1. **蒙特卡洛、TD、动态规划的关系**

三者之间的相同点：如果说三者的相同点 那么也只有都是用于进行值函数的描述与更新的了；

两者之间的共同点：蒙特卡洛和TD算法隶属于model-free，而动态规划属于model-based；TD算法和蒙特卡洛的方法 因为都是基于model-free的方法，因而对于后续状态的获知 也都是基于试验的方法；TD算法和动态规划的策略评估都能基于当前状态的下一步预测情况来得到对于当前状态的值函数的更新

两者之间的不同点：TD算法和蒙特卡洛的方法当然也有不同之处在于：TD算法不需要等到实验结束后 才能进行当前状态的值函数的计算与更新，而蒙特卡洛的方法需要试验交互 产生一整条的马尔科夫链「到最终状态」才能进行更新。TD算法和动态规划的策略评估不同之处还是就回到model-free和model-based 这一点上来了，动态规划可以凭借已知转移概率就能推断出来后续的状态情况，而TD只能借助试验才能知道了「因而 在强化学习的实验中 设置多次episode 以遍历尽量多的状态 和马尔科夫链 来降低方差」

蒙特卡洛方法和TD方法的一个区别，因为蒙特卡洛方法进行完整的采样来获取了长期的回报值，因而在价值估计上会有着更小的偏差，但是也正因为收集了完整的信息 所以价值的方差会更大「毕竟基于试验的采样得到 和真实的分布还是有差距 不充足的交互导致的较大方差」；而TD算法与其相反 因为只考虑了前一步的回报值 其他都是基于之前的估计值 因而估计具有偏差 但方差也较小；

1. **off-policy和on-policy的区别**

对于两者区别的划分众所周知：就是看选取动作的策略和进行评估的策略是不是同一个策略就是了；而一般不同之处都是在评估的时候，进行对于下一个状态值函数预测的时候，qlearning选用的直接是next\_state的最大的那个Q动作值函数，而sarsa还是依策略的和之前选取state时候动作action一样的选取next\_state的动作action\_进而确定Q动作值函数；举个SARSA和qlearning的例子就是：前者基于某一个策略如ε-greedy进行选取action，后面需要值函数进行更新的时候 也还是ε-greedy策略。后者则不然 选取action的策略是基于ε-greedy，更新值函数的时候依旧是选取max的那个；

on-policy优点因为是单纯的依照交互数据来进行更新，所以直接了当。但速度不一定快 毕竟试验交互时候还有样本的采样效率的问题，可以看到off-policy可以用到之前的经验，而on-policy只能从新采样，因而速度不一定快。同时劣势是不一定找到最优策略。

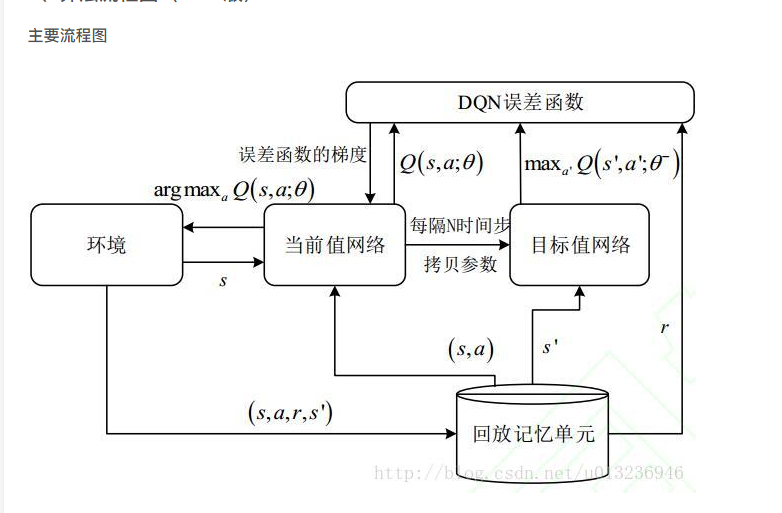
而off-policy优势在于更强的通用性 保证了探索性，其强大是因为它确保了数据全面性，所有行为都能覆盖。「这里所说的就是在选取snext对应action的时候需要选取max的一个 那么需要全部action的q值 于是覆盖了全部的行为，相比之下on-policy只是单纯的依照策略选取了某个action；」但劣势也很明显 就是每次对于状态的值函数的估计 都是过高的进行估计。同时毕竟是基于采样的方式的 所以会有些状态没有被采样到，这就产生的偏差

1. **值函数近似**

表格式到函数近似

本质上还是为了建立从状态到值函数 或者状态动作到值函数之间的映射，既然是映射，那么就肯定存在一个函数可以来表示这个映射关系，参考机器学习中的各类方法，我们可以建立一个模型来表示这个从状态到值函数 或者状态动作到值函数之间的映射关系，进而就转化为标准的机器学习中有监督学习的回归问题求解：选择好算法，损失函数，优化方法，进而就能完成求解

1. **DQN**



很显然DQN相比于qlearning主要有三处改变：

首先DQN采用了深度卷积神经网络来进行的值函数逼近，这里选取卷积神经网络的原因也在于 原文是针对Atari游戏来作为environment的，输入状态采用的是84x84的图片，那么这里其实就涉及一个深度学习中一个选取问题就是CNN的优势 毕竟不是深度学习的部分 这里直接说结论，CNN凭借着本身稀疏连接和参数共享 相比于FCN来说 计算量大大减少，存储的参数量也大大减少；进而本身有着的局部特征和平移等变性 也让CNN很适合用于处理图片。当然尽管说这么多，这也不是什么新鲜操作了；但加入了第二点 就完全不同了，解决了RL里面的一个痛点；

加入了经验回放的操作来训练强化学习；首先我们要知道 如果直接借助强化学习交互产生的数据 本身是带有关联性的，而在神经网络或者直接说机器学习中 对于数据的基本要求就是独立同分布。因而这里引入了经验回放这个操作 来打破了数据间的关联性，具体操作就是：agent在与环境交互的时候 将交互数据存放在一个库里面，然后训练的时候 从中随机采样数据进行训练；

加入了目标网络的概念来单独的处理TD算法中的TD偏差；首先这一点并没有在13年版的DQN里面使用，而是在15年版的DQN才出现的，这也是13年和15年版的区别。我们都知道 强化学习在表格式的时候 直接使用Q∗(s,a)=Q(s,a)+α(r+γmaxa′Q(s′,a′)−Q(s,a))Q∗(s,a)=Q(s,a)+α(r+γmaxa′Q(s′,a′)−Q(s,a)) 来对于值函数就直接更新了；但函数近似后不行，现在的值函数本质上就是个函数的形式如Q=f(θ;feature)Q=f(θ;feature) ，那么对于它的优化与更新也更多的是采用诸如梯度下降等方法来对于权值θ进行更新，有：TargetQ=r+γmaxa′Q(s′,a′;θ) TargetQ=r+γmaxa′Q(s′,a′;θ) 和 L(θ)=E[(TargetQ−Q(s,a;θ))2]L(θ)=E[(TargetQ−Q(s,a;θ))2] 。因而目标网络的改进就在于这里的TargetQTargetQ使用了一个单独的网络 ；「emmm 这么一说 本来qlearning里面就不是按照交互顺序 而是使用的最大值来得到的对于当前值函数的估计，现在又进一步打破关联性 使用另一个网络的输出来进行估计 进一步减少关联性」也就是说现在有着TargetQ=r+γmaxa′Q(s′,a′;θ−)TargetQ=r+γmaxa′Q(s′,a′;θ−) ，对于这个target 网络的权值θ−θ− 和前面的计算Q值的主网络不同，主网络是每一步一更新，而target网络每隔一段时间一更新；「联系后面的DDQN的话 两者的区别其实也就在于：DDQN是在选取动作的时候基于一个网络 而评价是基于另一个网络；也就是TargetQ=R+γQ(St+1,argmaxaQ(St+1,a;θt);θ′t)TargetQ=R+γQ(St+1,argmaxaQ(St+1,a;θt);θt′) 可以看到区别就是对于targetQ构建的时候 next state的action和q值选取还是不一样的」

1. **深度强化学习中的DQN里面的experience replay方法和A3C的asynchronous**

显而易见的 前者的经验重放就是一种标准的off-policy的方法，毕竟在DQN里面使用的时候是在对估计Q(s,a)，estimate\_Q(s, a) = r + lamda \* maxQ(s2, a) ，如果按照on-policy理解的话 这个时候的应该是estimate\_Q(s, a) = r + lamda \* Q(s2, a2) 也就是说这个时候应该保存当时候的网络模型来进行输出对应的Q值情况，进而进行基于相应的值函数 在基于相同的如ε-greedy来选择情况；现在经验重播中只保存了<s,a,r,snext> 谈何on-policy呢。

同时上面也说了是直接保存的交互数据进行训练，这样观察数据往往波动很大且前后sample相互关联「机器学习也要求样本彼此独立同分布」 经验重播的方法很不适合on-policy；

