强化学习综述

目录

第一章：引言

1.1 简介

强化学习是一种学习做什么或者说如何将当前的状态映射到做出什么动作才能使自己的累计获得的奖励最大化的学习。学习目标是使 agent 得到的累计奖励最大，为了达到其目标，agent 不需要被告知怎么做，而是必须通过自己不断尝试试错(trial and error)来发现哪些行为能获得最大的累积奖励。在强化学

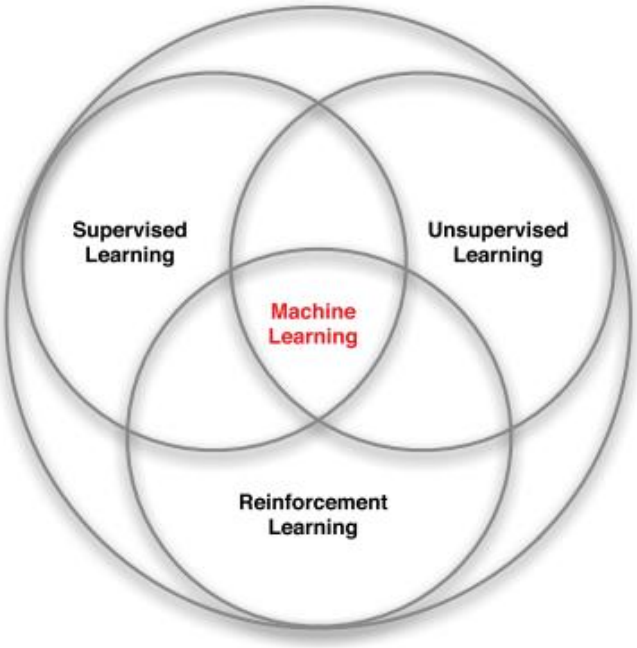
习中，最有挑战的问题是：在某一状态下，agent 所选择做出的 action，不仅仅会影响到当前的奖励（即时奖励），更会影响到后继所有的动作和奖励。所以说，强化学习中的不断试错和延时奖励是其最重要也是最有区分性的特征。我们运用动力学系统理论的思想，将强化学习问题形式化，具体来说就是，非完全信息状态下的马尔科夫据测过程的最优控制问题。强化学习中的 agent 必须要能在一定程度感知到其所处环境的状态信息并且有能力做出行为（action）去影响所在的环境状态。除此之外，agent 必须要有一个或多个明确的、与环境相关的目标。任何能解决这一类问题的方法，都认为是强化学习方法。

1.2 与其他机器学习方法的区别

监督学习是从有标记的训练数据来推导出的机器学习任务方法。 而强化学习不需要标记数据，数据来源是智能体与环境的交互得到，没有外界的知识信息。

无监督学习是从输入的大量的无标记数据中对这些数据数据分类，找出数据中隐藏的结构。强化学习虽不需要标记的数据样本，但也不属于无监督学习，强化学习是通过智能体与环境交互中的不断试错，找出获得累计奖励最大的方法，而不是找出数据中隐藏的结构。

因此，强化学习可以看做是除了有监督学习，无监督学习外的第三种机器学习范式，三者之间的联系如下图所示。



概括一下，强化学习与监督学习和无监督学习主要的区别有以下几点

1、 没有标签和专家信息，只有奖励函数

2、 数据是序列化的，其数据之间是有关联的，不是独立同分布的

3、 Agent 执行当前的动作会对后续的数据有所影响

1.3 基本概念

策略（policy）：策略定义了智能体在给定时间的行为方式。策略是一种映射，从环境的感知状态映射到在这些状态下要采取的行动。它对应于心理学中所谓的一套刺激反应规则或联想。在某些情况下，策略可能是一个简单的函数或查找表，而在其他情况下，它可能涉及大量的计算，如搜索过程。策略从某种意义上来说是强化学习问题的核心，因为有了策略就足够决定智能体的行为。策略可以是随机的，指定每个动作的概率。

奖励信号（reward）定义了强化学习问题的目标。在每个时间步（time step）上，环境向强化学习智能体发送一个数字，称为奖励。智能体的唯一目标是在长

期内获得最大的累计回报。在一个生物系统中，我们可能会认为奖励类似于快乐或痛苦的经历。它们是智能体所面临的问题的直接和定义特征。奖励信号是改变策略的主要依据;如果策略所选择的操作之后的奖励很低，那么策略可能会在以后的情况下被更改为选择其他操作。一般来说，奖励信号可以是环境状态和所采取行动的随机函数。

价值函数（value function）：指示从长远角度来看，智能体的哪些状态或什么状态下采取的什么行为对智能体本身的有利程度。一个状态的价值是智能体从这个状态开始，未来累积的奖励总和。奖励决定了环境状态的直接、内在的可取性，而价值则表明，在考虑到可能遵循的状态和这些状态中的奖励之后，这些状态的长期可取性。

模型（model）：用来模仿环境行为，或者说，用来对环境行为进行推断。例如，给定一个状态和动作，该模型可能预测下一个状态和下一个奖励的结果。模型被用于规划，我们指的是在实际经历之前通过考虑可能的未来情况来决定行动方向的任何方式。使用模型和规划来解决强化学习问题的方法称为基于模型的方法，而不是简单的无模型方法，相对地，不使用模型和规划的方法为无模型方法。

1.4 发展历史

强化学习的早期有两条主线，在现代强化学习相互交织之前，这两条主线是各自独立追求的。一种思路是通过不断试错来学习，起源于动物学习心理学。这条线索贯穿了人工智能领域的一些最早的研究，并引发了 20 世纪 80 年代初强化学习的复兴。第二条主线是关于最优控制问题及其使用值函数和动态规划的解

决方案。两条主线大多是独立的,但围绕第三条主线（差分法）在某些程度上相互联系。

“最优控制”一词在 20 世纪 50 年代后期开始使用，解决这个问题的方法之一是由理查德·贝尔曼（Richard Bellman）和其他人在 20 世纪 50 年代中期通过扩展 19 世纪汉密尔顿和雅可比理论而发展起来的。该方法使用动态系统的状态和值函数或“最优返回函数”的概念来定义函数方程，现在通常称为 Bellman 方程。通过求解该方程来解决最优控制问题的方法被称为动态规划（Bellman，1957a）。Bellman（1957b）还引入了称为马尔可夫决策过程（MDP）的最优控制问题的离散随机版本，Ron Howard（1960）设计了 MDP 的策略迭代方法。所有这些都是现代强化学习理论和算法的基本要素。

动态规划被广泛认为是解决一般随机最优控制问题的唯一可行方法。它遭受贝尔曼所谓的“维度的诅咒”，意味着它的计算需求随着状态变量的数量呈指数增长，但它仍然比任何其他通用方法更有效，更广泛适用。自 20 世纪 50 年代后期以来，动态规划得到了广泛的发展，包括对部分可观察的 MDP 的扩展（Lovejoy，1991 年调查），许多应用（White，1985,1988,1993），近似方法（Rust，1996 年调查），以及异步方法（Bertsekas，1982,1983）。

现代强化学习领域的另一个主要思路，即以试错学习的思想为中心。这个主题始于心理学，试错思想在早期的人工智能中，在与其他工程分支不同之前，一些研究人员开始探索试错学习作为工程原理。试验和错误学习的最早的计算研究可能是明斯基和法利克和克拉克在 1954 年。Farley 和 Clark 描述了另一种神经网络学习机，旨在通过反复试验来学习。在 20 世纪 60 年代，术语“强化”和“强化学习”首次被用于工程文献中。特别有影响力的是明斯基的论文“迈向人

工智能的步骤”（Minsky，1961），该论文讨论了与强化学习相关的几个问题。

第三条思路，基于时序差分部分起源于动物学习心理学，特别是在辅助强化学的概念中。辅助强化剂是与主要强化物（例如食物或疼痛）配对的刺激物，因此已经具有类似的增强特性。明斯基（1954）可能是第一个意识到这种心理学原理对人工学习系统很重要的人。亚瑟·塞缪尔（Arthur Samuel，1959）是第一个提出并实施一种包含时间差异思想的学习方法，作为他着名的跳棋运动项目的一部分。1972 年，Klopf 将试错学习与时差学习的重要组成部分结合起来。Klopf 对可扩展到大型系统学习的原则感兴趣，因此对局部强化的概念很感兴趣，因此整个学习系统的子组件可以相互加强。他提出了“广义强化”的概念，即每个组成部分（名义上，每个神经元）都以强化术语来看待所有输入：作为奖励的兴奋性输入和作为惩罚的抑制性输入。这与我们现在所知的时差学习并不是同一个想法，回想起它比 Samuel 的工作更远。另一方面，Klopf 将这一想法与试错学习联系起来，并将其与动物学习心理学的大量经验数据库联系起来。

Sutton（1978a，1978b，1978c）进一步发展了 Klopf 的思想，特别是与动物学习理论的联系，描述了由时间上连续预测的变化驱动的学习规则。他和巴托改进了这些观点，并开发了一种基于时差学习的经典条件心理模型（Sutton 和 Barto，1981a; Barto 和 Sutton，1982）。接下来是基于时间差异学习的其他几种有影响的经典条件心理模型（例如，Klopf，1988; Moore 等，1986; Sutton 和 Barto，1987,1990）。此时开发的一些神经科学模型在时间差异学习方面得到了很好的解释（Hawkins 和 Kandel，1984; Byrne，

