https://www.cnblogs.com/jinxulin/p/3511298.html

（一）基本概念

机器学习算法大致可以分为三种：

1. 监督学习(如回归，分类)

2. 非监督学习(如聚类，降维)

3. 增强学习

什么是增强学习呢？

增强学习（reinforcementlearning, RL）又叫做强化学习，是近年来机器学习和智能控制领域的主要方法之一。

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

也就是说增强学习关注的是智能体如何在环境中采取一系列行为，从而获得最大的累积回报。

通过增强学习，一个智能体应该知道在什么状态下应该采取什么行为。RL 是从环境状态到动作的映射的学习，我们把这个映射称为策略。

那么增强学习具体解决哪些问题呢，我们来举一些例子：

例 1. flappy bird 是现在很流行的一款小游戏，不了解的同学可以点链接进去玩一会儿。现在我们让小鸟自行进行游戏，但是我们却没有小鸟的动力学模型，也不打算了解它的动力学。要怎么做呢？ 这时就可以给它设计一个增强学习算法，然后让小鸟不断的进行游戏，如果小鸟撞到柱子了，那就获得-1 的回报，否则获得 0 回报。通过这样的若干次训练，我们最终可以得到一只飞行技能高超的小鸟，它知道在什么情况下采取什么动作来躲避柱子。

例 2. 假设我们要构建一个下国际象棋的机器，这种情况不能使用监督学习，首先，我们本身不是优秀的棋手，而请象棋老师来遍历每个状态下的最佳棋步则代价过于昂贵。其次，每个棋步好坏判断不是孤立的，要依赖于对手的选择和局势的变化。是一系列的棋步组成的策略决定了是否能赢得比赛。下棋过程的唯一的反馈是在最后赢得或是输掉棋局时才产生的。这种情况我们可以采用增强学习算法，通过不断的探索和试错学习，增强学习可以获得某种下棋的策略，并在每个状态下都选择最有可能获胜的棋步。目前这种算法已经在棋类游戏中得到了广泛应用。

可以看到，增强学习和监督学习的区别主要有以下两点：

1. 增强学习是试错学习(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互，通过试错的方式来获得最佳策略。

2. 延迟回报，增强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。

增强学习是机器学习中一个非常活跃且有趣的领域，相比其他学习方法，增强学习更接近生物学习的本质，因此有望获得更高的智能，这一点在棋类游戏中已经得到体现。Tesauro(1995)描述的 TD-Gammon 程序，使用增强学习成为了世界级的西洋双陆棋选手。这个程序经过 150 万个自生成的对弈训练后，已经近似达到了人类最佳选手的水平，并在和人类顶级高手的较量中取得 40 盘仅输 1 盘的好成绩。

下篇我们正式开始学习增强学习，首先介绍一下马尔可夫决策过程。

参考资料：

[1] R.Sutton et al. Reinforcement learning: An introduction , 1998

[2] T.Mitchell. 《机器学习》，2003

[3] Andrew Ng.CS229: Machine learning Lecture notes

（二）马尔科夫决策过程（MDP）

1. 马尔可夫模型的几类子模型

大家应该还记得马尔科夫链(Markov Chain)，了解机器学习的也都知道隐马尔可夫模型(Hidden Markov Model，HMM)。它们具有的一个共同性质就是马尔可夫性(无后效性)，也就是指系统的下个状态只与当前状态信息有关，而与更早之前的状态无关。

马尔可夫决策过程(Markov Decision Process, MDP)也具有马尔可夫性，与上面不同的是 MDP 考虑了动作，即系统下个状态不仅和当前的状态有关，也和当前采取的动作有关。还是举下棋的例子，当我们在某个局面（状态 s）走了一步(动作 a)，这时对手的选择（导致下个状态 s’）我们是不能确定的，但是他的选择只和 s 和 a 有关，而不用考虑更早之前的状态和动作，即 s’ 是根据 s 和 a 随机生成的。

我们用一个二维表格表示一下，各种马尔可夫子模型的关系就很清楚了：

	不考虑动作	考虑动作
状态完全可见	马尔科夫链(MC)	马尔可夫决策过程(MDP)
状态不完全可见	隐马尔可夫模型(HMM)	不完全可观察马尔可夫决策过程(POMDP)

2. 马尔可夫决策过程

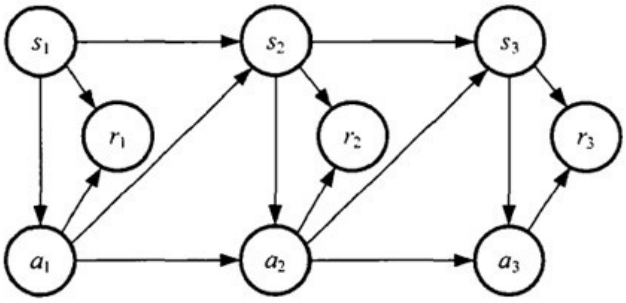
一个马尔可夫决策过程由一个四元组构成 M = (S, A, P_{sa}, R) [注 1]

- S: 表示状态集(states)，有 s∈S，s_i 表示第 i 步的状态。
- A:表示一组动作(actions)，有 a∈A，a_i 表示第 i 步的动作。
- P_{sa}: 表示状态转移概率。P_{sa} 表示的是在当前 s ∈ S 状态下，经过 a ∈ A 作用后，会转移到的其他状态的概率分布情况。比如，在状态 s 下执行动作 a，转移到 s'的概率可以表示为 p(s'|s,a)。
- R: S×A⟼ℝ ，R 是回报函数(reward function)。有些回报函数状态 S 的函数，可以简化为 R: S⟼ℝ。如果一组(s,a)转移到了下个状态 s'，那么回报函数可记为 r(s'|s, a)。如果(s,a)对应的下个状态 s'是唯一的，那么回报函数也可以记为 r(s,a)。