但多线程的synchronous不一样；因为是多个agent在多个环境实例中并行且异步的执行和学习。 数据就不存在上面所说的数据关联性的问题。多个并行的actor可以有助于exploration。在不同线程上使用不同的探索策略，使得经验数据在时间上的相关性很小。这样不需要DQN中的experience replay也可以起到稳定学习过程的作用，意味着学习过程可以是on-policy的。

1. **policy-based**

最优控制中求解最优值函数进而求解得到策略的思路

接在策略空间直接进行搜索，不再借助值函数那样的间接完成对于最优策略的确定；于是也带来了收敛性快这样的好处；方法包括策略梯度 DDPG等

同时和value-based相比，value-based对于连续动作 不能对每个动作都给予一个Q值 因而在连续动作集合中就不能有很好的表现；而policy-based可以有效地处理连续动作集的问题「值函数方法无法确定一个对应max Q的action」，但同样的容易收敛到局部最小值、方差较大。「这里的方差较大的原因其实和蒙特卡洛方法类似 每次交互都会产生一整条轨迹 然后基于这个策略更新的时候 会导致回报估计的波动 因而会导致高方差」

1. **actor-critic**

对于策略梯度这种因为采样问题导致的高方差，就像之前提到过MC和TD存在的问题一样，MC基于交互产生整条马尔科夫链而进行估计值函数，因为采样的问题 导致的高方差，而TD就只是利用连续两个时刻预测值的差值来更新模型，尽管基于后续状态对于当前状态进行估计，因此计算出来的策略梯度都存在偏差，但同样的换来较小的方差；

同样的思路：构建出来一个独立的模型来估计模型的长期误差，而不是单纯的使用轨迹的真实分布；进而产生了作为基于值函数方法和基于策略梯度方法的actor-critic方法；

actor是基于策略梯度的方法进行选取动作，而critic是基于值函数的方法来评价它，两者协作完成；

Actor - Critic的优点

（总的来说就是结合两种方式优点）：

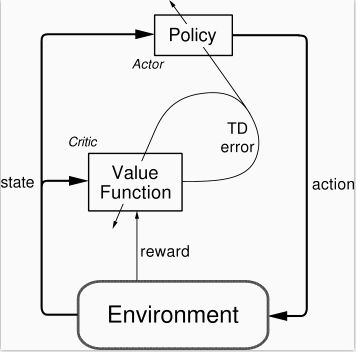
相比以值函数为中心的算法，Actor - Critic应用了策略梯度的做法，这能让它在连续动作或者高维动作空间中选取合适的动作, 而 Q-learning 做这件事会很困难甚至瘫痪。

相比单纯策略梯度，Actor - Critic应用了Q-learning或其他策略评估的做法，使得Actor Critic能进行单步更新而不是回合更新，比单纯的Policy Gradient的效率要高。最重要的还是不再使用采样得到的真实回报 降低了因为采样率导致的方差

Actor - Critic和两者的区别

和策略梯度的区别，很明显 关于策略梯度的算法表述里面 并没有涉及关于critic部分或者具体点说就是值函数更新的部分；参考RL:an introduction 里面所描述的那样P292中所描述的那样： 策略梯度方法里 状态值函数更多的是作为一种基准而非是critic；也就是说只是作为一种基准来判断哪个状态需要被更新「for state-function」而事实上 为了实现Policy Gradient，不管我们是计算Q，还是V，都需要一个对应的网络，这就是Critic。换句话讲，我们只有在使用Policy Gradient时完全不使用Q，仅使用reward真实值来评价，才叫做Policy Gradient，要不然Policy Gradient就需要有Q网络或者V网络，就是Actor Critic。 「对于其中采用什么值来当做R(τ) 参考这来理解https://zhuanlan.zhihu.com/p/26882898」

和值函数的方法 差别就更大了；首先对于策略的描述 就不是值函数方法里面借助 argmaxaQ(s,a) 而是使用的策略梯度的方法；当然使用对于该策略的好坏的评价 但也是一样的值函数；



如图 基于策略梯度的actor基于概率来对于某状态来选取动作action；而critic基于actor的行为判别行为的得分；actor进而基于该评价值来计算出来一个td error修改选择行为的概率「换句话说就是：actor的策略梯度的方法生成得到梯度的方向「也就说之前的logP(τ;θ)logP(τ;θ)」，然后进行沿着方向进行梯度的增减；我们需要一个值来判断这一个增减的方向是否正确 于是需要critic来计算出来td error」，同时基于选取的action计算TD error来更新critic；

具体到网络里面：Actor和Critic各为一个网络，Actor输入是状态输出的是动作，loss就是logprob×tderrorlogprob×tderror,(和策略梯度相对应，注意到这里的loss和Policy Gradient中的差不多，只是vt换成td\_error，引导奖励值vt换了来源（Critic给的）而已)，Critic输入的是状态输出的是Q值，loss是square((r+gamma\*Q\_next) - Q\_eval)也就是square(td\_error)，也就是说这里更新critic对应Q-learning是一样的均方误差。

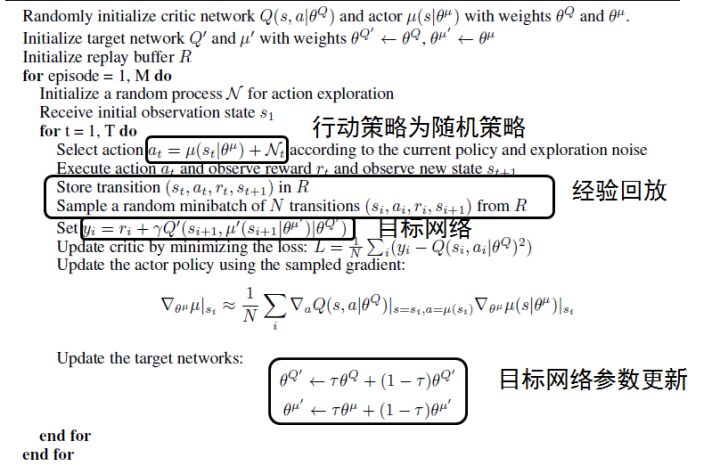
1. **A3C**

A3C的全名是Asynchronous Advantage Actor-Critic

而是将one-step Sarsa, one-step Q-learning, n-step Q-learning和advantage AC扩展至多线程异步架构。 可以看到几种方法的不同： 「AC是on-policy的policy搜索方法，而Q-learning是off-policy value-based方法。这也体现了该框架的通用性。 」

1. **DPG和DDPG**

DDPG 和DPG的区别也只是在于：使用DNN来逼近值函数和确定性策略



1. **强化学习解决的是什么样的问题？**

我答：“序列决策问题。”

1. **强化学习的损失函数（loss function）是什么**

log likelihood——对数似然函数值，更常使用log likelihood log： log (pi(a|s,theta)), 我们就可以构造一个损失函数如下：

L(theta) = sum( log(pi(a|s,theta)\*f(s,a) ))

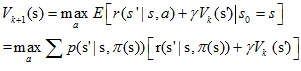
1. **贝尔曼方程**

“贝尔曼方程（Bellman Equation）”也被称作“动态规划方程（Dynamic Programming Equation）”，由理查·贝尔曼（Richard Bellman）发现。贝尔曼方程是动态规划（Dynamic Programming）这种数学最佳化方法能够达到最佳化的必要条件。此方程将“决策问题在特定时间点的值”以“来自初始选择的报酬 及 由初始选择衍生的决策问题的值”的形式表示。藉这个方式将动态最佳化问题变成较简单的子问题，而这些子问题遵守由贝尔曼所提出的“最佳化原理”。

1. **最优值函数和最优策略为什么等价？**

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Vπ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：



1. **使得策略梯度的方差最小时的基线b**

给出的策略梯度是无偏的，但是方差很大。我们在回报中引入常数基线b来减小方差

首先，证明当回报中引入常数b时，策略梯度不变

1. **强化学习的损失函数（loss function）是什么？**

值函数近似（Function Approximation）的方法就是为了解决状态空间过大，也称为“维度灾难”的问题。通过用函数而不是Q表来表示 Q( s,a ) ，这个函数可以是线性的也可以使非线性的。

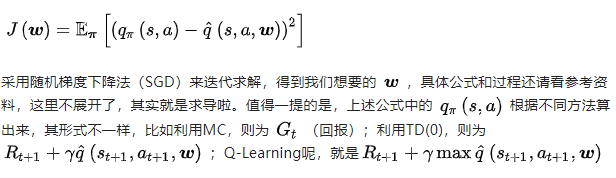


其中{w} 称为“权重”。那怎么把这个权重求出来，即拟合出这样一个合适的函数呢？这里就要结合机器学习算法里的一些有监督学习算法，对输入的状态提取特征作为输入，通过MC/TD计算出值函数作为输出，然后对函数参数 {w} 进行训练，直到收敛。这里主要说的是回归算法，比如线性回归、决策树、神经网络等。

这里，就可以引入DQN（Deep Q-Network）了，实际上它就是Q-Learning和神经网络的结合，将Q-Learning的Q表变成了Q-Network。

好，现在关键问题来了。这么去训练这个网络呢？换句话说，怎么去确定网络参数 {w} 呢？第一，我们需要一个Loss Function；第二，我们需要足够的训练样本。