Gingrich 和 Baxter，1990; Gelperin，Hopfield 和 Tank，1985; Tesauro，1986; Friston 等。 ，1994)，虽然在大多数情况下没有历史联系。Schultz，Dayan 和 Montague（1997）总结了时间差分学习与神经科学思想之间的联系。

1981 年，sutton 等人开发了一种在试错法学习中使用时差学习的方法，Actor-critic，并将此方法应用于 Michie 和 Chambers 的极点平衡问题（Barto，Sutton 和 Anderson，1983）。这种方法在 Sutton（1984）的博士论文中得到了广泛的研究。论文并扩展到 Anderson（1986）博士论文中使用反向传播神经网络。论文。大约在这个时候，Holland（1986）将时间差异思想明确地纳入了他的分类器系统。sutton 在 1988 年采取了关键步骤，将时间差异学习与控制分开，将其作为一般预测方法。该论文还介绍了 TD（λ）算法并证明了它的一些收敛性。

时序差分思想和最优控制思想在 1989 年完全结合在了在一起，Chris Watkins 开发了 Q-learning。这项工作扩展并整合了强化学习研究的所有三个主题的先前工作。Paul Werbos（1987）通过争论自 1977 年以来试错学习和动态规划的融合，为这种整合做出了贡献。到沃特金斯的工作时期，强化学习研究已经有了巨大的增长，主要是在机器学习子领域，人工智能，还有更广泛的神经网络和人工智能。1992 年，Gerry Tesauro 的游戏项目 TD-Gammon 的成功引起了人们对该领域的更多关注。1994 年，rummery 提出了 sarsa 算法，2014 年，David silver 提出了确定性策略梯度算法—DPG。2015 年以后，由 Google deepmind 将强化学习方法与深度学习结合，开发出 DQN 算法及其各种改进方法，更多人了解到深度强化学习这一领域，深度强化学习被认为是迈向通用人工

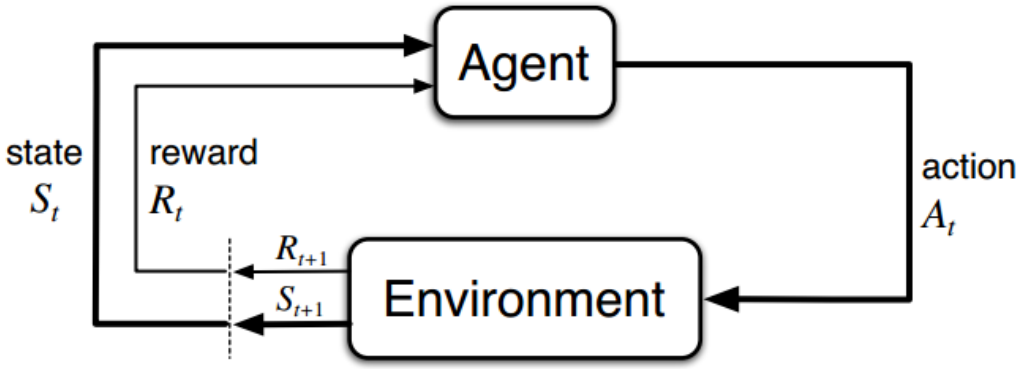
智能（Artificial General Intelligence，AGI）的重要途径。

第二章：马尔可夫决策过程

2.1 强化学习基本框架

MDP 是指从交互中学习以实现目标的问题框架。几乎所有的强化学习问题都能转化成 MDP 形式，MDP 是顺序决策的经典形式化，是强化学习问题的一种数学理想化形式，可以对其进行精确的理论表述。

主动学习和决策的主体被称为 agent。与 agent 交互的对象(包括除 agent 以外的所有信息)称为环境。agent 选择行为，环境对这些行为做出反应并给 agent 呈现出新情况。环境对 agent 行为的反应包括回馈给 agent 一个奖励函数，agent 的目标即为最大化累计收到的奖励函数。下图为 agent 与环境的交互过程。



agent 和环境在一个离散时间序列上相互作用，t=0, 1, 2, 3,…. 在每个时间点 t, agent 收到环境状态信息，并且在此基础上选择动作，在$t+1$ 时间点，agent 收到由所选择的动作带来的代表奖励的数值，并且处于新的状态，agent

与环境交互的过程可由S_0，A_0，R_1，S_1，A_1，R_2，S_2，A_2，R_3 ...序列表示

2.2 收益与价值

2.2.1 收益（return)

Agent 的目标是最大化自己的累计收益,而收益的精确表示可以标记为G_t，定义为从第 t个时间步开始，奖励序列的某种特定函数。G_t最简单的一种情况是 agent 在与环境交互过程中，所有 reward 的累加和，如下式所示

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

其中T是交互过程中的最后一个时间步，S_T为终止状态。但很多情况下，agent 与环境之间的交互是连续的，即没有终止状态，交互序列是无限的，此时为了避免无穷大的回报及数学计算上的便利，G_t往往采用一种有折扣的形式，采用折扣因子$\gamma \in [0,1]$，表示为t时刻之后的每个奖励乘以折扣因子的递增次幂的累计，如下式所示。

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

2.2.2 价值（value function)

价值函数分为状态价值和状态-动作对价值，分别用 v 和 q 表示。状态价值函数是指，agent 在某一策略 π 下，从状态 s 开始获得的累计回报期望

$$\mathrm{v(s)} = \mathbb{E}[G_t | S_t = \mathrm{s}]$$

动作价值函数是指在状态s下，采取某个行为所产生的累计回报期望

2.3 Markov 相关概念

2.3.1 Markov 性质

马尔可夫性质是指，在现有状态信息已知的情况下，过去与将来是独立的，即当且仅当，满足$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1，\ \ldots，\ S_t]$时，称一个状态具有马尔科夫性质。具有马尔科夫性质指，我们需要了解对 agent 过去的历史信息只通过上一时刻的 state 就能全部知道。即 agent 做决策，当前的状态信息包含了 agent 所有过去的信息，不需了解别的信息，一旦了解现在所处的状态信息，那么之前的那些信息都可以被抛弃。

2.3.2 Markov process

马尔可夫序列也称马尔可夫链，是一个具有马尔可夫性质的，随机的状态序列。形式为一个两元组，$< \mathrm{S}, \mathrm{P} >$,即一个状态空间和一个状态转移概率矩阵。状态空间指，该马尔可夫序列中，包含的所有状态信息（有限个状态）。状态转移概率是指，在马尔可夫序列中，相邻两个状态之间的转移概率。状态转移概率矩阵即状态空间中的有先后顺序的两两状态之间，所有的状态转移概率组成的矩阵。

2.3.3Markov reward process

Markov reward process 是指带有价值判断的马尔可夫序列。其形式为一个四元组$< S, P, R, \gamma >$，其中γ为折扣因子,R是奖励函数，R_s为状态 s 下，所获得奖励的期望 $R_s = \mathbb{E}[R_{t+1}|S_t = s]$

在已知状态转移概率，奖励函数的情况下，可计算出价值。

$$v(s) = \mathbb{E}[G_t|S_t = s]$$

展开G_t，对 value function 递归分解即得

$$v(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \middle| S_t = s\right]$$

将R_{t+1}后的γ提出，即得

$$v(s) = \mathbb{E}\left[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots)\middle| S_t = s\right]$$

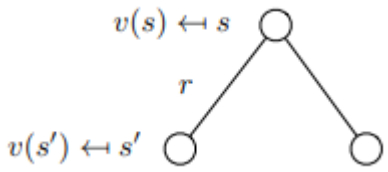
可写为

$$v(s) = \mathbb{E}[R_{t+1} + \gamma G_{t+1})|S_t = s]$$

$$v(s) = \mathbb{E}[R_{t+1} + \gamma \, \mathrm{v}(S_{t+1})|S_t = s]$$

上式称为贝尔曼期望方程

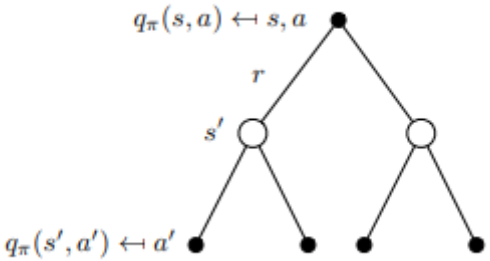
若 s 后续状态 s′ 的概率及其价值已知，也可以根据概率分布求得



$$v(s) = \sum_{s' \in S} P_{ss'} \, v(s')$$

同理，对于状态动作价值函数也有：

$$q_\pi(\mathrm{s}, \mathrm{a}) = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$



$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}\, \pi(a'|s')\, q_\pi(s',a')$$