MDP 的动态过程如下：某个智能体(agent)的初始状态为 s_0，然后从 A 中挑选一个动作 a_0 执行，执行后，agent 按 P_{sa} 概率随机转移到了下一个 s_1 状态，$s_1 \in P_{s0a0}$。然后再执行一个动作 a_1，就转移到了 s_2，接下来再执行 a_2…，我们可以用下面的图表示状态转移的过程。

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

如果回报 r 是根据状态 s 和动作 a 得到的，则 MDP 还可以表示成下图：



3. 值函数(value function)

上篇我们提到增强学习学到的是一个从环境状态到动作的映射（即行为策略），记为策略 π: S→A。而增强学习往往又具有延迟回报的特点: 如果在第 n 步输掉了棋，那么只有状态 s_n 和动作 a_n 获得了立即回报 $r(s_n,a_n)=-1$，前面的所有状态立即回报均为 0。所以对于之前的任意状态 s 和动作 a，立即回报函数 r(s,a)无法说明策略的好坏。因而需要定义值函数(value function，又叫效用函数)来表明当前状态下策略 π 的长期影响。

用 $V^{\pi}(s)$表示策略 π 下，状态 s 的值函数。r_i 表示未来第 i 步的立即回报，常见的值函数有以下三种：

a) $$V^{\pi}(s)=E_{\pi}\left[\sum_{i=0}^{h} r_i \middle| s_0=s\right]$$

b) $$V^{\pi}(s)=\lim_{h\to\infty} E_{\pi}\left[\frac{1}{h}\sum_{i=0}^{h} r_i \middle| s_0=s\right]$$

c) $$V^{\pi}(s)=E_{\pi}\left[\sum_{i=0}^{\infty} \gamma^i r_i \middle| s_0=s\right]$$

其中：

a)是采用策略 π 的情况下未来有限 h 步的期望立即回报总和；

b)是采用策略 π 的情况下期望的平均回报；

c)是值函数最常见的形式，式中 γ∈[0,1]称为折合因子，表明了未来的回报相对于当前回报的重要程度。特别的，γ=0 时，相当于只考虑立即不考虑长期回报，γ=1 时，将长期回报和立即回报看得同等重要。接下来我们只讨论第三种形式，

现在将值函数的第三种形式展开，其中 r_i 表示未来第 i 步回报，s'表示下一步状态，则有：

$$V^{\pi}(s) = E_{\pi}\left[r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \ldots \middle| s_0 = s \right]$$
$$= E_{\pi}\left[r_0 + \gamma E\left[\gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \ldots \right] \middle| s_0 = s \right]$$
$$= E_{\pi}\left[r(s' \mid s, a) + \gamma V^{\pi}(s') \middle| s_0 = s \right]$$

给定策略 π 和初始状态 s，则动作 a=π(s)，下个时刻将以概率 p(s'|s,a)转向下个状态 s'，那么上式的期望可以拆开，可以重写为：

$$V^{\pi}(s) = \sum_{s' \in S} p(s' \mid s, a)\left[r(s' \mid s, a) + \gamma V^{\pi}(s') \right]$$

上面提到的值函数称为状态值函数(state value function)，需要注意的是，在 $V^{\pi}(s)$中，π 和初始状态 s 是我们给定的，而初始动作 a 是由策略 π 和状态 s 决定的，即 a=π(s)。

定义动作值函数(action value functionQ 函数)如下：

$$Q^{\pi}(s, a) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i \middle| s_0 = s, a_0 = a \right]$$

给定当前状态 s 和当前动作 a，在未来遵循策略 π，那么系统将以概率 p(s'|s,a)转向下个状态 s'，上式可以重写为：

$$Q^{\pi}(s, a) = \sum_{s' \in S} p(s' \mid s, a)\left[r(s' \mid s, a) + \gamma V^{\pi}(s') \right]$$

在 $Q^{\pi}(s,a)$中，不仅策略 π 和初始状态 s 是我们给定的，当前的动作 a 也是我们给定的，这是 $Q^{\pi}(s,a)$和 $V^{\pi}(a)$的主要区别。

知道值函数的概念后，一个 MDP 的最优策略可以由下式表示：

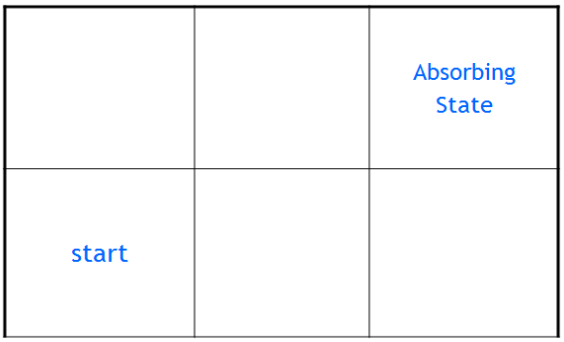
$$\pi^* = \arg\max_{\pi} V^{\pi}(s), (\forall s)$$

即我们寻找的是在任意初始条件 s 下，能够最大化值函数的策略 π*。

4. 值函数与 Q 函数计算的例子

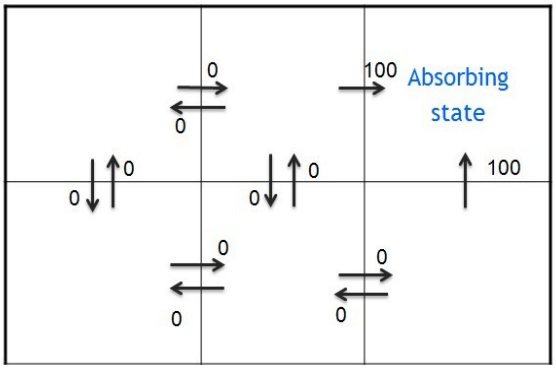
上面的概念可能描述得不够清晰，接下来我们实际计算一下，如图所示是一个格子世界，我们假设 agent 从左下角的 start 点出发，右上角为目标位置，称为吸收状态(Absorbing state)，对于进入吸收态的动作，我们给予立即回报 100，对其他动作则给予 0 回报，折合因子 γ 的值我们选择 0.9。