训练样本好说，通过 epsilon -greedy 策略去生成就好。回忆一下Q-Learning，我们更新Q表是利用每步的reward和当前Q表来迭代的。那么我们可以用这个计算出来的Q值作为监督学习的“标签”来设计Loss Function，我们采用如下形式，即近似值和真实值的均方差



在David Silver的课里，他根据每次更新所参与样本量的不同把更新方法分为增量法（Incremental Methods）和批处理法（Batch Methods）。前者是来一个数据就更新一次，后者是先攒一堆样本，再从中采样一部分拿来更新Q网络，称之为“经验回放”，实际上DeepMind提出的DQN就是采用了经验回放的方法。为什么要采用经验回放的方法？因为对神经网络进行训练时，假设样本是独立同分布的。而通过强化学习采集到的数据之间存在着关联性，利用这些数据进行顺序训练，神经网络当然不稳定。经验回放可以打破数据间的关联。

我们知道在监督式学习中一般会选择一种 loss function, 如 Square loss, Hinge loss（合页误差）, Logistic loss 等，来表示真实值和实际值的差距从而据此在反向传播中进行参数的更新。在策略梯度学习中同样需要类似的函数表示当前的效果，这就是目标函数。

策略梯度的目标函数

根据上述策略梯度的基本原理，我们可以把它的目标形式化的描述为以下表达式：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image017.png

其中 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image018.png 表示在策略 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image019.png 条件下一轮交互（ https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image020.png 到 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image021.png 步）中的累计奖励的期望值，这里是期望而不是确定值是因为每一步的奖励是根据策略得到的期望值而不是确定值。由于 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image022.png 可以由 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image023.png 计算得到，因此可以把累计奖励的期望值写成如下：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image024.png

我们把单个轮次的累计奖励作为 PG 的目标函数 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image025.png，则策略梯度的目标就是确定构成策略的参数 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image026.png 使得 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image025.png 取得最大的期望值：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image027.png

现在策略梯度的学习就变成了一个对 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image025.png 求最大值的问题，和监督式学习中使用的梯度下降 (gradient descent) 求损失函数 (loss function) 的最小值类似，策略梯度中使用梯度上升 (gradient ascent) 来更新 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image026.png 根据期望值的数学定义：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image028.png

对这个积分表达式求导数：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image029.png

由于 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image030.png 本身依赖于 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image026.png，我们无法直接求导，因此要使用一个小技巧，根据 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image031.png可得下式：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image032.png

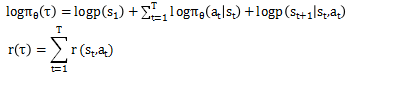
因此：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image033.png

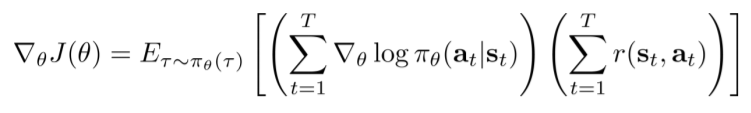
再根据期望值的定义：

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image034.png

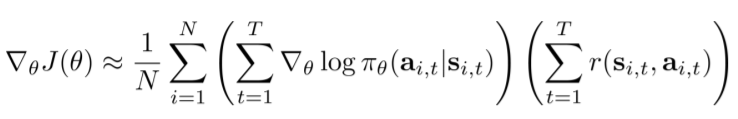
由于：



可得：



还需要指出的是上述表达式描述了当前策略 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image019.png 通过一轮获得的导数，前面我们已经提到过由于策略产生的是非确定的动作，因此相同策略在多轮次中会产生不同的轨迹，为了避免个体的偏差，我们需要多次取样并取均值来提高准确性，所以，



至此，我们就得到了可计算的目标函数的导数https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image039.png，在轮次的反向传播 (back propagation) 中使用学习率 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image040.png与https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image039.png的乘积作为差值https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image041.png更新https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image026.png

https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image042.png

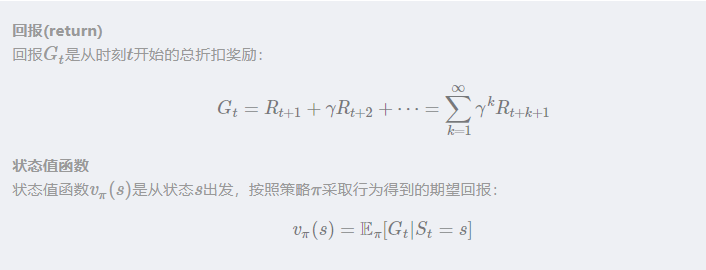
以上为了推导用于反向传播的可计算的https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image039.png列出了很多表达式，目的是帮助读者理解策略梯度的算法实现，因为在代码实现中会直接使用表达式 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image043.png 计算 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image039.png，如果直接看代码而不了解 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image039.png 的推导过程的话恐怕会觉得费解。不过从 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image044.png 和 https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image045.png 的基本关系还是能够做出这样的直观解释：如果奖励 (https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image046.png) 比较高时，策略 (https://www.ibm.com/developerworks/cn/analytics/library/ba-lo-deep-introduce-policy-gradient/image030.png) 会倾向于增加相应的动作的概率，如果奖励比较低时，策略会倾向于降低相应动作的概率。从机器学习的原理的角度来看，策略梯度和传统的监督式学习的学习过程还是比较相似的，每轮次都由前向反馈和反向传播构成，前向反馈负责计算目标函数，反向传播负责更新算法的参数，依此进行多轮次的学习指导学习效果稳定收敛。唯一不同的是，监督式学习的目标函数相对直接，即目标值和真实值的差，这个差值通过一次前向反馈就能得到；而策略梯度的目标函数源自轮次内所有得到的奖励，并且需要进行一定的数学转换才能计算，另外由于用抽样模拟期望，也需要对同一套参数进行多次抽样来增加模拟的准确性。

1. **写出贝尔曼方程**

<https://blog.csdn.net/victoriaw/article/details/78839929>

* 值函数是对未来奖励的一个预测

回报(return)



* 状态值函数

状态值函数可以用来评价状态的好坏

状态值函数V(s)是从状态s出发，按照策略π采取行为得到的期望回报：

根据定义可以得到：

vπ(s)=Eπ[Rt+1+γGt+1|St=s]=Eπ[Rt+1+γvπ(St+1)|St=s]

* 行为值函数

行为值函数qπ(s,a)是从状态s出发，采取行为a后，然后按照策略π采取行为得到的期望回报：

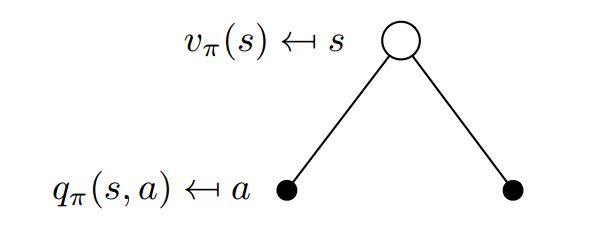
qπ(s,a)=Eπ[Gt|St=s,At=a]

qπ(s,a)=E[Rt+1+γqπ(St+1,At+1)|St=s,At=a]

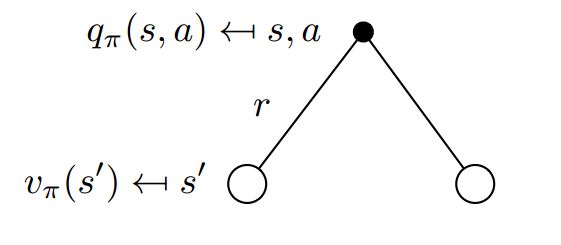
根据定义可以得到：

* Bellman期望方程其实就是vπ(s)和qπ(s,a)自身以及相互之间的递推关系

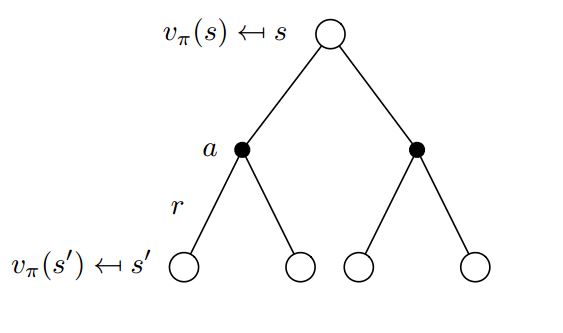
vπ(s)=∑a∈Aπ(a|s)qπ(s,a)



qπ(s,a)=Ras+γ∑s′∈SPass′vπ(s′)(2)

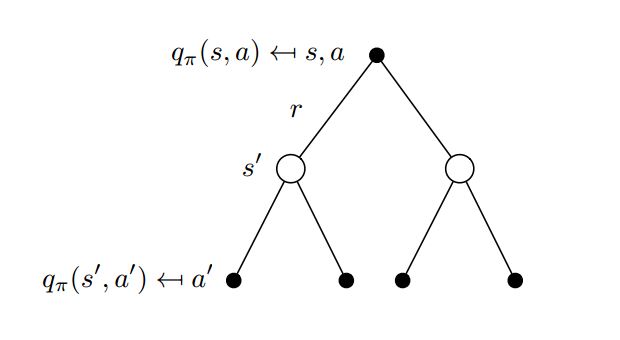


vπ(s)=∑aπ(a|s)[Ras+γ∑s′Pass′vπ(s′)]

