上可以使得估计的值函数更准确，因此收敛效果会更好，但更新时需要等待的步数增加了。下图是使用n-Step TD方法在random walk任务上的RMS error对比。





n-Step TD只使用了从当前状态开始执行n步未来回报的估计值$G_t^{(n)}$，其实为
以使用不同的n对应的$G_t^{(n)}$的平均值。比如可以把2-step和4-step的均值作为G

$$G_t^{avg} = \frac{1}{2} G_t^{(2)} + \frac{1}{2} G_t^{(4)}$$

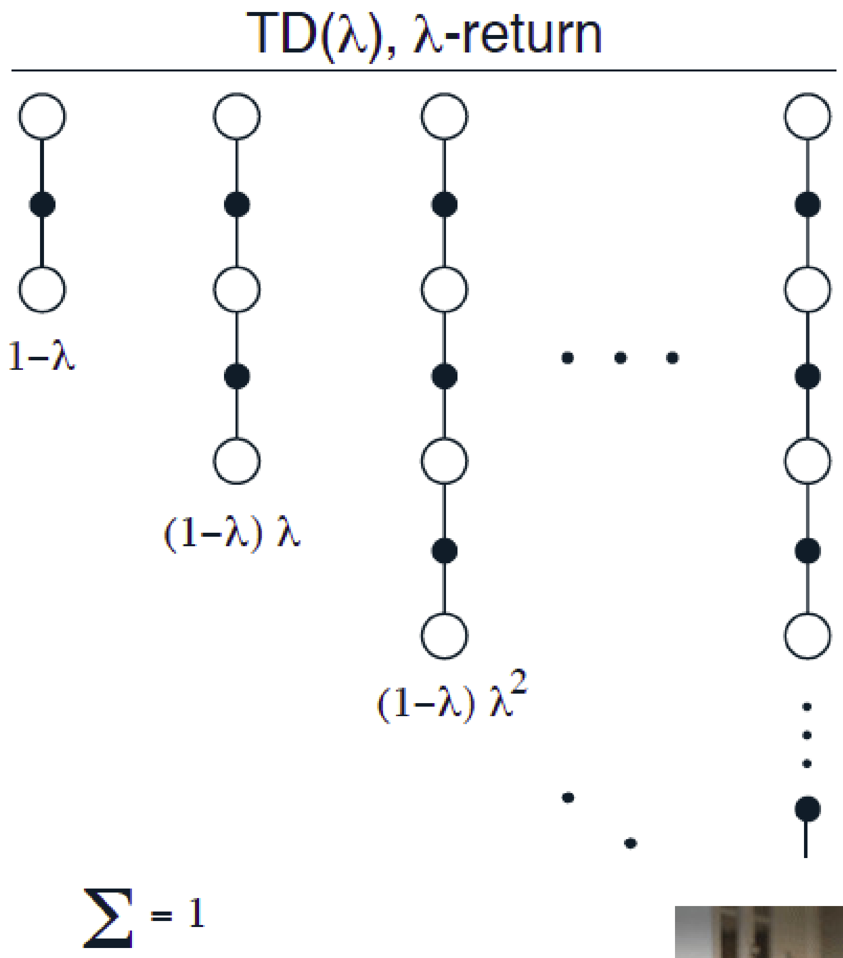
TD(λ)也可以理解为一种特殊的n-step平均算法，每个n-step的权重为$(1-\lambda)\lambda^{(n-1)}$，所有权重和仍然为1，因此有：

$$G_t^{(\lambda)} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)} \tag{4-4}$$

公式(4-4)表示的是没有终止状态的情况，对于最终存在终止状态的episode任务或截断任务[注1]来讲，为了保证所有权重的和为1，最后一个n-step的权重被设置为λ^{T-t-1}，其中T为episode总步数。

$$G_t^{(\lambda)} = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_t^{(n)} + \lambda^{T-t-1}G_t$$