为了方便描述，记第 i 行，第 j 列的状态为 s_{ij}, 在每个状态，有四种上下左右四种可选的动作，分别记为 a_u,a_d,a_l,a_r。（up，down，left，right 首字母），并认为状态按动作 a 选择的方向转移的概率为 1。

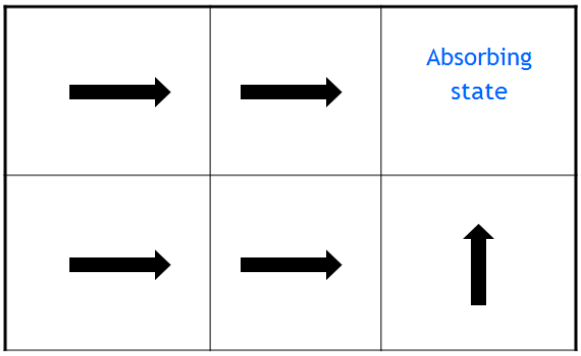


1.由于状态转移概率是 1，每组(s,a)对应了唯一的 s'。回报函数 r(s'|s,a)可以简记为 r(s,a)

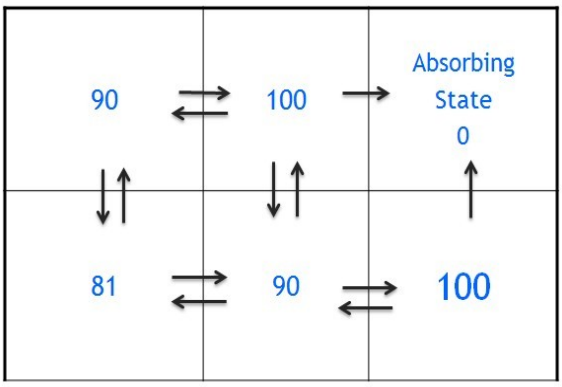
如下所示，每个格子代表一个状态 s，箭头则代表动作 a，旁边的数字代表立即回报，可以看到只有进入目标位置的动作获得了回报 100，其他动作都获得了 0 回报。　即 $r(s_{12},a_r) = r(s_{23},a_u) =100$。



2. 一个策略 π 如图所示：



3. 值函数 $V^{\pi}(s)$如下所示



根据 V^{π} 的表达式，立即回报，和策略 π，有

$V^{\pi}(s_{12}) = r(s_{12},a_r) = r(s_{13}|s_{12},a_r) = 100$

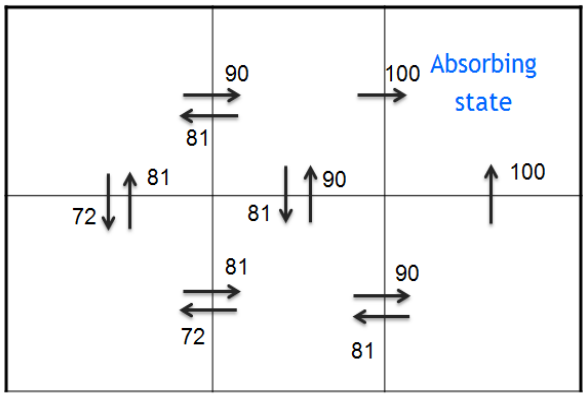
$V^{\pi}(s_{11})= r(s_{11},a_r)+\gamma*V^{\pi}(s_{12}) = 0+0.9*100 = 90$

$V^{\pi}(s_{23}) = r(s_{23},a_u) = 100$

$V^{\pi}(s_{22}) = r(s_{22},a_r)+\gamma*V^{\pi}(s_{23}) = 90$

$V^{\pi}(s_{21}) = r(s_{21},a_r)+\gamma*V^{\pi}(s_{22}) = 81$

4. Q(s,a)值如下所示



有了策略 π 和立即回报函数 r(s,a), $Q^{\pi}(s,a)$如何得到的呢?

对 s_{11} 计算 Q 函数（用到了上面 V^{π} 的结果）如下：

$Q^{\pi}(s_{11},a_r)=r(s_{11},a_r)+ \gamma *V^{\pi}(s_{12}) =0+0.9*100 = 90$

$Q^{\pi}(s_{11},a_d)=r(s_{11},a_d)+ \gamma *V^{\pi}(s_{21}) = 72$

至此我们了解了马尔可夫决策过程的基本概念，知道了增强学习的目标（获得任意初始条件下，使 V^{π} 值最大的策略 π^*），下一篇开始介绍求解最优策略的方法。

PS:发现写东西还是蛮辛苦的，希望对大家有帮助。另外自己也比较菜，没写对的地方欢迎指出~~

[注]采用折合因子作为值函数的 MDP 也可以定义为五元组 M=(S, A, P, γ, R)。也有的书上把值函数作为一个因子定义五元组。还有定义为三元组的，不过 MDP 的基本组成元素是不变的。

参考资料：

[1] R.Sutton et al. Reinforcement learning: An introduction , 1998

[2] T.Mitchell. 《机器学习》，2003

[3] 金卓军，逆向增强学习和示教学习算法研究及其在智能机器人中的应用[D]，2011

[4] Oliver Sigaud et al，Markov Decision Process in Artificial Intelligence[M], 2010

（三）MDP 的动态规划解法

上一篇我们已经说到了，增强学习的目的就是求解马尔可夫决策过程(MDP)的最优策略，使其在任意初始状态下，都能获得最大的 V^{π} 值。(本文不考虑非马尔可夫环境和不完全可观测马尔可夫决策过程(POMDP)中的增强学习)。

那么如何求解最优策略呢？基本的解法有三种：

动态规划法(dynamic programming methods)

蒙特卡罗方法(Monte Carlo methods)