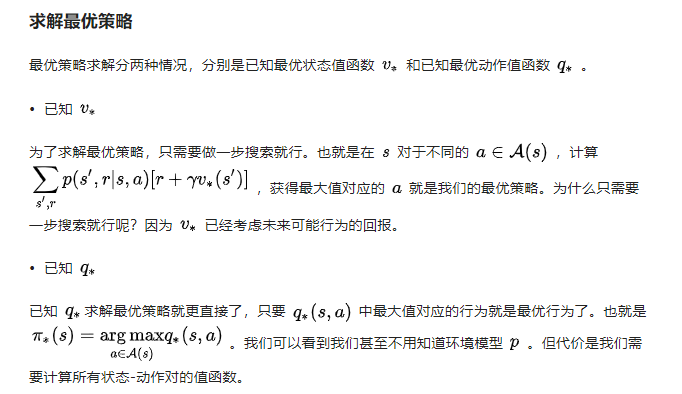


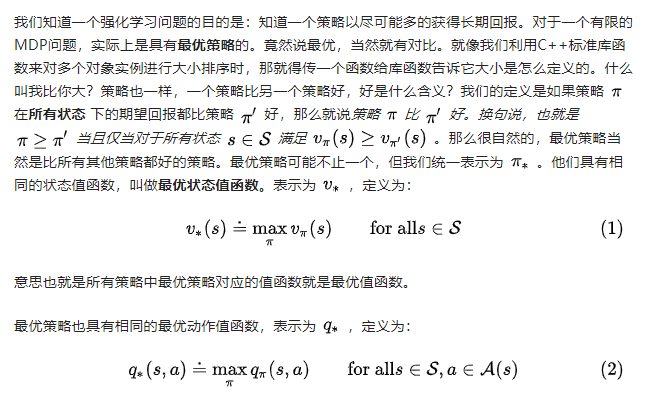
qπ(s,a)qπ(s,a)自身的递推关系

把公式(1)代入(2)中得到：

qπ(s,a)=Ras+γ∑s′Pass′∑a′π(a′|s′)qπ(s′,a′)

1. **最优值函数和最优策略为什么等价？**





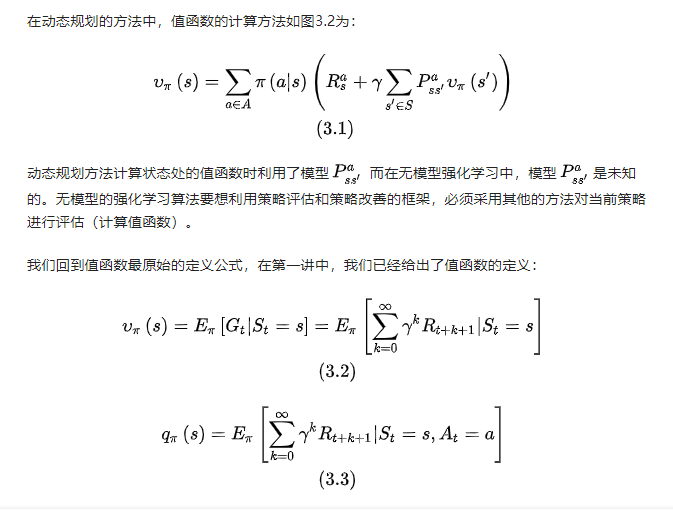
1. **求解马尔科夫决策过程都有哪些方法？有模型用什么方法？动态规划是怎么回事？**

马尔科夫决策过程(MDP)中状态价值函数的贝尔曼方程：

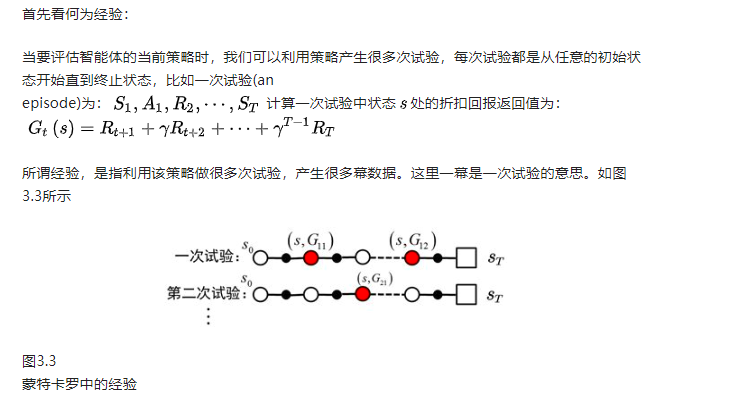


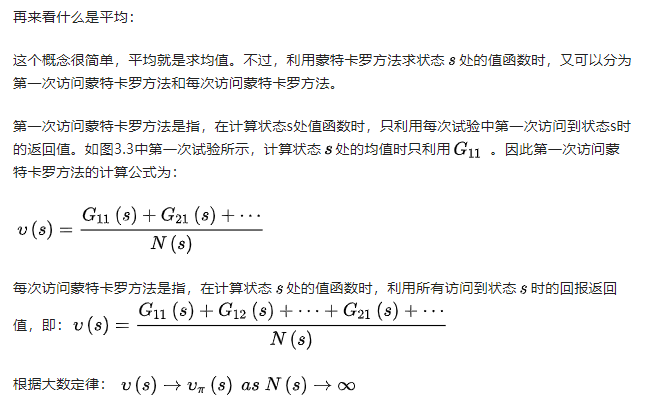
从这个式子我们可以看出，我们可以定义出子问题求解每个状态的状态价值函数，同时这个式子又是一个递推的式子, 意味着利用它，我们可以使用上一个迭代周期内的状态价值来计算更新当前迭代周期某状态s的状态价值。可见，使用动态规划来求解强化学习问题是比较自然的。

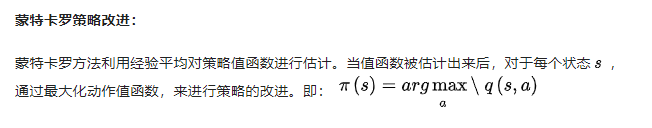
1. **基于值函数的强化学习方法，基于蒙特卡洛的强化学习方法，简述蒙特卡罗估计值函数的算法。**



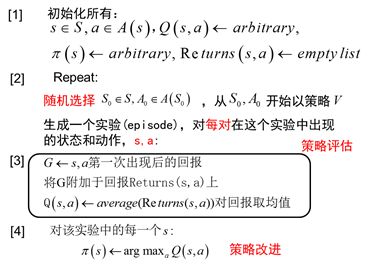
状态值函数和行为值函数的计算实际上是计算返回值的期望。如图3.2，动态规划的方法是利用模型对该期望进行计算。在没有模型时，我们可以采用蒙特卡罗的方法计算该期望，即利用随机样本来估计期望。在计算值函数时，蒙特卡罗方法是利用经验平均代替随机变量的期望。此处，我们要理解两个词，何为经验？何为平均。