2.3.4 MDP

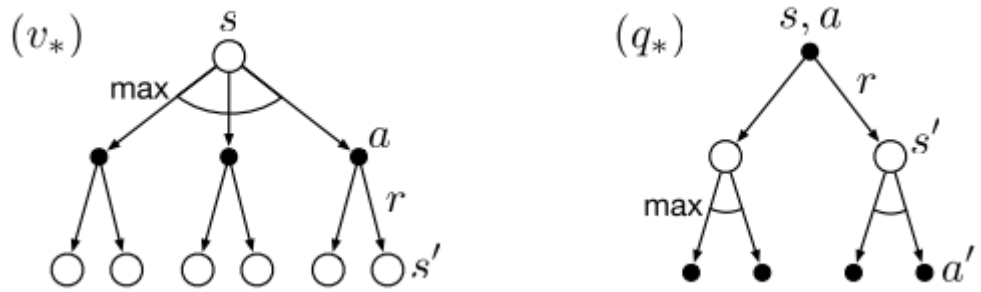
MDP 相比于 MRP 加入了动作集合，是一个五元组，$< \mathrm{S, A, P, R, \gamma} >$，其中的状态转移概率为$P_{ss'}^a = \mathbb{P}[S_{t+1} = \mathrm{s}'|S_t = \mathrm{s}, A_t = \mathrm{a}]$，奖励函数为$R_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$，指的是在状态 s 下，采取某个行为的奖励的期望。总是可以把 MDP 分解为一个 MP 和 MRP。

2.4 最优值函数与最优策略

解决强化学习问题就意味着找到一种能够在长期内获得最多回报的策略。对于有限的 MDP，如果策略π的预期收益大于或等于所有状态π′ 的预期收益，则该策略π被定义为优于或等于策略π′ 。换言之，当且仅当，$V_\pi(\mathrm{s}) \geq V_{\pi'}(\mathrm{s})$，对于状态集中所有状态都成立时，称 $\pi \geq \pi'$，即策略π优于π′。当$V_*(\mathrm{s}) = max_\pi\, V_\pi(s)$，或$q_*(\mathrm{s,a}) = max_\pi q_\pi(s,a)$时，称 π 是最优策略，记做$\pi^*$。

强化学习问题中，总存在至少一个的最优策略

下图表示了V_*和q_*的 Bellman 最优方程中考虑的未来状态和可选的动作范围



对应的贝尔曼最优方程

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big].
\end{aligned}
$$

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\
&= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big].
\end{aligned}
$$

一旦了解了V_*，就相对容易找到最优策略，所有有非零概率到达V_*状态的策略，都被认为是最优策略，或称为 greedy 策略。了解V_*信息后，在任意 state，agent 还需做一步单步搜索，找到下一步可到达状态中价值最大的状态，直到终止状态，即得最优策略。

如果了解q_*的全部信息，则更容易寻找最优策略，agent 在每个状态，只需

寻找该状态下所有状态动作价值中最大的一个即可，按照该动作指示，可得到每步应该采取的动作集。

第三章：动态规划

3.1 简介

3.1.1 动态规划

动态规划是一种将复杂问题分解为子问题，通过解决各个子问题从而解决整个问题的思想。其中的“动态”指整个问题的子问题序列或者规划指优化问题的方法。解决动态规划问题需要（1）解决各个子问题（2）将子问题的解合并起来。

能够使用动态规划方法解决的问题通常具有以下属性：

（1） 最优子结构

全局最优可以分解为局部最优，即，整个问题的最优解也是问题所有子问题最优解的集合。

（2） 重复子问题

指子问题会重复出现并且其解决方案可以存储下来，重复利用，即会重复用子问题的解

3.1.2 动态规划在求解 MDP 问题的应用

而上章提到的马尔科夫问题中，状态价值的贝尔曼期望方程

$$v_\pi(s) = \sum_{a\in A} \pi(a|s)\,(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a,\, v_\pi(s'))$$

可以看出对于状态集中的所有状态，$v_\pi(s)$与其所有后继的$v_\pi(s')$是有线性关系的。等号右边的其他量均为已知常量（动态规划假设了解所有 MDP 的信息，包括状态转移概率矩阵及奖励函数），所以当$v_\pi(s')$最优时，$v_\pi(s)$也为最优，满足动态规划需要的第一个条件:最优子结构。而且 value 函数的存在，记录了 MDP 中每个 state 的价值，可以重复利用，满足第二个条件：重复子问题。满足这两个条件，说明 MDP 问题可以利用动态规划求解。

3.1.3 解决 MDP 问题的一般步骤

解决 MDP 问题，即找到使得 agent 的累计回报期望最大的最优策略。求解过程通常分为两步

（1） 预测(prediction)：对于给定的策略，评估在此策略下，各个状态的状态价值，即 Vπ(s)

Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and policy π
or: MRP $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$
Output: value function v_π

（2） 控制(control)：根据策略的评估结果（Vπ(s)）及确定的 MDP 信息，优化策略，得到最优策略，即 V_*，q_*

Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
Output: optimal value function v_*
and: optimal policy π_*

3.2 策略评估（prediction）

评估方法：迭代使用贝尔曼期望方程

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)\,(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a\, v_\pi(s'))$$

在第 k+1 次迭代的过程中，使用第 k 次的结果（$v_\pi(s')$），去评估当前状态的价值 V(s)，策略评估的伪代码如下图所示

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

3.3 control

3.3.1 策略迭代（policy iteration)

通过策略评估的结果,对策略Π进行优化,每评估一次,就可以优化一次策略,直至达到最优策略。优化的方法是贪心算法（greedily)，就是在邻接的状态中选择$R_{ss'}^a$+γ$v(s')$最大的，即 $\pi' = greedy(v_\pi), \pi'(s) = argmax_a\, q_\pi(s,a)$,
且策略改进是在每次策略评估之后开始

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

策略迭代及策略评估的伪代码如下

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

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

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

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

关于伪代码的说明：

（1）Θ的作用：

因为往往不需要等到评估收敛至一个固定值，策略就已经迭代到最优了，所以引入一个停止机制（stopping condition）：值函数的ϵ-convergency（ϵ代表了评估的精度）当相邻两次的值函数改变幅度小于ϵ时，认为值函数已经收敛，停止评估。

（2）通过最后的 policy stable 可以看出 iteration 是紧跟着 evaluation 的，即做完评估，就迭代一次策略函数，若对于任一 s∈ S，都有

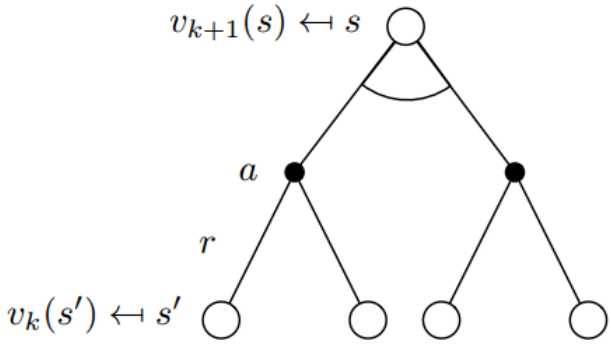
$$q_\pi\big(s, \pi'(s)\big) = max_{a\in A}\, q_\pi(s, a) = q_\pi\big(s, \pi(s)\big) = v_\pi(s)$$

此时，满足贝尔曼最优方程，π 收敛为最优策略

$$v_\pi(s) = max_{a\in A}\, q_\pi(s, a)$$

3.3.2 价值迭代（value iteration）

价值迭代的思想是，在了解子问题的最优解的情况下，V_*可以从后往前推出，通过迭代使用贝尔曼最优方程，寻找 $V_*(s)$，直接迭代贝尔曼最优方程直至其收敛



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

第四章:无模型预测(model free-prediction)

4.1 简介

不同与动态规划，无模型预测是指在不了解所有 MDP 信息的情况下（缺失状态转移概率和奖励函数），直接从 agent 与环境过去的交互中学习，从经验中评估各个状态的价值，继而不断改善策略。

4.2 蒙特卡洛方法

蒙特卡洛方法指，从 agent 与环境交互的完整序列（i.e. 包含终止状态）中采样，直接通过求采样序列的价值平均值，评估状态的好坏，通过大数定理，可以知道，当采样的数据足够多时，通过计算平均值求出来的价值函数几乎等同于真正的价值函数。

蒙特卡洛方法的目标：

从经验序列中采样，学习策略π下的价值函数

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

其中，G_t是指单个序列包括折扣的期望总和

$$v_\pi(\mathrm{s}) = \mathbb{E}[G_\mathrm{T} | S_\mathrm{t} = \mathrm{s}]$$

蒙特卡洛评估值函数一般有两种方法：初访蒙特卡洛方法和每访蒙特卡洛方法。

（1）初访蒙特卡洛法

计算采样的序列中第一次到达状态 s 的价值的期望，伪代码如下。

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$

条件：之前没有出现过，St是在此episode中第一次出现

（2）每访蒙特卡罗法

不论该状态是不是第一次出现，直接对序列中的状态的价值求平均，总收益

/出现总次数。

4.2 时序差分方法

4.2.1 简介

时序差分方法（Temporal-Difference Learning）可以说是强化学习的核心和创新之处。其思想结合了蒙特卡洛方法(Monte carlo)与动态规划方法(DP)。和蒙特卡洛方法类似，时序差分方法也是一种无需了解环境中的信息（如转移概率，奖励函数等），直接从 agent 与环境的交互经验中学习，评估给定 policy 下状态空间中各个状态的 value function 的方法（model free）。与 DP 方法的思想类似，TD 通过自己的预测更新 value，无需等到整个决策完成。

4.2.2 与蒙特卡洛方法的联系

时序差分方法与蒙特卡洛方法都是利用样本估计价值函数的方法，称为样本更新(sample updates)。因为它们都涉及到展望(look ahead)一个样本的待评估状态的后继状态,使用后继状态的价值函数和奖励来更新待评估状态的价值。