时间差分法(temporal difference)。

动态规划法是其中最基本的算法，也是理解后续算法的基础，因此本文先介绍动态规划法求解 MDP。本文假设拥有 MDP 模型 M=(S, A, P_{sa}, R)的完整知识。

1. 贝尔曼方程（Bellman Equation）

上一篇我们得到了 V^{π} 和 Q^{π} 的表达式，并且写成了如下的形式

$$V^{\pi}(s)=\sum_{s'\in S}p(s'|s,\pi(s))\left[r(s'|s,\pi(s))+\gamma V^{\pi}(s')\right]=E_{\pi}\left[r(s'|s,a)+\gamma V^{\pi}(s')\middle|s_0=s\right]$$

$$Q^{\pi}(s,a)=\sum_{s'\in S}p(s'|s,a)\left[r(s'|s,a)+\gamma V^{\pi}(s')\right]=E_{\pi}\left[r(s'|s,a)+\gamma V^{\pi}(s')\middle|s_0=s,a_0=a\right]$$

在动态规划中，上面两个式子称为贝尔曼方程，它表明了当前状态的值函数与下个状态的值函数的关系。

优化目标 π*可以表示为：$\pi^*(s)=\arg\max_{\pi}V^{\pi}(s)$

分别记最优策略 π*对应的状态值函数和行为值函数为 V*(s)和 Q*(s, a)，由它们的定义容易知道，V*(s)和 Q*(s, a)存在如下关系：$V^*(s)=\max_a Q^*(s,a)$

状态值函数和行为值函数分别满足如下贝尔曼最优性方程(Bellman optimality equation)：

$$V^*(s)=\max_a E\left[r(s'|s,a)+\gamma V^*(s')\middle|s_0=s\right]$$

$$=\max_{a\in A(s)}\sum p(s'|s,\pi(s))\left[r(s'|s,\pi(s))+\gamma V^{\pi}(s')\right]$$

$$Q^*(s)=E\left[r(s'|s,a)+\gamma\max_{a'}Q^*(s',a')\middle|s_0=s,a_0=a\right]$$

$$=\sum p(s'|s,\pi(s))\left[r(s'|s,\pi(s))+\gamma\max_{a\in A(s)}Q^*(s',a')\right]$$

有了贝尔曼方程和贝尔曼最优性方程后，我们就可以用动态规划来求解 MDP 了。

2. 策略估计(Policy Evaluation)

首先，对于任意的策略 π，我们如何计算其状态值函数 $V^{\pi}(s)$？这个问题被称作策略估计，

前面讲到对于确定性策略，值函数 $$V^{\pi}(s)=\sum_{s'\in S}p(s'|s,\pi(s))\left[r(s'|s,\pi(s))+\gamma V^{\pi}(s')\right]$$

现在扩展到更一般的情况，如果在某策略 π 下，π(s)对应的动作 a 有多种可能，每种可能记为 π(a|s)，则状态值函数定义如下：

$$V^{\pi}(s)=\sum_{a}\pi(a|s)\sum_{s'\in S}p(s'|s,\pi(s))\left[r(s'|s,\pi(s))+\gamma V^{\pi}(s')\right]$$

一般采用迭代的方法更新状态值函数，首先将所有 $V^{\pi}(s)$的初值赋为 0（其他状态也可以赋为任意值，不过吸收态必须赋 0 值），然后采用如下式子更新所有状态 s 的值函数（第 k+1 次迭代）：

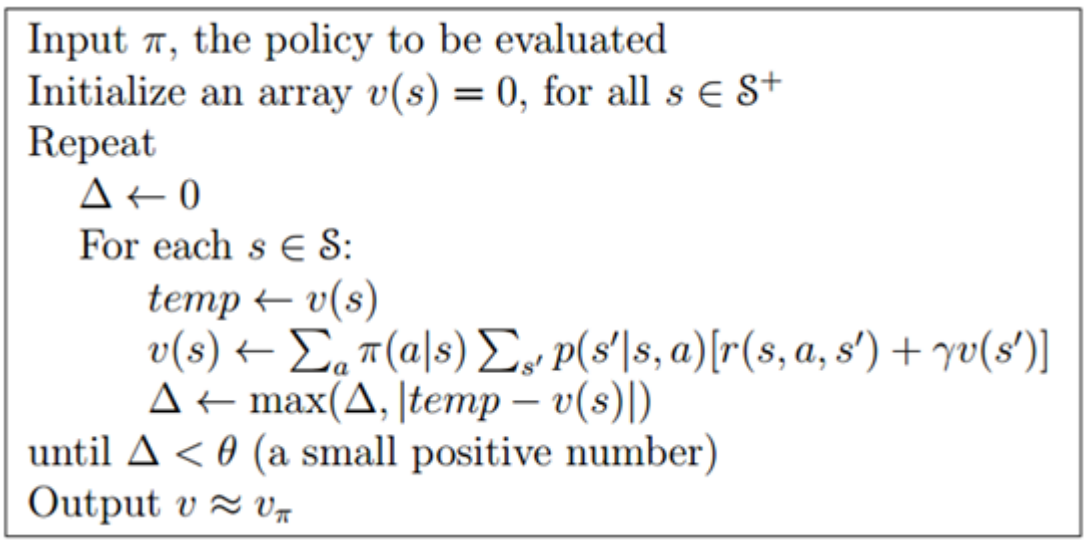
$$V_{k+1}(s)=\sum_{a}\pi(a|s)\sum_{s'\in S}p(s'|s,a)\left[r(s'|s,a)+\gamma V_{k}(s')\right]$$

对于 $V^{\pi}(s)$，有两种更新方法，

第一种：将第 k 次迭代的各状态值函数$[V^{k}(s_1),V^{k}(s_2),V^{k}(s_3)..]$保存在一个数组中，第 k+1 次的 $V^{\pi}(s)$采用第 k 次的 $V^{\pi}(s')$来计算，并将结果保存在第二个数组中。