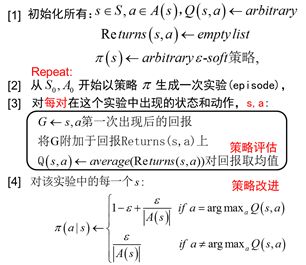




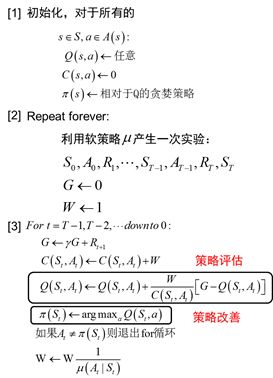
递增计算平均的方法：



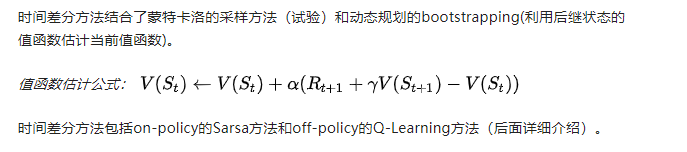
同策略蒙特卡罗强化学习



异策略每次访问蒙特卡罗算法的伪代码：



1. **简述时间差分算法**

****

时间差分方法结合了蒙特卡洛的采样方法（试验）和动态规划的bootstrapping(利用后继状态的值函数估计当前值函数)。

蒙特卡洛方法：利用所有回报的累积估计值函数（值函数最原始的定义）。

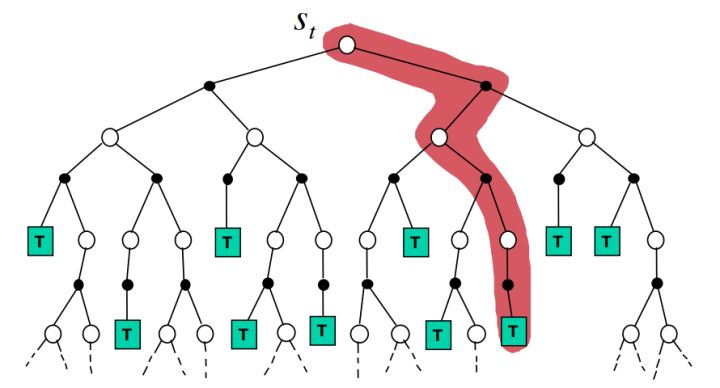
动态规划和时间差分：用一步预测方法计算当前状态值函数。

不同：1.动态规划方法利用模型得到后继状态；2.时间差分方法利用实验得到后继状态。

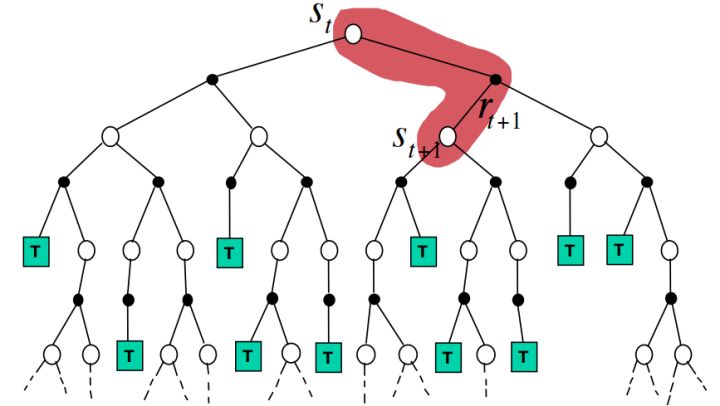
1. **蒙特卡洛和时间差分的对比：MC和TD分别是无偏估计吗，为什么？MC、TD谁的方差大，为什么？**

前者无偏后者不是，前者方差大，从其值函数更新公式来谈；

MC：

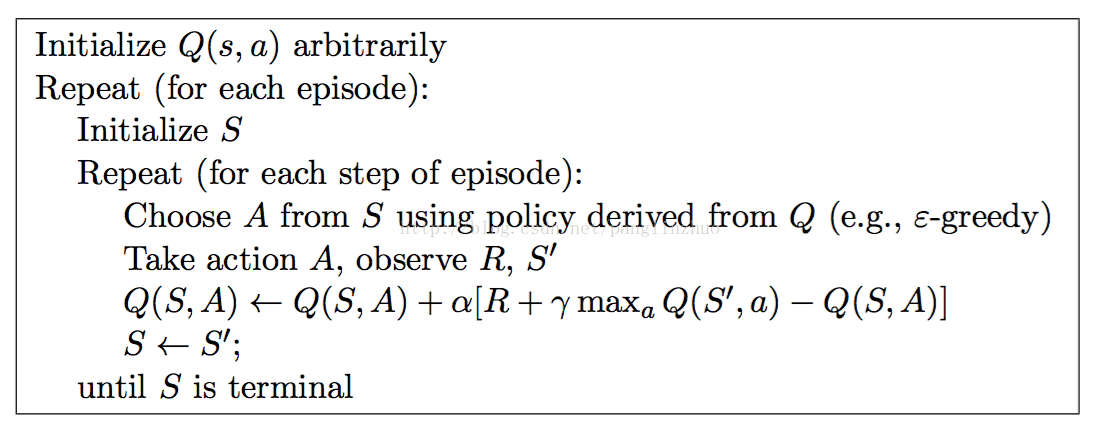


TD（0）：

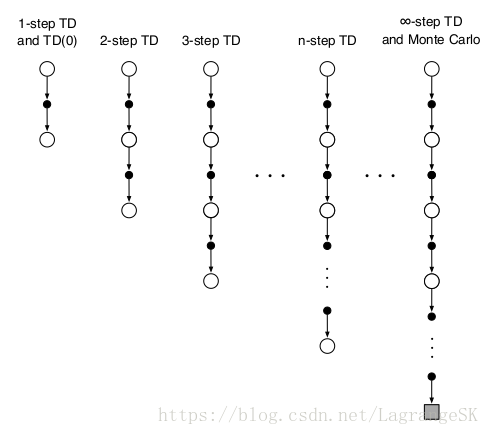


* MC不能用于continuous task（比如倒立摆）， 而TD(0)可以：理由同上。
* 相反，MC能用于一些围棋之类的规划较深的任务.
* MC必须在整轮episode迭代结束后进行更新(也就是一盘围棋要下完才更新一次)，而TD(0)在下一个状态s(t+1)后就可以进行更新

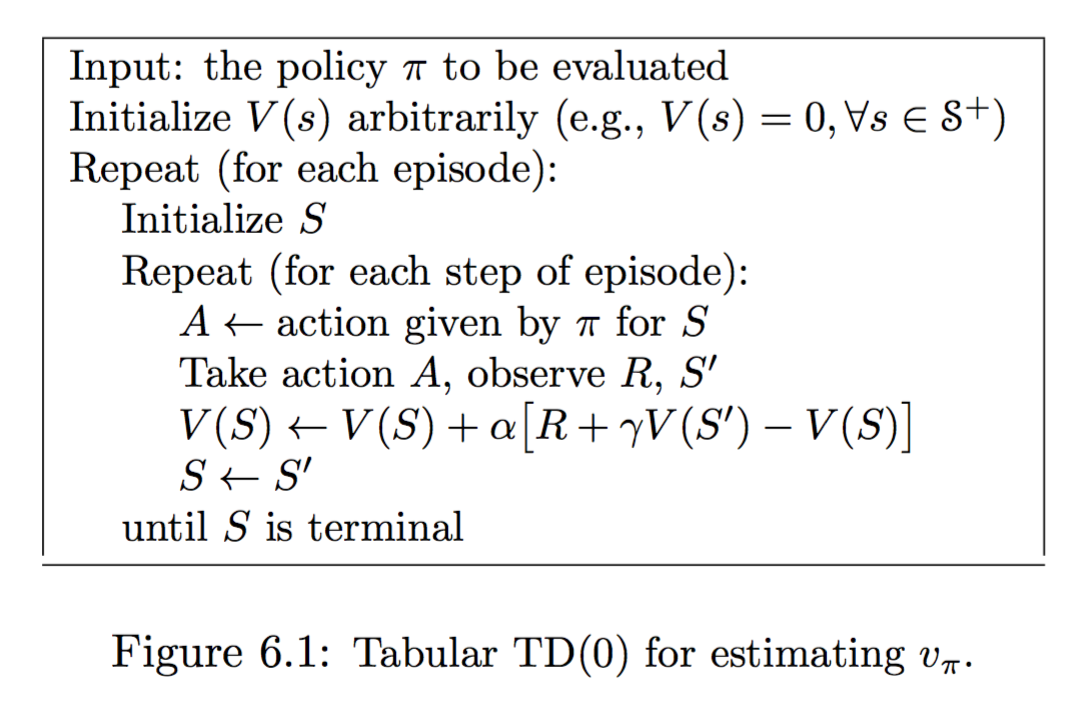
1. **简述Q-Learning，写出其Q(s,a)更新公式。它是on-policy还是off-policy，为什么？**