4.2.3 与蒙特卡洛方法的区别

（1） online

时序差分方法更新 v(s)是 online 的，不需要等到序列终止，在序列执行

的过程中即可更新各个 v(s)。

（2） 利用了马尔科夫性质

时序差分法的 TD target 使用了贝尔曼期望方程，说明其利用了马尔科夫性质，即只依赖 state 获取环境中的信息，或者说 state 包含了环境中所有有用的信息。

（3） 效率更高

而蒙特卡洛方法没有利用马尔科夫的性质，如果在非马尔科夫环境中（不仅仅依赖 state 的信息时，只能得到部分信息）会更加有效。

（4） 属于有偏估计

时序差分法的特点是：低方差，高偏差。时序差分法因为根据后续状态预测价值更新，所以属于有偏估计，但因其只用到了一步随即状态和动作，所以 TD target 的随机性较小，方差也小。而蒙特卡洛方法因为计算价值函数完全符合价值函数的定义，即$V_{\pi(s)}=\mathbb{E}(G_t|S_t=\mathrm{s})$，所以属于无偏估计。但是，$G_t$值要等到序列终止才能求出，这个过程中会经历很多随机的状态和动作，随机性大，所以方差很大。

4.2.2 时序差分法的具体流程

时序差分法借鉴了递增计算平均值的蒙特卡洛思想，只不过，蒙特卡洛是真实的样本序列奖励，而时序差分法是使用预测的价值函数代替真实样本序列。如图 2 与图 1 的对比。

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]$$

图 1 递增求平均的蒙特卡洛方法

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

图 2 时序差分方法

图 2 中的$R_{t+1} + \gamma V(S_{t+1})$称为 TD 目标(target)，$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$

称为 TD 偏差(error)。

时序差分算法的伪代码如下：

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

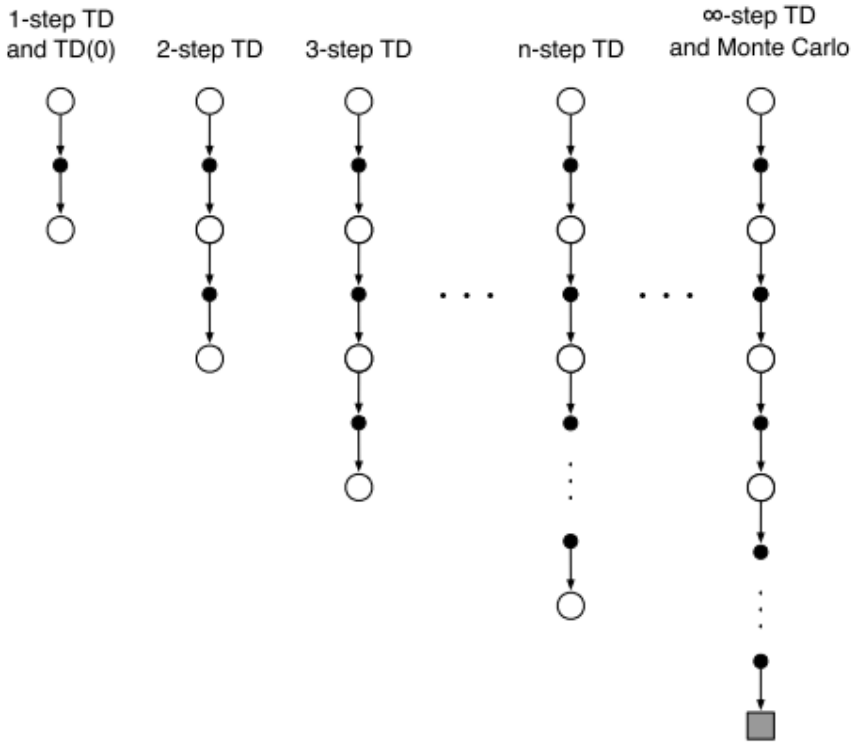
三种方法 prediction 方面的总结：

Model-based/model free	Method name	Advantages	Disadvantages	Bootstrap or not	Sampling or not	Update

Model-based	DP	无偏差，精确的评估价值，总能收敛到最优策略	计算量大，不适用于大规模 MDP 问题，迭代到收敛需要的次数多，	bootstrap	No sampling	Online
Model free	MC	属于无偏差估计，对初始值不敏感，简单易用	高方差，效率低，必须用有终止状态的序列训练至收敛	No	Full width sampling	Not online
Model free	TD	效率更高，无方差	有偏差，对初始值较为敏感	Bootstrap	(1-n step) sampling	Online

4.3 n-step TD 方法

蒙特卡洛方法和 TD 方法在实践中往往都不是最好的方法，这里介绍的 n-step 方法的一个极端是蒙特卡洛方法，另一个极端是单步 TD 方法 (TD(0))。一般实践中最好的方法是 n-step 方法中的中间值。



公式：$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)], 0 \leq t < T$

其中：$G_{t:t+n}$=$R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$

$G_{t:t+n}$中，有 n 个 reward 是 real world 中的，最后一个 V 是预测，像这样迭代，总会收敛到真实的 V

算法伪代码：

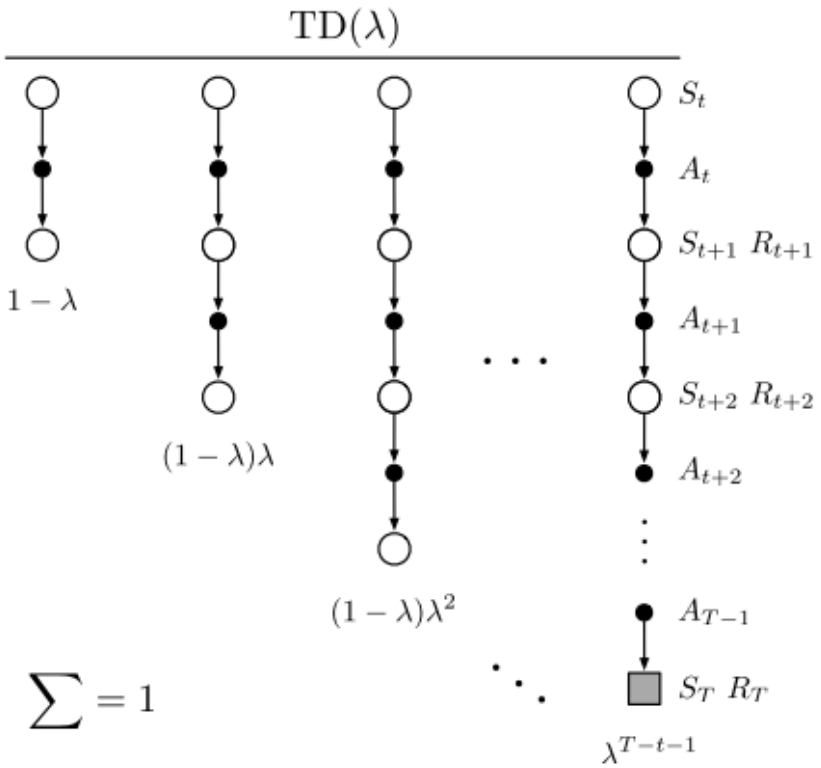
n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T - 1$

4.4 n-step TD 算法的优化：Eligibility Traces TD(λ)

n-step 方法并没有给出能求得使方差和错误率都最小的 n 的方法，所以，介绍一种利用算术加权融合 n 个估计值的方法，称为 TD(λ)，算法思想是，对 n=1 到 n 所有 n-step 得到的结果通过乘相应的权重相加，得到综合考量所有 n 值的方法。

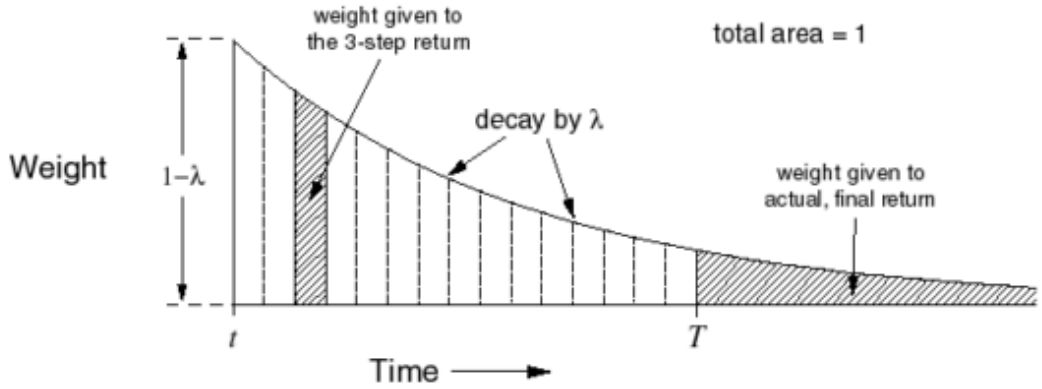


其中λ ∈ [0,1]，为痕迹衰退参数，用来计算权重

λ为 1 时，TD(λ)算法为蒙特卡洛方法，λ为 0 时，TD(0)为单步时序差分方法。每项的$G_t(n)$前乘以权重，权重和为 1。

此时的 TD target 为

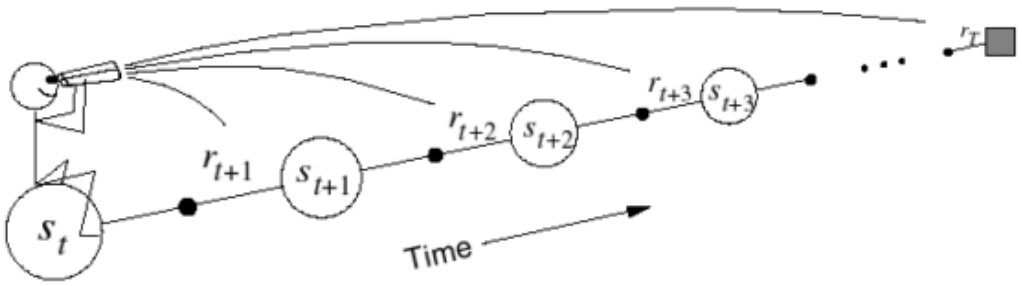
$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$$