第二种：即仅用一个数组保存各状态值函数，每当得到一个新值，就将旧的值覆盖,形如 $[V^{k+1}(s_1),V^{k+1}(s_2),V^{k}(s_3)..]$，第 k+1 次迭代的 $V^{\pi}(s)$可能用到第 k+1 次迭代得到的 $V^{\pi}(s')$。

通常情况下，我们采用第二种方法更新数据，因为它及时利用了新值，能更快的收敛。整个策略估计算法如下图所示：



3. 策略改进(Policy Improvement)

上一节中进行策略估计的目的，是为了寻找更好的策略，这个过程叫做策略改进(Policy Improvement)。

假设我们有一个策略 π，并且确定了它的所有状态的值函数 $V^{\pi}(s)$。对于某状态 s，有动作 $a_0=\pi(s)$。 那么如果我们在状态 s 下不采用动作 a_0，而采用其他动作 $a\neq\pi(s)$是否会更好呢？要判断好坏就需要我们计算行为值函数 $Q^{\pi}(s,a)$，公式我们前面已经说过：

$$Q^{\pi}(s,a)=\sum_{s'\in S}p(s'|s,a)\left[r(s'|s,a)+\gamma V^{\pi}(s')\right]$$

评判标准是：$Q^{\pi}(s,a)$是否大于 $V^{\pi}(s)$。如果 $Q^{\pi}(s,a)> V^{\pi}(s)$，那么至少说明新策略【仅在状态 s 下采用动作 a，其他状态下遵循策略 π】比旧策略【所有状态下都遵循策略 π】整体上要更好。

策略改进定理(policy improvement theorem)：π 和 π'是两个确定的策略，如果对所有状态 s∈S 有 $Q^{\pi}(s,\pi'(s))\geqslant V^{\pi}(s)$，那么策略 π'必然比策略 π 更好，或者至少一样好。其中的不等式等价于 $V^{\pi'}(s)\geqslant V^{\pi}(s)$。

有了在某状态 s 上改进策略的方法和策略改进定理，我们可以遍历所有状态和所有可能的动作 a，并采用贪心策略来获得新策略 π'。即对所有的 s∈S, 采用下式更新策略：

$$\begin{aligned}\pi'(s)&=\arg\max_a Q^{\pi}(s,a)\\&=\arg\max_a E_{\pi}\left[r(s'|s,a)+\gamma V^{\pi}(s')\middle|s_0=s,a_0=a\right]\\&=\arg\max_a\sum_{s'\in S}p(s'|s,a)\left[r(s'|s,a)+\gamma V^{\pi}(s')\right]\end{aligned}$$

这种采用关于值函数的贪心策略获得新策略，改进旧策略的过程，称为策略改进(Policy Improvement)

最后大家可能会疑惑，贪心策略能否收敛到最优策略，这里我们假设策略改进过程已经收敛，即对所有的 s，$V^{\pi'}(s)$等于 $V^{\pi}(s)$。那么根据上面的策略更新的式子，可以知道对于所有的 s∈S 下式成立：

$$\begin{aligned}V^{\pi'}(s)&=\max_a E\left[r(s'|s,a)+\gamma V^{\pi'}(s')\middle|s_0=s\right]\\&=\max_a\sum_{s'\in S}p(s'|s,a)\left[r(s'|s,a)+\gamma V^{\pi'}(s')\right]\end{aligned}$$

可是这个式子正好就是我们在 1 中所说的 Bellman optimality equation，所以 π 和 π'都必然是最优策略！神奇吧！

4. 策略迭代(Policy Iteration)

策略迭代算法就是上面两节内容的组合。假设我们有一个策略 π，那么我们可以用 policy evaluation 获得它的值函数 $V^{\pi}(s)$，然后根据 policy improvement 得到更好的策略 π'，接着再计算 $V^{\pi'}(s)$,再获得更好的策略 π''，整个过程顺序进行如下图所示：

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

完整的算法如下图所示：

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

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $temp \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma v(s')\right]$
      If $temp \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return v and π; else go to 2

5. 值迭代(Value Iteration)

从上面我们可以看到，策略迭代算法包含了一个策略估计的过程，而策略估计则需要扫描(sweep)所有的状态若干次，其中巨大的计算量直接影响了策略迭代算法的效率。我们必须要获得精确的 V^{π} 值吗？事实上不必，有几种方法可以在保证算法收敛的情况下，缩短策略估计的过程。

值迭代（Value Iteration）就是其中非常重要的一种。它的每次迭代只扫描(sweep)了每个状态一次。值迭代的每次迭代对所有的 s∈S 按照下列公式更新：

$$V_{k+1}(s) = \max_a E\left[r(s'|s,a) + \gamma V_k(s') \middle| s_0 = s\right]$$
$$= \max_a \sum p(s'|s,\pi(s))\left[r(s'|s,\pi(s)) + \gamma V_k(s')\right]$$

即在值迭代的第 k+1 次迭代时，直接将能获得的最大的 $V^{\pi}(s)$值赋给 V_{k+1}。值迭代算法直接用可能转到的下一步 s'的 V(s')来更新当前的 V(s)，算法甚至都不需要存储策略 π。而实际上这种更新方式同时却改变了策略 π_k 和 V(s)的估值 $V_k(s)$。 直到算法结束后，我们再通过 V 值来获得最优的 π。

此外，值迭代还可以理解成是采用迭代的方式逼近 1 中所示的贝尔曼最优方程。

值迭代完整的算法如图所示：

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

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

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma v(s')\right]$

由上面的算法可知，值迭代的最后一步，我们才根据 V*(s)，获得最优策略 π*。

一般来说值迭代和策略迭代都需要经过无数轮迭代才能精确的收敛到 V*和 π*， 而实践中，我们往往设定一个阈值来作为中止条件，即当 V^{π}(s)值改变很小时，我们就近似的认为获得了最优策略。在折扣回报的有限 MDP(discounted finite MDPs)中，进过有限次迭代，两种算法都能收敛到最优策略 π*。

至此我们了解了马尔可夫决策过程的动态规划解法，动态规划的优点在于它有很好的数学上的解释，但是动态要求一个完全已知的环境模型，这在现实中是很难做到的。另外，当状态数量较大的时候，动态规划法的效率也将是一个问题。下一篇介绍蒙特卡罗方法，它的优点在于不需要完整的环境模型。

PS: 如果什么没讲清楚的地方，欢迎提出，我会补充说明...