1. **写出用第n步的值函数更新当前值函数的公式（1-step，2-step，n-step的意思）。当n的取值变大时，期望和方差分别变大、变小？**

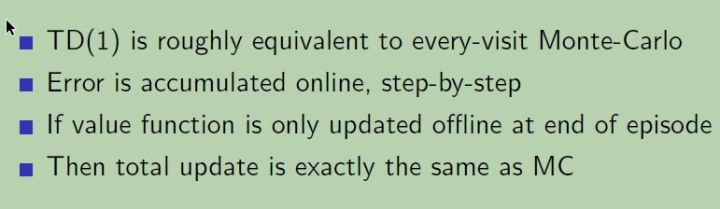


N变大，偏差减小，方差变大

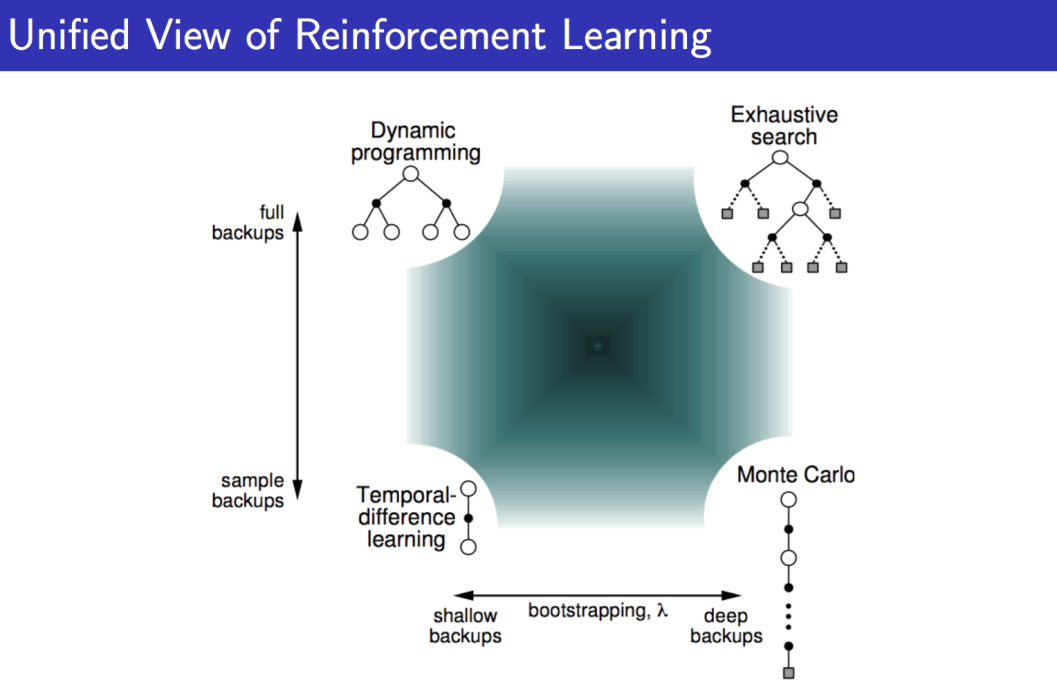


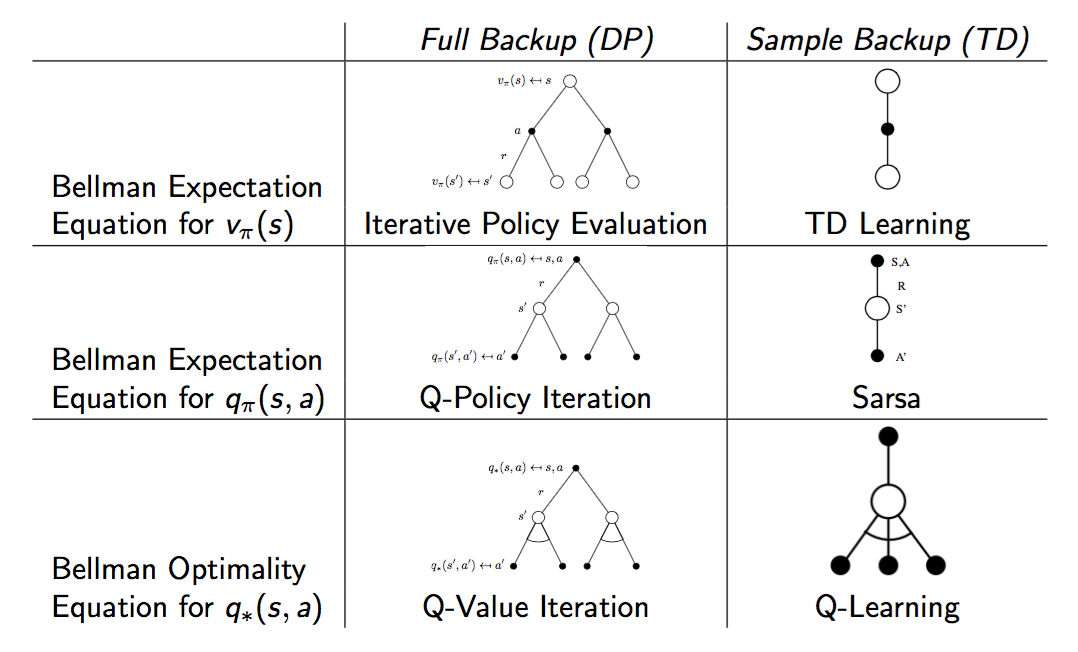
1. **TD（λ）方法：当λ=0时实际上与哪种方法等价，λ=1呢？**

当λ=1时，TD(1)粗略看与每次访问的MC算法等同；在线更新时，状态价值差每一步都会有积累；离线更新时，TD(1)等同于MC算法。

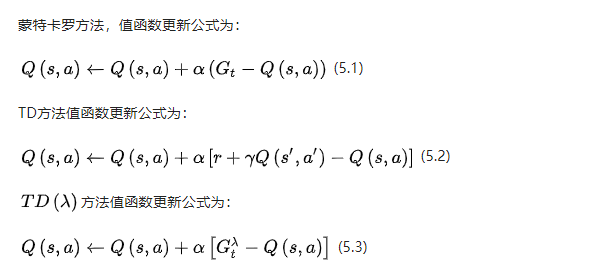


TD算法在这里就不同，它是每过一个time step就更新一次，利用奖励 R\_{t+1} 和值函数 V( S\_{t+1} ) ，当然，这里所说的one-step TD 方法，也可以两步一更新，三步一更新…..是不是等到N步以后再更新就是蒙特卡洛算法了？对，就是这么一回事。



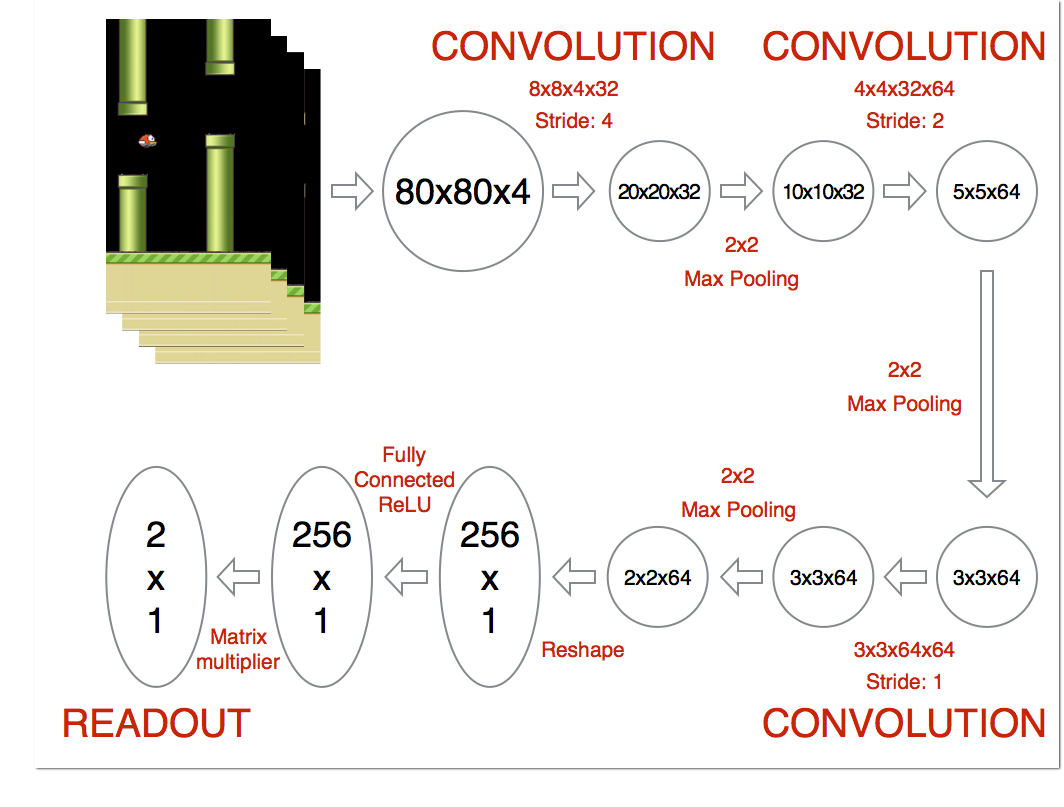


1. **写出蒙特卡洛、TD和TD（λ）这三种方法更新值函数的公式**



1. **DQN玩Flappy Bird的流程图。在这个游戏中，状态是什么，状态是怎么转移的？奖赏函数如何设计，有没有奖赏延迟问题？**

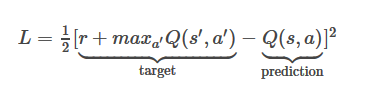
Deep Q network其实就是使用神经网络来估计Q函数。神经网络的输入时图片，输出是采取各个行为所能获得的Q函数。



在flappy bird的例子中，输入时连续四帧图片。输出READOUT用来表示在这情况下，是否“跳”所能获得的奖赏。

在学习的时候会对图片做个预处理，就是简单的阈值操作

那么我们要学习的目标就是：



这里Q(s′,a′)也是神经网络的预测值。我们要优化的目标是当前神经网络对于s状态的估计值于通过后向传递获得的估计值的残差。那么，训练时候获得整个运动序列的作用仅仅只是想要知道s状态所能转移的下一个状态s′，也就是当前图片显示的flappy bird在按了一下跳或者不跳后，图片会变成什么样子。

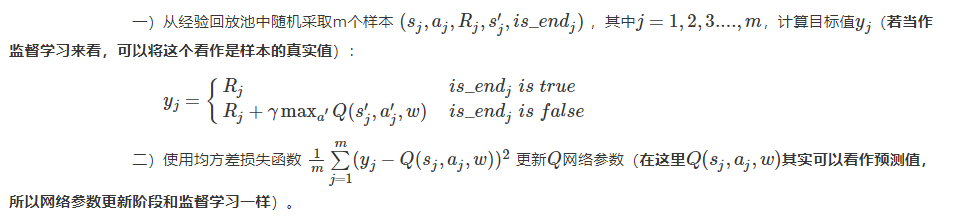
那么这里就涉及到选择策略：探索（exploration）和利用（exploitation）。

* 深度Q学习： 用cnn来估计Q函数。要学习的目标是：当前状态的估计Q函数和下一个状态反向获得的奖赏的误差最小。
* 策略梯度：奖赏值是从结束状态往回传递。要学习的目标是：当前估计的行为的概率和实际采取行为的误差乘以反向奖赏值.