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

前向 TD(λ):通过观看将来状态的价值函数来估计当前的价值函数

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t^{\lambda} - V(S_t)\right)$$



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

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

前向 TD(λ)由于根据后来所有状态的价值函数估计当前状态的价值函数，所以需要等到序列完成之后才能更新。

第五章：无模型控制

5.1 简介

在上一章中，介绍了在未知 MDP 条件下，评估 agent 在环境各个状态下的价值函数，这一章将主要介绍，如何在无模型环境中最优化策略，使价值函数达到最大。

控制的目标：得到最优化价值函数

控制的方法：

1、 On-policy 学习：

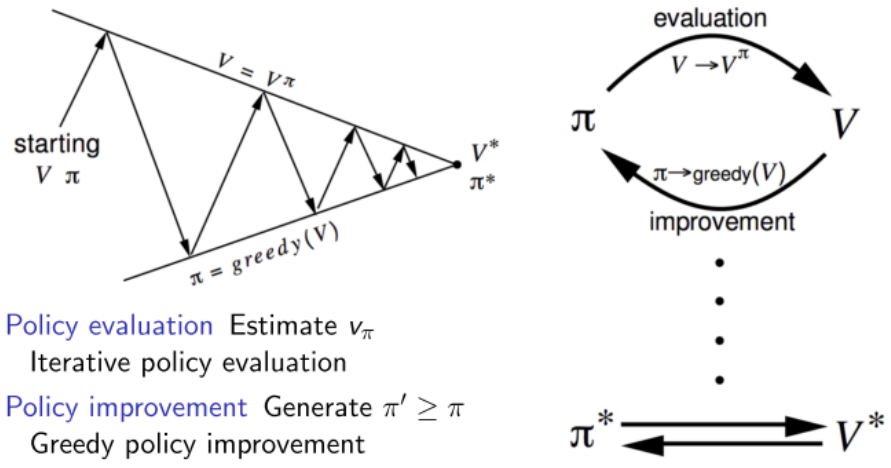
在学习当前策略的过程中，只使用当前策略下产生的样本数据作为经验学习

2、 Off-policy 学习：

在学习某策略的过程中，“站在巨人的肩膀上”，使用别的样本数据作为经验学习，其中样本数据可能来源于专家数据，来源于人类，其他 agent 等等

二者区别是，更新值函数时是否只使用当前策略所产生的样本

在无模型情况下的控制(control)方面，动态规划，蒙特卡洛方法和时序差分方法都使用了 GPI 框架（generalized policy iteration ）或者 GPI 的变种形式



动态规划中的 GPI 框架

5.2 on policy 蒙特卡洛控制

之前已经介绍过，在已知 MDP 的情况下使用动态规划方法的策略迭代算法，即先交替进行策略评估、策略改善，其中遵循贪婪策略的策略改善为：

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

状态价值函数的 Look ahead

但未知 MDP 的情况下，我们不知道状态转移概率，不知道某个状态的所有后续状态，也就无法在某个状态下可以采取哪些行为，若只有评估得到的一堆状态价值函数的集合{ V(s) | s∈S}，从中无法推出具体策略，更谈不上去优化策略

所以我们使用行为价值函数来代替状态价值函数：

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q(s, a)$$

评估的问题解决了，用动作状态价值函数 q 代替状态价值函数 v，但在策略改进方面仍有一定欠缺。如果采用 greedy 方法，考虑到 exploration，无法保证贪婪策略下的策略是最优策略，因此为了平衡 exploitation 和 exploration 的问题，采用ϵ-greedy 方法。

ϵ-greedy 策略：

ϵ-greedy 方法就是有ϵ的概率在动作集中随机选择，1- ϵ的概率从 evaluate 后的动作中，选择价值最大的那个。

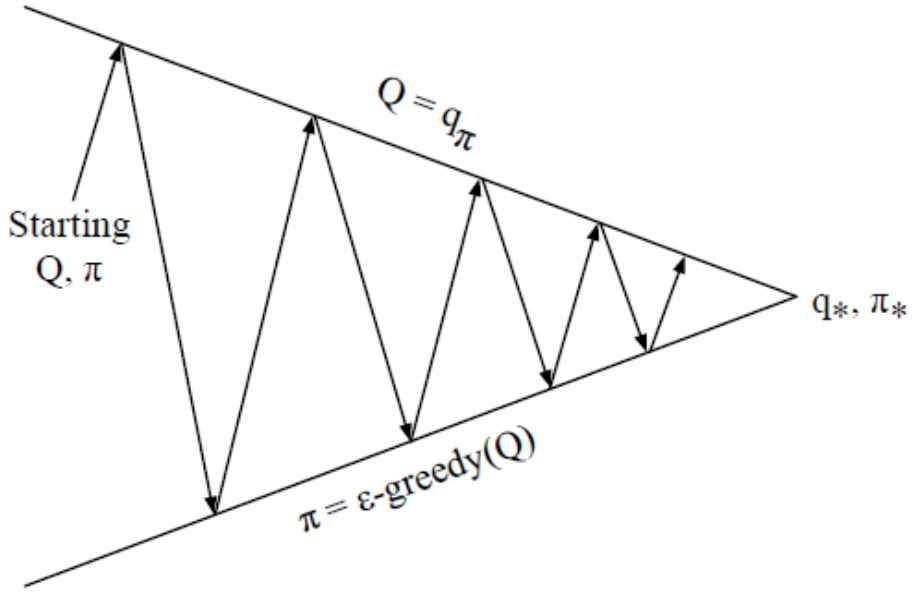
$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

m 个动作下，ϵ-greedy 方法在 state s 下的动作选择

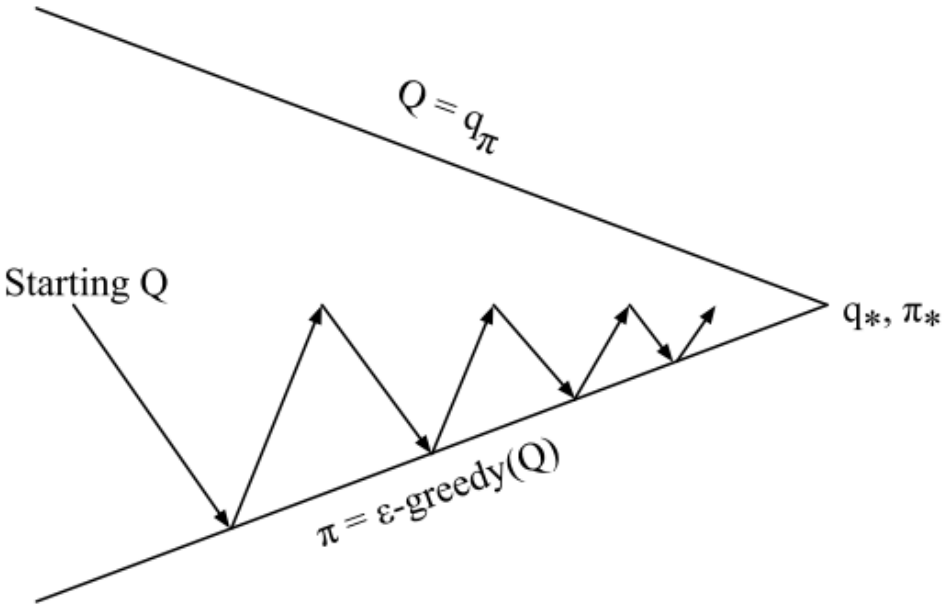
可以证明，使用了ϵ-greedy 方法的策略π′，最终收敛的价值函数比贪婪算法高。证明过程如下图所示

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_{a \in \mathcal{A}} \pi'(a|s) q_\pi(s,a) \\ &= \epsilon/m \sum_{a \in \mathcal{A}} q_\pi(s,a) + (1-\epsilon) \max_{a \in \mathcal{A}} q_\pi(s,a) \\ &\geq \epsilon/m \sum_{a \in \mathcal{A}} q_\pi(s,a) + (1-\epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \epsilon/m}{1-\epsilon} q_\pi(s,a) \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a) = v_\pi(s) \end{aligned}$$

此时的 on-policy 蒙特卡洛控制过程如下图所示



在策略评估阶段，理论上需要经历无限次回合，并对回合进行采样求平均，才能收敛到 Q=qπ，但实际上不需要等到蒙特卡洛法收敛到精准的 q 函数，我们可以每经历一个回合，就进行策略评估，并且改进策略，算法仍会收敛到最优 q 函数和最优策略