当$\lambda=1$时，这时TD(λ)就相当于MC，而当$\lambda=0$时，TD(λ)就退化成了TD(0)。



Sarsa

接下来我们考虑一下如何使用TD预测进行策略改进。首先我们知道可以使用
使用公式(3-2)进行策略改进。但问题来了，公式(3-2)中的$p(s' \mid s, a)$是未知
下蒙特卡洛控制方法，TD也可以直接对动作值函数q_π进行估计。与v_π的更新

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}$$

有了状态值函数，接下来就可以使用公式(3-2)进行策略改进了。在公式(4-3)中，每次非结束状态S_t转移到下一个状态时都进行一次值函数的更新，每次更新都只与$(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$有关，因此叫做Sarsa算法。如果状态$S_{t+1}$为终止状态，则$Q(S_{t+1}, A_{t+1}) = 0$。下面是Sarsa $\epsilon-greedy$算法的完整过程，由于评估和改进时采用的策略与生成episode的策略是同一个策略，因此Sarsa算法是一种on-policy方法。

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Sarsa的Q值更新公式与$TD(0)$一致，实际上也可以采用$TD(\lambda)$的形式进行Q值更新，这个改进算法就:
关于Sarsa(λ)的具体介绍请参考《Reinforcement Learning: An Introduction》一书第七章。


Q-Learning

下面介绍的Q学习是一种off-policy方法，并被认为是强化学习算法最重要的
数的更新完全独立于生成episode的策略，使得学习到的$Q(S_t, A_t)$直接是最优

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},$$

公式(4-4)为Q-learning的单步更新公式，与Sarsa唯一的不同是：类似于动态
使用最优的$Q(S_{t+1}, A_{t+1})$进行更新，也就相当于策略只采用了最大Q值对应
收敛性证明的难度，使得它的收敛性很早就得到了证明。但与前面介绍的蒙

最大的动作，因此这个算法也会导致部分state-action对不会被策略生成，相应的动作值函数也无法得到更新。为了确保能收敛到最优策略，下面的算法在生成episode时同样使用了$\epsilon - greedy$策略，但更新时仍然采用确定性策略（即策略只选择Q值最大的动作）。

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$;
  until S is terminal

DQN

DQN改进算法

强化学习在内容推荐中的应用

参考资料

1、Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. B
2、Playing Atari with Deep Reinforcement Learning，DeepMind Technologies，
3、Human-level control through deep reinforcement learning，DeepMind Techno
4、DeepMind官网 https://deepmind.com/blog/deep-reinforcement-learning

5、https://www.nervanasys.com/demystifying-deep-reinforcement-learning

6、http://www.cnblogs.com/jinxulin/p/3511298.html

7、Introduction to Reinforcement Learning，David Silver

注释

1、截断任务：在强化学习中，非episode任务由于不存在终止状态，为了便于训练可以将非episode任务截断成episode。

上一篇 tensorflow源码安装

相关文章推荐

- 深度强化学习初探
- Q-学习：强化学习
- 强化学习（Reinforcement Learning）知识整理
- 强化学习介绍（Introduction to RL）
- 强化学习介绍
- 浅谈增强学习
- 【深度学习介绍
- 强化学习
- 强化学习（Reinf
- 从特征描述符到

猜你在找

- 【直播】机器学习&深度学习系统实战（唐宇迪）
- 【直播】Kaggle 神器：XGBoost 从基础到实战（冒教授）
- 【直播回放】深度学习基础与TensorFlow实践（王琛）
- 【直播】计算机视觉原理及实战（屈教授）
- 【直播】机器学习之凸优化（马博士）
- 【直播】机器学习之矩阵（黄博士）
- 【直播】机器学习之概率与统计推断（冒教授）
- 【直播】机器学习之数学基础
- 【直播】TensorFlow实战进阶（智亮）
- 【直播】深度学习30天系统实训（唐宇迪）

查看评论

暂无评论

发表评论

用 户 名： haijunz

评论内容：

提交