参考资料：

[1] R.Sutton et al. Reinforcement learning: An introduction , 1998

[2] 徐昕，增强学习及其在移动机器人导航与控制中的应用研究[D],2002

（四）蒙特卡罗方法

1. 蒙特卡罗方法的基本思想

蒙特卡罗方法又叫统计模拟方法，它使用随机数（或伪随机数）来解决计算的问题，是一类重要的数值计算方法。该方法的名字来源于世界著名的赌城蒙特卡罗，而蒙特卡罗方法正是以概率为基础的方法。

一个简单的例子可以解释蒙特卡罗方法，假设我们需要计算一个不规则图形的面积，那么图形的不规则程度和分析性计算（比如积分）的复杂程度是成正比的。而采用蒙特卡罗方法是怎么计算的呢？首先你把图形放到一个已知面积的方框内，然后假想你有一些豆子，把豆子均匀地朝这个方框内撒，散好后数这个图形之中有多少颗豆子，再根据图形内外豆子的比例来计算面积。当你的豆子越小，撒的越多的时候，结果就越精确。

2. 增强学习中的蒙特卡罗方法

现在我们开始讲解增强学习中的蒙特卡罗方法，与上篇的 DP 不同的是，这里不需要对环境的完整知识。蒙特卡罗方法仅仅需要经验就可以求解最优策略，这些经验可以在线获得或者根据某种模拟机制获得。

要注意的是，我们仅将蒙特卡罗方法定义在 episode task 上，所谓的 episode task 就是指不管采取哪种策略 π，都会在有限时间内到达终止状态并获得回报的任务。比如玩棋类游戏，在有限步数以后总能达到输赢或者平局的结果并获得相应回报。

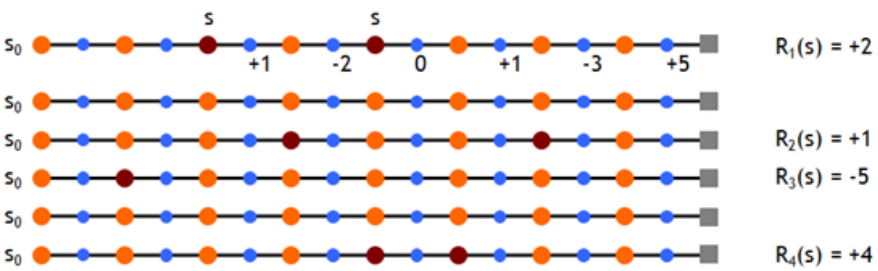
那么什么是经验呢？经验其实就是训练样本。比如在初始状态 s，遵循策略 π，最终获得了总回报 R，这就是一个样本。如果我们有许多这样的样本，就可以估计在状态 s 下，遵循策略 π 的期望回报，也就是状态值函数 $V^{\pi}(s)$了。蒙特卡罗方法就是依靠样本的平均回报来解决增强学习问题的。

尽管蒙特卡罗方法和动态规划方法存在诸多不同，但是蒙特卡罗方法借鉴了很多动态规划中的思想。在动态规划中我们首先进行策略估计，计算特定策略 π 对应的 V^{π} 和 Q^{π}，然后进行策略改进，最终形成策略迭代。这些想法同样在蒙特卡罗方法中应用。

3. 蒙特卡罗策略估计(Monte Carlo Policy evalution)

首先考虑用蒙特卡罗方法来学习状态值函数 $V^{\pi}(s)$。如上所述，估计 $V^{\pi}(s)$的一个明显的方法是对于所有到达过该状态的回报取平均值。这里又分为 first-visit MC methods 和 every-visit MC methods。这里，我们只考虑 first MC methods，即在一个 episode 内，我们只记录 s 的第一次访问，并对它取平均回报。

现在我们假设有如下一些样本，取折扣因子 γ=1，即直接计算累积回报，则有



根据 first MC methods，对出现过状态 s 的 episode 的累积回报取均值，有 $V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

容易知道，当我们经过无穷多的 episode 后，$V^{\pi}(s)$的估计值将收敛于其真实值。

4. 动作值函数的 MC 估计(Mote Carlo Estimation of Action Values)

在状态转移概率 p(s'|a,s)已知的情况下，策略估计后有了新的值函数，我们就可以进行策略改进了，只需要看哪个动作能获得最大的期望累积回报就可以。然而在没有准确的状态转移概率的情况下这是不可行的。为此，我们需要估计动作值函数 $Q^{\pi}(s,a)$。$Q^{\pi}(s,a)$的估计方法前面类似，即在状态 s 下采用动作 a，后续遵循策略 π 获得的期望累积回报即为 $Q^{\pi}(s,a)$，依然用平均回报来估计它。有了 Q 值，就可以进行策略改进了

$$\pi'(s) = \arg\max_a Q^{\pi}(s,a)$$

5. 持续探索(Maintaining Exploration)

下面我们来探讨一下 Maintaining Exploration 的问题。前面我们讲到，我们通过一些样本来估计 Q 和 V，并且在未来执行估值最大的动作。这里就存在一个问题，假设在某个确定状态 s_0 下，能执行 a_0, a_1, a_2 这三个动作，如果智能体已经估计了两个 Q 函数值，如 $Q(s_0,a_0), Q(s_0,a_1)$，且 $Q(s_0,a_0)>Q(s_0,a_1)$，那么它在未来将只会执行一个确定的动作 a_0。这样我们就无法更新 $Q(s_0,a_1)$的估值和获得 $Q(s_0,a_2)$的估值了。这样的后果是，我们无法保证 $Q(s_0,a_0)$就是 s_0 下最大的 Q 函数。