On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

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

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

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

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

On-policy 蒙特卡洛方法的伪代码描述

GILE(Greedy in the Limit with Infinite Exploration)方法

当我们已经通过上述方法得到了一个相对不错的策略，不想再探索很多次，但为了探索到环境中所有的可能性，ϵ-greedy 方法仍需要进行探索，GILE 方法就有效的平衡了这一问题，解决了如何在有限次数的继续探索中探索所有的可能。GILE 方法能适应所有的控制模式，只要模式符合以下两种条件

1、 可以探索到所有可能的情况(状态，动作对)，确保没有遗漏任何一种可能

$$\lim_{k\to\infty} N_k(s,a) = \infty$$

2、 策略最终会收敛到贪婪策略上

$$\lim_{k\to\infty} \pi_k(a|s) = 1(a = argmax_{a' \in A} Q_k(s,a'))$$

GILE 方法的核心：

将ϵ-greedy 方法中的ϵ逐渐减小至 0，例如：令ϵk=1/k,随着探索次数 k 的增大，选择随机行为探索的可能性也就越来越小，直至不再探索随机行为，此时，策略 π 也收敛至最优策略。

GILE 蒙特卡洛控制：

首先对使用策略 π 的实验进行采样，产生实验序列$\{S_1, A_1, R_2, \ldots S_T\}$

对于实验中经历的每个状态行为对，采用递增求平均的方法，计算更新每个状态行为对的价值函数 $Q(S_t, A_t)$

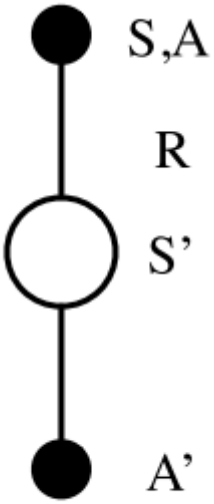
添加 GLIE 方法就是为了保证收集到的数据能够最终收敛到平均价值函数

$$\epsilon \leftarrow 1/k$$
$$\pi \leftarrow \epsilon\text{-greedy}(Q)$$

5.3 Sarsa

Sarsa 算法与 on-policy 蒙特卡洛控制方法的区别就是使用 TD 方法估计 Q 值，即，将 GPI 框架中 evaluation 的方法换成 TD 方法，improve 的方法仍为ϵ-greedy。Sarsa 算法相比于蒙特卡洛控制有几点优点：

1、online　2 、不需要完整序列 3、提高了 improve policy 的频率：每走一个 time step，就改善一次策略更新。



Sarsa 算法从某一特定 state，action 开始，获得相应 reward 之后，根据实际获得的 reward 与下一状态，动作价值函数的估计值（TD target）与当前动作状态价值函数值之差更新。如上图所示，两个状态动作对连接起来刚好是“SARSA”，也正是 sarsa 算法的名称来历。

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma Q(S',A') - Q(S,A)\right)$$

Sarsa 估计动作状态价值函数的过程

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

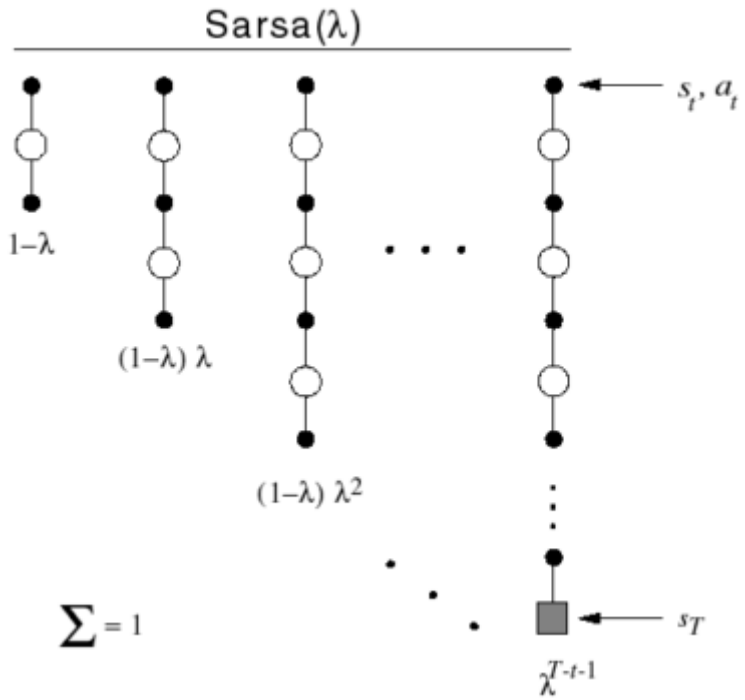
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Sarsa 算法的伪码描述

5.4 Sarsa(λ)

类似的，在评估状态动作价值函数时，不仅仅可以向前走一步，也可以综合考虑 n-step 的 TD 来更新状态动作价值函数。可以参考上章的 TD(λ)方法，将 evaluate 的方法从 TD(0)改为 TD(λ)即可，，就得到了 Sarsa(λ)



Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  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)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S,A) \leftarrow E(S,A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha \delta E(s,a)$
      $E(s,a) \leftarrow \gamma \lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Sarsa(λ)的伪码描述

5.5 off policy learning

5.5.1 off policy TD

异策略的 TD 指使用 TD 方法在目标策略 π(a|s)的基础上更新行为价值函数，进而优化行为策略。

$$V(S_t) \leftarrow V(S_t) + \alpha \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1})) - V(S_t) \right)$$

其中，π为目标策略:，μ为行为策略，agent 处在状态S_t中，执行基于策略μ的行为A_t，之后进入新的状态S_{t+1}。异策略 TD 要做的就是，比较借鉴策略和行为策略在状态S_t下产生同样的行为A_t的概率之比。

概率之比代表了向目标策略的学习程度，可以归类于以下三种：

1、比值接近 1，说明两个策略在状态S_t下采取的行为A_t的概率很相似，此次对于状态S_t价值的更新同时得到两个策略支持，S_t的价值函数较大。

2、比值很小，说明借鉴策略 π 选择A_t的机会要小一些，此时为了从借鉴策略学习，认为这一步状态价值函数的更新不太符合借鉴策略，因而S_t的价值函数更新较小。

3、比值大于 1，说明按照借鉴策略，选择A_t的几率大于当前的行为策略，为了学习借鉴策略，S_t价值函数的更新较大

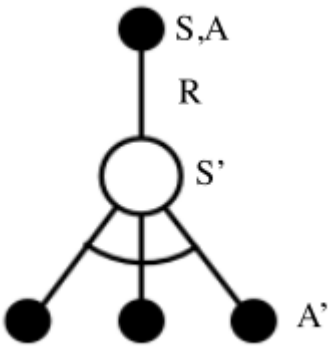
5.5.2 Q-learning

由异策略 TD 算法，我们可以得知，概率比值增大会使价值函数 V（s）增

大。所以不再进行重要性采样，直接取目标策略 π 为 greedy 策略，采用状态-动作价值函数，朝着使下个状态价值最大的方向更新，最终 Q（s,a）会收敛至最优动作价值函数 q*(s,a)。

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

其中的α是学习效率，指的是其中括号内差值的多少会被学习（0<α<1）。



基于当前策略，在当前状态下执行动作，并观察下个状态及下个状态能够采取的动作，在下一个状态中选择动作价值函数最大的那个，更新当前的动作价值函数。

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
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'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

Q-learning 的伪码描述

5.4.3 Double Q-learning

之前介绍的 Q-learning，sarsa 等算法都存在过高估计问题（Maxmization Bias），Q-learning 用最大的价值去更新当前的价值，导致过高的估计值被当做真实估计值，即产生过高估计问题。造成过高估计的原因为：相同的采样序列，既用来决定产生最大化的价值的动作，又用来估计值函数。Double learning 算法就有效地解决了这个问题，将采样的序列分为两组，用来产生两个独立的估计值函数,记为$Q_1(a)$，$Q_2(a)$，一组用来决定使价值最大化的动作行为，$A^*=argmax_a Q_1(a)$。一组用来产生估计值函数$Q_2(A^*)=Q_2(argmax_a\ Q_1(a))$，此时的估计为无偏估计。

将 double learning 的思路与 Q-learning 结合，就得到了 double Q-learning，其算法伪代码为

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

第六章：值函数近似

6.1 简介

6.1.1 强化学习的问题规模

前五章的方法都是基于价值函数更新而去更新策略的（value-based），而之前的方法中价值函数的更新又是基于表格的迭代更新(tabular solution)。而在现实中，状态空间一般都较大，就会产生“维数灾难”的问题。尤其是当状态空间是连续的时候，会有无穷的状态空间。下图是西洋双陆棋，围棋所需要的状态空间。此时，我们无法用之前基于表格的方法为每个状态或者状态动作对都求得一个价值的特定值，因此需要有一种方法去适应不断增加的数据集，能够适应无限的状态集。

Backgammon: 10^{20} states
Computer Go: 10^{170} states
Helicopter: continuous state space