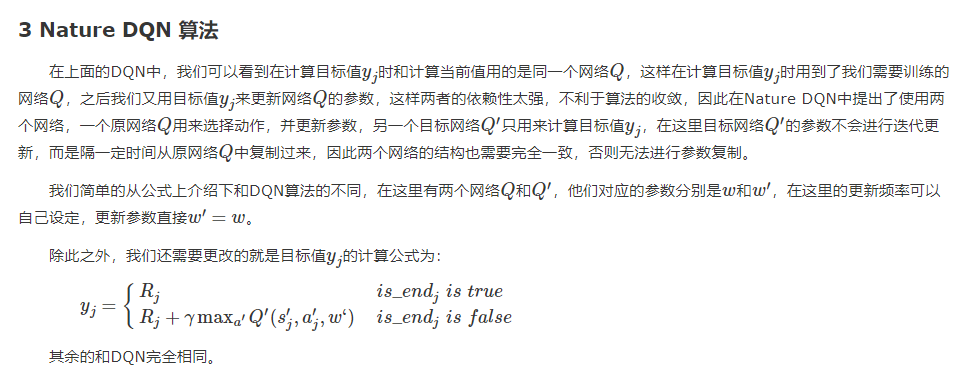
延迟奖励（delayed reward）是强化学习最困难的问题之一，因为（a）这需要做很多探索性工作，（b）你面临一个困难的信度赋值（credit assignment）问题。例如，如果你想象一个变体，比如你最喜欢的视频游戏的一个变体，其中你只在游戏结束时获得 -1/+1 的奖励来判断输赢，这使得你的学习变得异常困难，特别是游戏如果有 1000 万帧或 10 亿帧的时候。

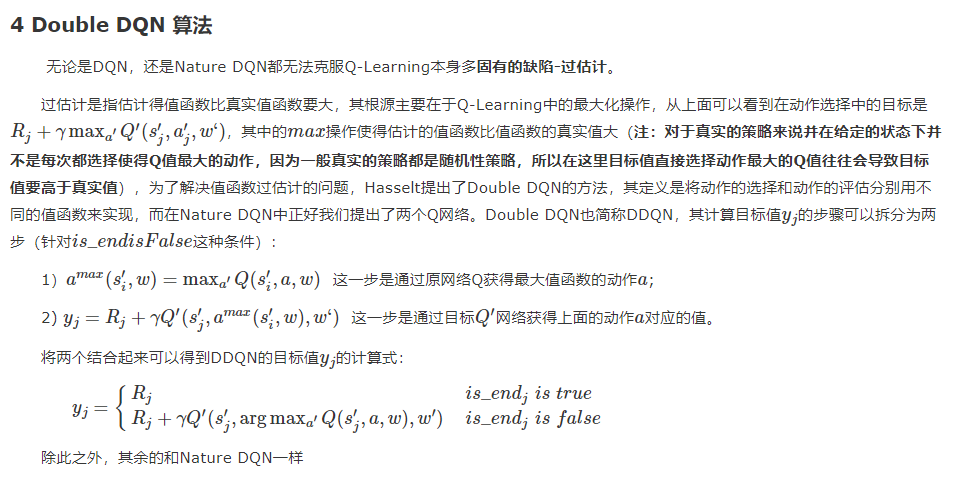
1. **DQN都有哪些变种？引入状态奖励的是哪种？**

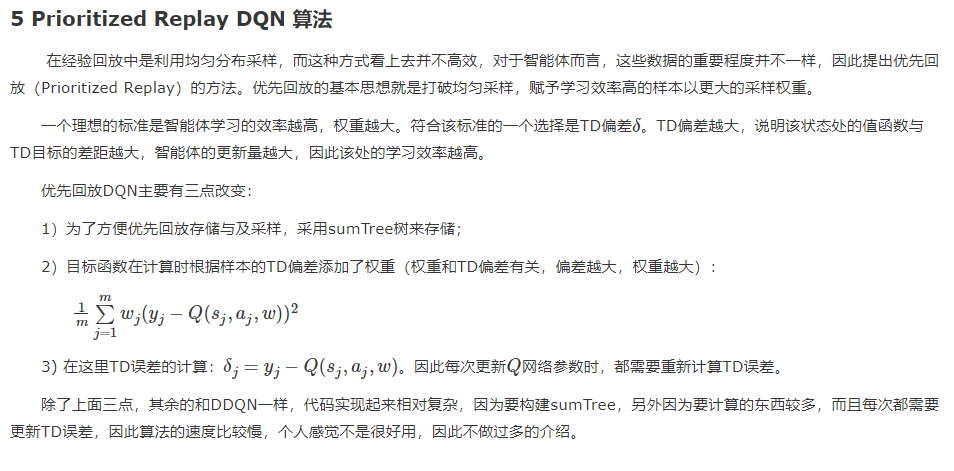
* DQN：



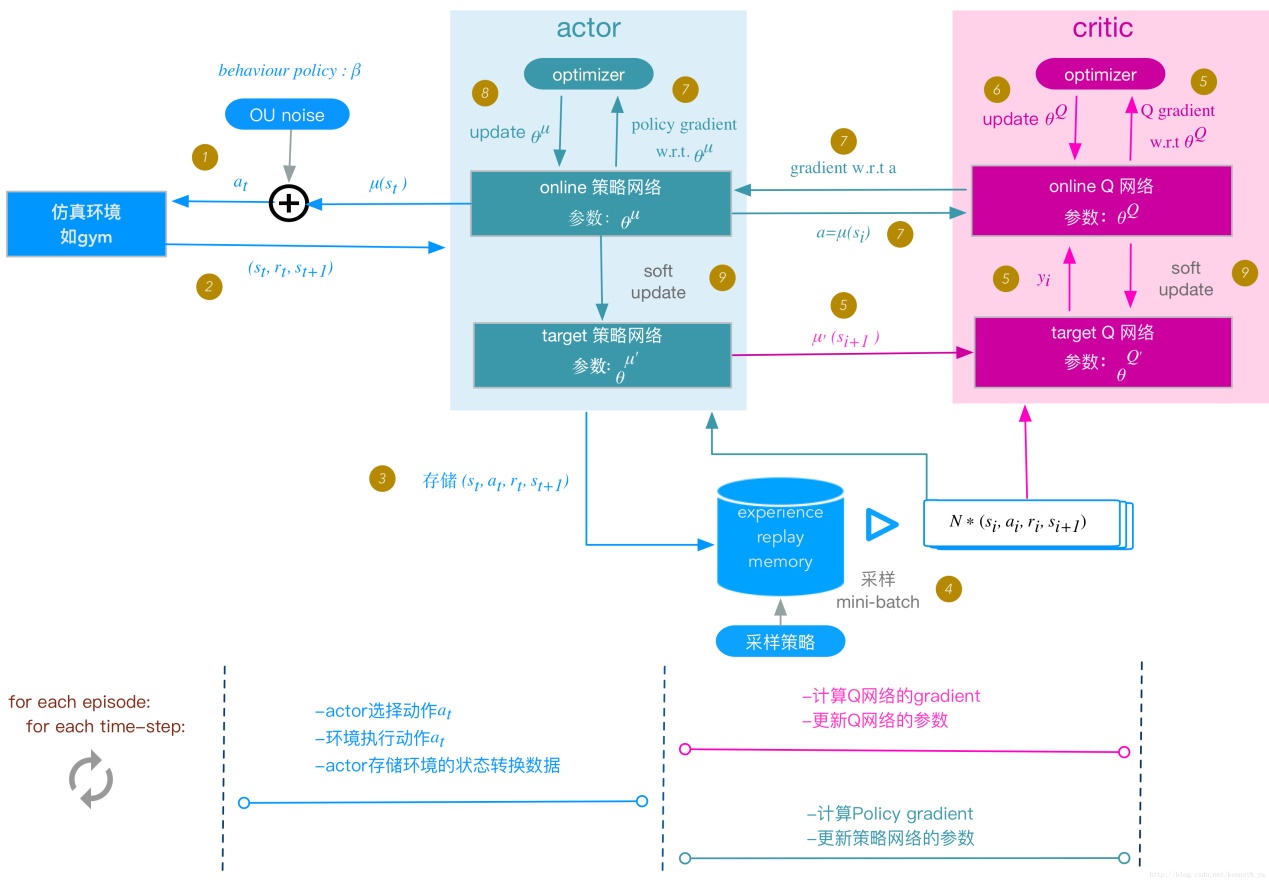
* Nature DQN:



* Double DQN:  
  