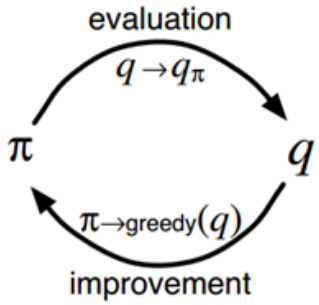
Maintaining Exploration 的思想很简单，就是用 *soft* policies 来替换确定性策略，使所有的动作都有可能被执行。比如其中的一种方法是 ε-greedy policy，即在所有的状态下，用 1-ε 的概率来执行当前的最优动作 a_0，ε 的概率来执行其他动作 a_1, a_2。这样我们就可以获得所有动作的估计值，然后通过慢慢减少 ε 值，最终使算法收敛，并得到最优策略。简单起见，在下面 MC 控制中，我们使用 exploring start，即仅在第一步令所有的 a 都有一个非零的概率被选中。

6. 蒙特卡罗控制(Mote Carlo Control)

我们看下 MC 版本的策略迭代过程：

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*$$

根据前面的说法，值函数 $Q^{\pi}(s,a)$的估计值需要在无穷多 episode 后才能收敛到其真实值。这样的话策略迭代必然是低效的。在上一篇 DP 中，我们了值迭代算法，即每次都不用完整的策略估计，而仅仅使用值函数的近似值进行迭代，这里也用到了类似的思想。每次策略的近似值，然后用这个近似值来更新得到一个近似的策略，并最终收敛到最优策略。这个思想称为广义策略迭代。



具体到 MC control，就是在每个 episode 后都重新估计下动作值函数（尽管不是真实值），然后根据近似的动作值函数，进行策略更新。这是一个 episode by episode 的过程。

一个采用 exploring starts 的 Monte Carlo control 算法，如下图所示，称为 Monte Carlo ES。而对于所有状态都采用 *soft* policy 的版本，这里不再讨论。

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

Repeat forever:
  (a) Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
    Generate an episode starting from S_0, A_0, following π
  (b) For each pair s,a appearing in the episode:
      $G \leftarrow$ return following the first occurrence of s,a
      Append G to $Returns(s,a)$
      $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  (c) For each s in the episode:
      $\pi(s) \leftarrow \arg\max_a Q(s,a)$

7. 小结

Monte Carlo 方法的一个显而易见的好处就是我们不需要环境模型了，可以从经验中直接学到策略。它的另一个好处是，它对所有状态 s 的估计都是独立的，而不依赖与其他状态的值函数。在很多时候，我们不需要对所有状态值进行估计，这种情况下蒙特卡罗方法就十分适用。

不过，现在增强学习中，直接使用 MC 方法的情况比较少，而较多的采用 TD 算法族。但是如同 DP 一样，MC 方法也是增强学习的基础之一，因此依然有学习的必要。

参考资料:

[1] R.Sutton et al. Reinforcement learning: An introduction, 1998

[2] Wikipedia，蒙特卡罗方法

（五）时间差分学习（O-learnning，Sarsa-learning)

接下来我们回顾一下动态规划算法(DP)和蒙特卡罗方法(MC)的特点，对于动态规划算法有如下特性：

- 需要环境模型，即状态转移概率 Psa
- 状态值函数的估计是自举的(*bootstrapping*)，即当前状态值函数的更新依赖于已知的其他状态值函数。

相对的，蒙特卡罗方法的特点则有：

- 可以从经验中学习不需要环境模型
- 状态值函数的估计是相互独立的
- 只能用于 episode tasks

而我们希望的算法是这样的：

- 不需要环境模型
- 它不局限于 episode task，可以用于连续的任务

本文介绍的**时间差分学习**(Temporal-Difference learning, TD learning)正是具备了上述特性的算法，它结合了 DP 和 MC，并兼具两种算法的优点。

TD Learing 思想

在介绍 TD learning 之前，我们先引入如下简单的蒙特卡罗算法，我们称为 **constant-α**

MC，它的状态值函数更新公式如下：

$$V(st)\leftarrow V(st)+\alpha[Rt-V(st)](1)$$

其中 Rt 是每个 episode 结束后获得的实际累积回报，α 是学习率，这个式子的直观的理解就是**用实际累积回报 Rt 作为状态值函数 $V(st)$的估计值**。具体做法是对每个 episode，考察实验中 st 的实际累积回报 Rt 和当前估计 $V(st)$的偏差值，并用该偏差值乘以学习率来更新得到 $V(St)$

的新估值。

现在我们将公式修改如下，把 Rt

换成 $rt+1+\gamma V(st+1)$，就得到了 TD(0)的状态值函数更新公式：

$$V(st)\leftarrow V(st)+\alpha[rt+1+\gamma V(st+1)-V(st)](2)$$

$$V\pi(s)=E\pi[r(s'|s,a)+\gamma V\pi(s')](3)$$

容易发现这其实是根据(3)的形式，利用真实的立即回报 $rt+1$ 和下个状态的值函数 $V(st+1)$来更新 $V(st)$

，这种就方式就称为时间差分(temporal difference)。由于我们没有状态转移概率，所以要利用多次实验来得到期望状态值函数估值。类似 MC 方法，在足够多的实验后，状态值函数的估计是能够收敛于真实值的。

那么 MC 和 TD(0)的更新公式的有何不同呢？我们举个例子，假设有以下 8 个 episode, 其中 A-0 表示经过状态 A 后获得了回报 0:

index	samples
episode 1	A-0, B-0
episode 2	B-1
episode 3	B-1
episode 4	B-1
episode 5	B-1
episode 6	B-1
episode 7	B-1
episode 8	B-0