为什么需要 value approximation ：

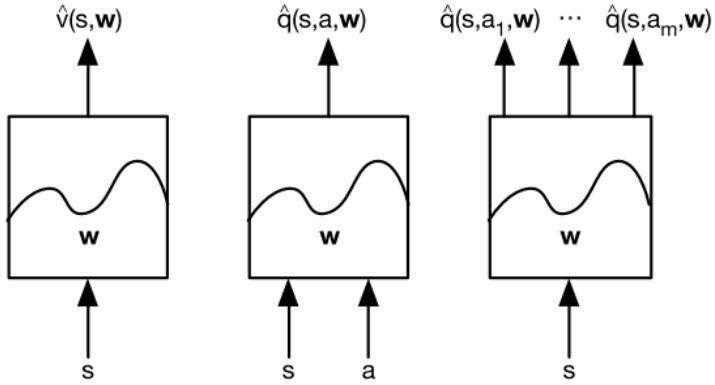
1、表格强化学习方法需要的存储空间过大

2、泛化的需要：假设离某个状态非常接近的另一状态，比如就二者之间的物理位置就相隔了一毫米，没有必要单独存储其价值，因此，需要泛化，泛化到未知的状态中去

6.1.2 解决方案

状态集或状态动作集的价值不再需要表格存储记忆，评估每个状态或状态行为对的价值只需建立一个函数去近似价值函数(V_s或 Q(s,a))。

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$



最右图为普遍方法：输入一个状态的信息，输出该状态下，所有状态动作对的价值函数，直接进行控制

6.2 随机梯度下降方法

随机梯度下降方法与剃度下降法的区别为：每次随机使用单条记录进行更新调整参数的方向。因为没有像梯度下降使用全部的数据集计算 loss function，所以每次迭代的方向并不是使 loss function 下降最快的梯度，但是整体的下降方向是向全局最优解的。这样做带来的好处就是相比于梯度下降综合所有数据集寻找最快的调整方向，随机梯度下降方法要快很多。

以线性回归为例，随机梯度下降算法中的拟合函数为$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$，

，其单条记录的损失函数为$\text{cost}\left(\theta,\left(x^{(i)},y^{(i)}\right)\right)=\frac{1}{2}(h_\theta\left(x^{(i)}\right)-y^{(i)})^2$，所以整体的损失函数就为$J_{train}(\theta)=\frac{1}{m}\sum_{i=1}^{m}cost(\theta,\left(x^{(i)}\right)-y^{(i)}))$，对单条数据的损失函数求导，就得到$(h_\theta\left(x^{(i)}\right)-y^{(i)})x_j^{(i)}$，所以参数更新的公式为：$\theta_j:=\theta_j-\alpha(h_\theta\left(x^{(i)}\right)-y^{(i)})x_j^{(i)}$。具体操作就是：将数据集打乱，然后按照打乱的 m 条记录逐条更新参数。

6.3 线性近似函数方法

近似函数需要选择可导，适用于训练非静态数据，非独立同分布的数据的方法。可导就可用梯度法寻找合适的参数。近似函数可选为线性函数。当近似函数为线性函数时，可使用特征向量与参数向量的线性组合作为近似函数。使用线性函数近似，只有一个最优值，因此可以收敛到全局最优

$$\hat{v}(S,\mathbf{w})=\mathbf{x}(S)^\top\mathbf{w}=\sum_{j=1}^{n}\mathbf{x}_j(S)\mathbf{w}_j$$

其中，x(s)为特征向量，w 为参数向量

$$\mathbf{x}(S)=\begin{pmatrix}\mathbf{x}_1(S)\\ \vdots\\ \mathbf{x}_n(S)\end{pmatrix}$$

定义目标函数为真实的价值函数与近似函数得出的价值函数平方差的期望，为关于参数 w 的二次函数多项式

$$J(\mathbf{w})=\mathbb{E}_\pi\left[(v_\pi(S)-\mathbf{x}(S)^\top\mathbf{w})^2\right]$$

对目标函数求导可得（自变量为参数 w），

$$\Delta \mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)$$

所以，参数更新值=学习率*真实价值函数与近似得到的价值函数的差值乘特征向量。

6.3.1 递增型方法

为了训练近似的值函数，需要将参数向着真实值函数与估计值函数最小的方向更新，类似于监督学习，需要有学习的目标，但强化学习没有标签数据，所以更新目标为真实价值函数的估计函数 $V_\pi(\mathrm{S})$，用蒙特卡洛，TD 等方法的值函数评估方法估值。

当真实值函数评估方法为蒙塔卡罗法时，目标函数为

$$\Delta \mathrm{w} = \alpha(G_t - \hat{v}(S_t, \mathrm{w}))\nabla \hat{v}(S_t, \mathrm{w})$$

当真实值函数评估方法为 TD(0)时，目标函数为

$$\Delta \mathrm{w} = \alpha(R_t + \gamma \hat{v}(S_{t+1}, \mathrm{w}) - \hat{v}(S_t, \mathrm{w}))\nabla \hat{v}(S_t, \mathrm{w})$$

当真实值函数评估方法为 TD(λ)时，目标函数为

$$\Delta \mathrm{w} = \alpha(G_t^\lambda - \hat{v}(S_t, \mathrm{w}))\nabla_w \hat{v}(S_t, \mathrm{w})$$

同理，当价值函数为动作价值函数 q 时，对状态和动作同时提取特征，将 v 换为相应的 q 即可。

6.3.2 batch 方法

增量式的梯度更新方法虽然简单，但是每次更新之后就丢掉了数据，数据利用效率并不高。Batch 方法是指，给定经验数据集 D，从数据集中拟合出最优的近似函数，最小化数据集与近似函数值的差值。

数据集为 agent 与环境交互一段时间之后的记录，包括状态信息和价值

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$$

LS(w)为这段时间内所有时间步的目标价值与近似价值差值平方的和，优化的目的是确定一个最优参数，使得 LS(w)最小

$$\begin{aligned} LS(\mathbf{w}) &= \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2 \\ &= \mathbb{E}_{\mathcal{D}} \left[(v^\pi - \hat{v}(s, \mathbf{w}))^2 \right] \end{aligned}$$

$$\mathbf{w}^\pi = \underset{\mathbf{w}}{\operatorname{argmin}}\ LS(\mathbf{w})$$

（1）经验回放（experience replay）

一种方法为随机梯度下降法结合经验回放的思想，具体做法为：从已知的经验数据集中采样状态和价值，使用随机梯度下降法更新参数 w，$\Delta \mathrm{w} = \alpha\big(v^\pi - \hat{v}(\mathrm{s}, \mathrm{w})\big)\nabla_w \hat{v}(s, w)$，当 w 不再变化时，说明此时参数已收敛至最优

$w^\pi = argmin_w LS(w)$。经验回放可以打破数据之间的关联性

（2）最小二乘法直接运算

反向思考，如果参数为最优，则此时 LS(w)为最小值，则有，根据经验集中

的各个状态，价值采样对应参数更新的期望值为 0，即参数适用于所有采样的结果

直接反解出 w 即得

$$\mathbb{E}_{\mathcal{D}}\left[\Delta \mathbf{w}\right] = 0$$

$$\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)(v_t^{\pi} - \mathbf{x}(s_t)^{\top}\mathbf{w}) = 0$$

$$\sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi} = \sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\mathbf{w}$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi}$$

但次方法时间复杂度较大，为 O（n3）,适合求解特征比较少的问题

6.4 非线性近似函数

6.4.1 DQN

近似函数除了线性函数，还可使用非线性函数神经网络。TD 方法结合非线性的神经网络函数近似可能不会收敛，但 DQN(Deep Q-Networks) 使用经验回放和固定的 Q 目标值能做到收敛而且保持很好的鲁棒性。

Q-learning 算法是 1989 年由 Watkins 提出的，2015 年 nature 论文提到的 DQN 就是在 Q-learning 的基础上修改得到的。

DQN 对 Q-learning 的修改主要在三个方面：

（1） DQN 利用深度卷积神经网络逼近值函数

（2） DQN 利用了经验回放训练强化学习的学习过程

（3） DQN 设置了目标网络来单独处理 TD 方法中的 TD 差值

Q-learning 算法流程：

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
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'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

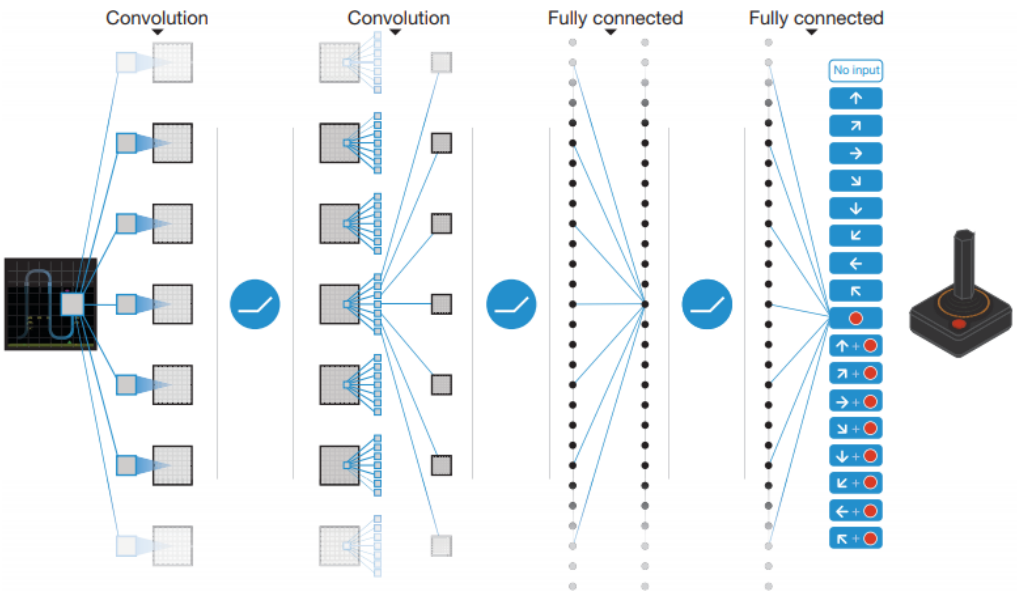
如果对其采用值函数近似方法，则其参数更新公式为：$\theta_{t+1} = \theta_t + \alpha[r + \gamma max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)]\nabla Q(s, a; \theta)$，其中 TD 目标$\mathrm{R} + \gamma max_a Q(s', a; \theta)$中的参数与梯度计算中要逼近的值函数所用的网络参数相同，容易导致数据之间存在关联，使训练不稳定。而 DQN 中，TD 目标的参数θ^-与用于实际更新的参数θ不同，可避免这个问题。DQN 中的参数更新公式：

$$\theta_{t+1} = \theta_t + \alpha[r + \gamma max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)]\nabla Q(s, a; \theta)$$