1. **画出DDPG框架。DDPG是on-policy还是off-policy，为什么？**



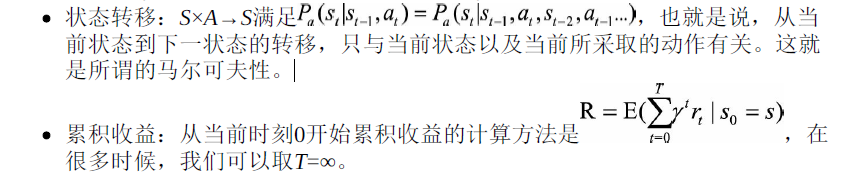
1. **actor-critic框架中的critic起了什么作用？**

Actor-Critic从名字上看包括两部分，演员(Actor)和评价者(Critic)。其中Actor使用我们上一节讲到的策略函数，负责生成动作(Action)并和环境交互。而Critic使用我们之前讲到了的价值函数，负责评估Actor的表现，并指导Actor下一阶段的动作。

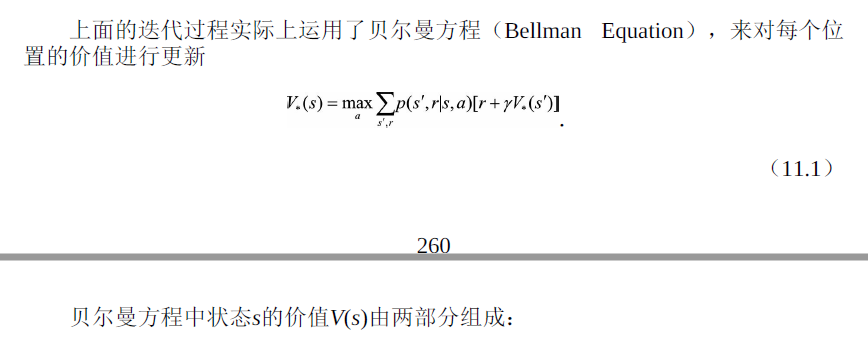
1. **简述A3C算法。A3C是on-policy还是off-policy，为什么？**

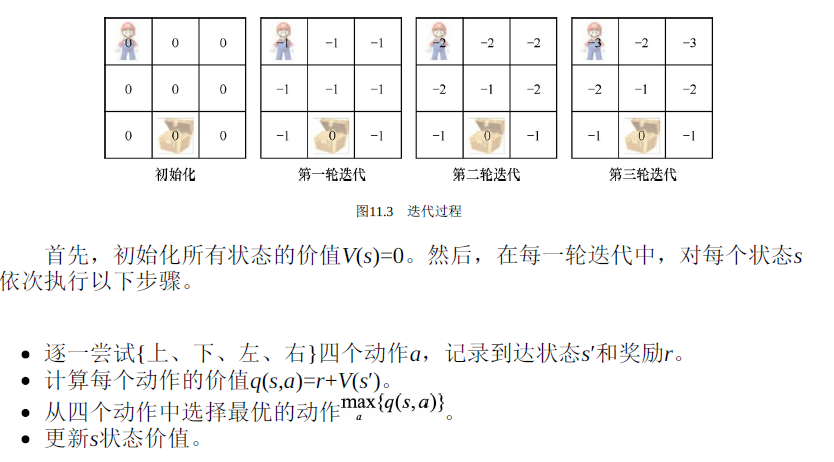
A3C算法，使用了多线程的方式，一个主线程负责更新Actor和Critic的参数，多个辅线程负责分别和环境交互，得到梯度更新值，汇总更新主线程的参数。而所有的辅线程会定期从主线程更新网络参数。这些辅线程起到了类似DQN中经验回放的作用，但是效果更好

1. **强化学习概念**

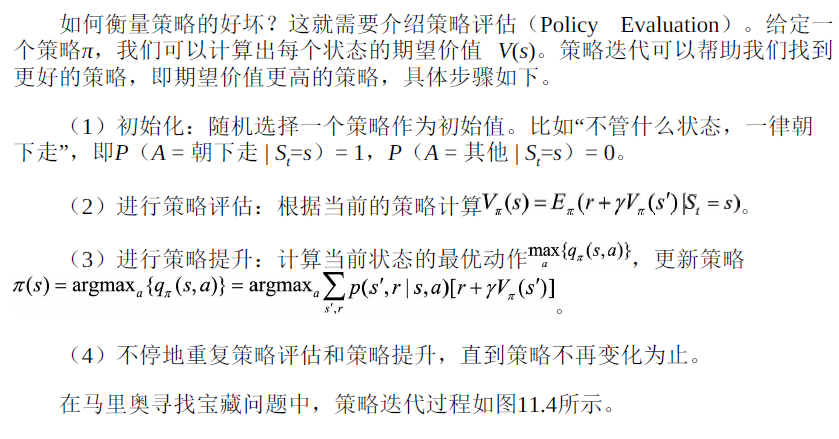


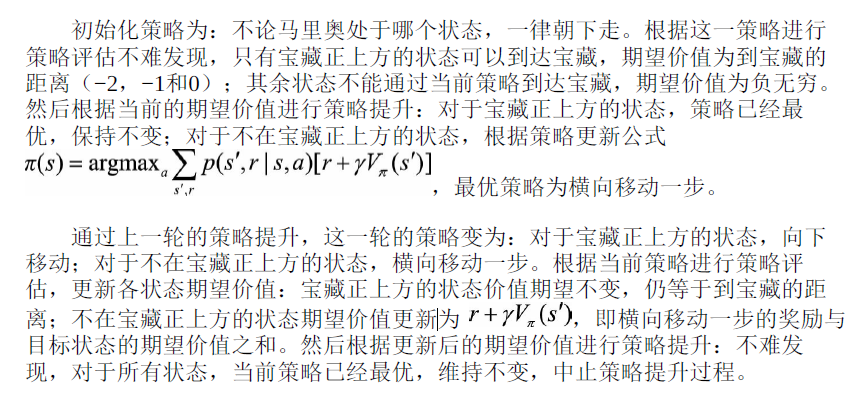
* **价值迭代**

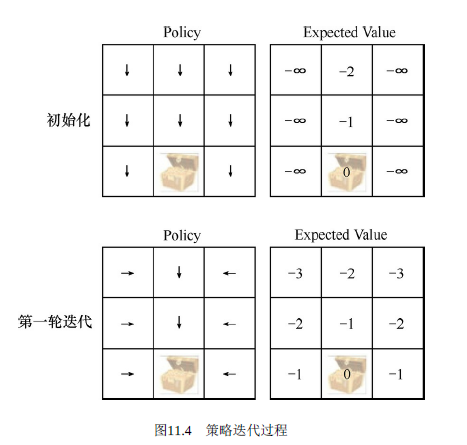




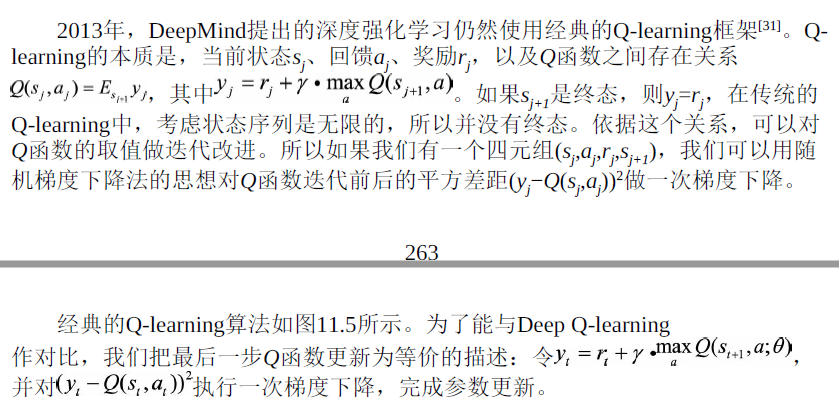
* **策略迭代**

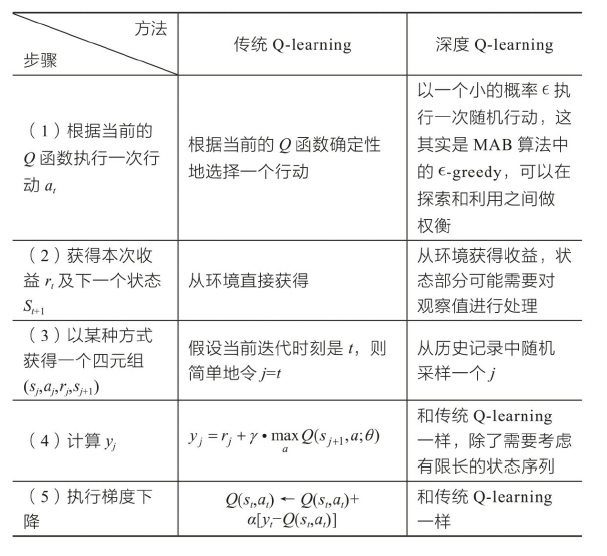
****

****

****

* **DQN：**

****

****

1. **什么是策略梯度，它和传统Q-learning有什么不同，相对于Q-learning来说有**

**什么优势?**

和Q-learning不同的是，策略梯度并不估算Q函数本身，而是利用当前状态直接生成动作a\_t。这有效避免了在连续状态空间上最大化Q函数的困难。同时，直接用梯度的方法优化R(θ)可以保证至少是局部收敛的。