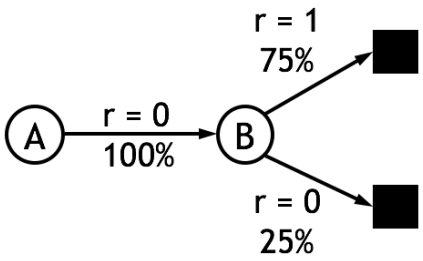
首先我们使用 constant-α

MC 方法估计状态 A 的值函数，其结果是 $V(A)=0$

，这是因为状态 A 只在 episode 1 出现了一次，且其累计回报为 0。

现在我们使用 TD(0)的更新公式，简单起见取 $\lambda=1$

，我们可以得到 $V(A)=0.75$。这个结果是如何计算的呢？ 首先，状态 B 的值函数是容易求得的，B 作为终止状态，获得回报 1 的概率是 75%，因此 $V(B)=0.75$。接着从数据中我们可以得到状态 A 转移到状态 B 的概率是 100%并且获得的回报为 0。根据公式(2)可以得到 $V(A)\leftarrow V(A)+\alpha[0+\lambda V(B)-V(A)]$，可见在只有 $V(A)=\lambda V(B)=0.75$ 的时候，式(2)收敛。对这个例子，可以作图表示：



可见式(2)由于能够利用其它状态的估计值，其得到的结果更加合理，并且由于不需要等到任务结束就能更新估值，也就不再局限于 episode task 了。此外，实验表明 TD(0)从收敛速度上也显著优于 MC 方法。

将式(2)作为状态值函数的估计公式后，前面文章中介绍的**策略估计**算法就变成了如下形式，这个算法称为 TD prediction:

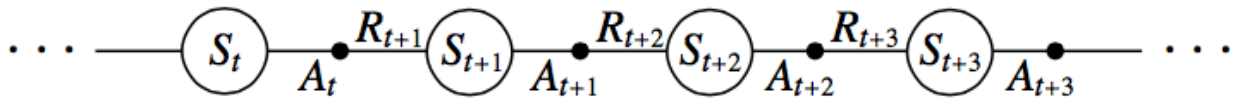
输入：待估计的策略 π

任意初始化所有 $V(s)$，$(e.g.,V(s)=0, \forall s \in s+)$
Repeat(对所有 episode):
    初始化状态 s
    Repeat(对每步状态转移):
        $a \leftarrow$*策略* π *下状态* s *采取的动作*
        采取动作 a，*观察回报* r，*和下一个状态* s'
        $V(s) \leftarrow V(s)+\alpha[r+\lambda V(s')-V(s)]$
        $s \leftarrow s'$
    Until st *is terminal*
  Until 所有 $V(s)$*收敛*
输出 $V\pi(s)$

Sarsa 算法

现在我们利用 TD prediction 组成新的强化学习算法，用到决策/控制问题中。在这里，强化学习算法可以分为**在策略(on-policy)**和**离策略(off-policy)**两类。首先要介绍的 sarsa 算法属于 on-policy 算法。
与前面 DP 方法稍微有些区别的是，sarsa 算法估计的是**动作值函数(Q 函数)**而非状态值函数。也就是说，我们估计的是策略 π

下，任意状态 s 上所有可执行的动作 a 的动作值函数 $Q\pi(s,a)$，Q 函数同样可以利用 TD Prediction 算法估计。如下就是一个状态-动作对序列的片段及相应的回报值。



给出 sarsa 的动作值函数更新公式如下：

$$Q(st,at) \leftarrow Q(st,at)+\alpha[rt+1+\lambda Q(st+1,at+1)-Q(st,at)] \quad (4)$$

，在到达下个状态 $st+1$ 后，都可以利用上述公式更新 $Q(st,At)$，而如果 st 是终止状态，则要令 $Q(st+1=0,at+1)$。由于动作值函数的每次更新都与$(st,at,rt+1,st+1,at+1)$相关，因此算法被命名为 sarsa 算法。sarsa 算法的完整流程图如下：

Initialize $Q(s,a)$ arbitrarily
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

算法最终得到所有状态-动作对的 Q 函数，并根据 Q 函数输出最优策略 π

Q-learning

在 sarsa 算法中，**选择动作时遵循的策略**和**更新动作值函数时遵循的策略**是相同的，即 $\epsilon-greedy$ 的策略，而在接下来介绍的 Q-learning 中，动作值函数更新则**不同于选取动作时遵循的策略**，这种方式称为**离策略(Off-Policy)**。Q-learning 的动作值函数更新公式如下：

$$Q(st,at) \leftarrow Q(st,at) + \alpha[rt+1+\lambda \max_a Q(st+1,a) - Q(st,at)] \quad (5)$$

可以看到，Q-learning 与 sarsa 算法最大的不同在于**更新 Q 值的时候，直接使用了最大的 $Q(st+1,a)$值——相当于采用了 $Q(st+1,a)$值最大的动作，并且与当前执行的策略，即选取动作 at 时采用的策略无关。** Off-Policy 方式简化了证明算法分析和收敛性证明的难度，使得它的收敛性很早就得到了证明。Q-learning 的完整流程图如下：

Initialize $Q(s,a)$ arbitrarily
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

小结

本篇介绍了 TD 方法思想和 TD(0),Q(0),Sarsa(0)算法。TD 方法结合了蒙特卡罗方法和动态规划的优点，能够应用于无模型、持续进行的任务，并拥有优秀的性能，因而得到了很好的发展，其中 Q-learning 更是成为了强化学习中应用最广泛的方法。在下一篇中，我们将引入**资格迹(Eligibility Traces)**提高算法性能，结合 Eligibility Traces 后，我们可以得到 $Q(\lambda)$,$Sarsa(\lambda)$

等算法

参考资料

[1] R.Sutton et al. Reinforcement learning: An introduction, 1998