用于动作值函数逼近的网络每一步都更新，而用于计算 TD 目标的网络则是每个固定的步数更新一次

非线性函数近似方法与上一节所述的线性函数方法，都属于参数逼近的方法。只不过，此处的参数是指神经网络中每层的权重参数，更新值函数其实是更新参数。

DQN 的网络结构由卷积层和全连接层组成。如下图所示



DQN 算法的具体流程如下:

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

首先，初始化回放记忆 D，用随机参数θ初始化动作价值函数 Q，同样的θ值赋给θ^{-}，用θ^{-}初始化 TD 目标 Q 值，从第一个时间步开始，有ε的概率选择随机行为，$1-\varepsilon$的概率选择使 Q 值最大的行为：$a_t = argmax_a Q(\emptyset(s_t), a; \theta)$。在仿真器中执行动作$a_t$，观察奖励 r 及图像。将这一状态转换及动作，奖励记录下来，存储在回放记忆 D 中。从回放记忆 D 中均匀随机采样一个转换样本数据，用

$(\emptyset_t, a_t, r_t, \emptyset_{t+1})$表示，用 TD 目标网络$\theta^-$计算 TD 目标$\mathrm{r} + \gamma max_{a'}\mathrm{Q}(\mathrm{s}', \mathrm{a}'; \theta^-)$，执行动作值函数逼近的参数更新，$\Delta\theta = \alpha[\mathrm{r} + \gamma max_{a'}\mathrm{Q}(\mathrm{s}', \mathrm{a}'; \theta^-) - \mathrm{Q}(\mathrm{s}, \mathrm{a}; \theta)]\nabla\mathrm{Q}(\mathrm{s}, \mathrm{a}; \theta)$，重复上述过程，注意其中每隔 c 步更新一次 TD 目标网络参数，即令$\theta^- = \theta$，直至走完所有时间步。

6.4.2DQN 提升

（1）Double DQN

如第五章中 Double Q-learning 所述的过估计问题一样，由于 DQN 中更新思想与 Q-learning 大致一样，TD 目标均为最大化的动作价值函数，所以会导致过估计问题。Double Q-learning 是将动作的选择与动作的评估分别用不同的价值函数来实现，

Double Q-learning 中的 TD 目标，

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)\,.$$

Double DQN 中的 TD 目标

$$Y_t^{\mathrm{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)\,.$$

（2）prioritized experience replay

上一节所述的 DQN 算法采用了经验回放机制，经验回放时，利用均匀分布随机采样。但随机抽样并不高效，因为 agent 的经验池中所有数据，对 agent 的学习并非具有同等重要的意义。prioritized experience replay 按照一定重要

程度抽样，赋予重要的数据以更大的采样权重，会大大缩短训练的时间。

选择权重的一个方法是根据 TD 偏差（TD error）的大小，TD 偏差越大，说明该状态处的值函数与 TD 目标的差距越大，agent 的更新幅度变大，学习效率就更高。

对于经验池中第 i 条样本，设置它的权重为

$$\mathrm{P(i)} = \frac{P_i^a}{\sum_k p_k^a}$$

其中P_i^a由 TD 偏差决定，TD 偏差越大，P_i^a也就越大，对应的权重就越高。因为采样分布与动作值函数的分布是两个完全不同的分布，为了矫正这个偏差，更新参数时，需要在 TD 偏差前乘一个重要性采样系数

$$\mathcal{W}(\mathrm{i}) = (\frac{1}{N} * \frac{1}{P(i)})^{\beta}$$

算法伪代码如下

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

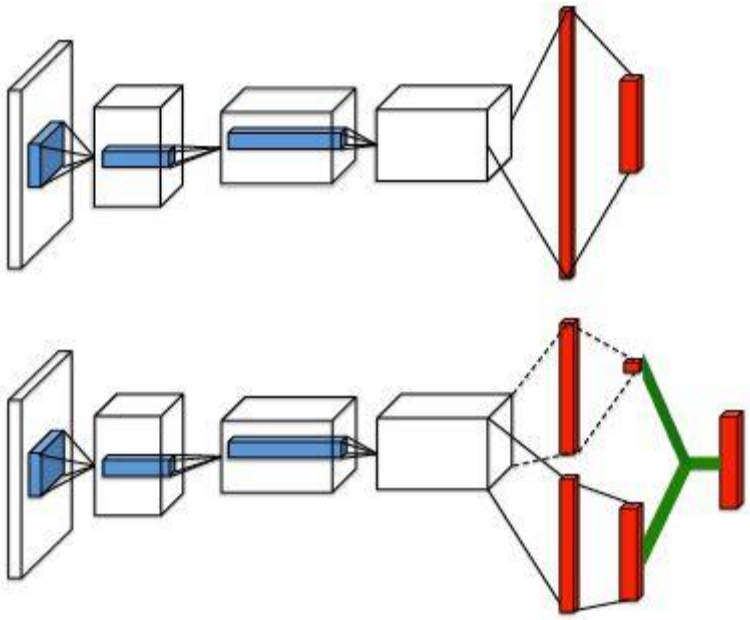
（3）Dueling DQN

相比于 DQN ，Dueling DQN 在网络结构上有所优化，将动作价值函数网络分成两部分：价值函数和优势函数。价值函数仅仅与状态 S 有关，与具体要

采用的动作 A 无关，记做$\mathrm{V}(\mathrm{s},\omega,\alpha)$,优势函数同时与状态状态 S 和动作 A 有关，记为$\mathrm{A}(\mathrm{S},\mathrm{A},\omega,\beta)$,动作价值函数可以表示价值函数与优势函数的组合：

$$\mathrm{Q}(\mathrm{S},\mathrm{A},\omega,\alpha,\beta)=\mathrm{V}(\mathrm{S},\omega,\alpha)+(\mathrm{A}(\mathrm{S},\mathrm{A},\omega,\beta)-\frac{1}{\mathcal{A}}\sum_{a'\in\mathcal{A}}A(S,a',\omega,\beta))$$

ω是公共部分的网络参数，而α是价值函数的网络参数，而β是优势函数的网络参数。下图为 DQN（上）与 Dueling DQN（下）网络结构上的区别。



第七章：策略梯度

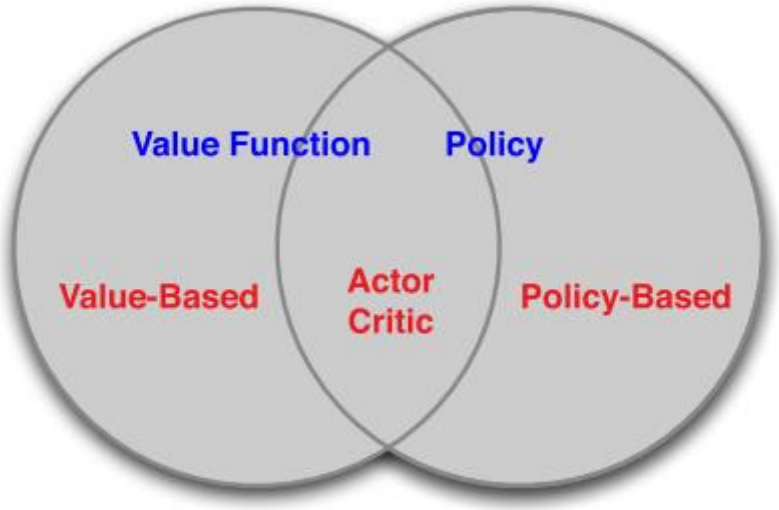
7.1 简介

7.1.1 策略梯度含义

策略梯度：在求解无模型强化学习问题时，不需要对 agent 的各个状态或状态-动作对进行值函数评估，备份，直接参数化策略，进而调整参数使得累计期望回报最大。通常策略梯度函数为以下形式，在某一参数θ下，输入状态，动作

对，输出为该条件下（某一参数、状态）所有可选动作的概率分布。我们通过不断梯度调整参数θ，使得该 agent 的每个状态下做出动作的选择都是最优的，即P[a|s, θ]中，P[a*|a, θ]的概率分布值最大

$$\pi_\theta(s,a) = \mathbb{P}[a \mid s, \theta]$$



7.1.2 与基于值函数强化学习比较

相比于值函数迭代求解最优策略，策略梯度算法有以下优缺点：

1、 有更好的收敛性，值函数迭代求解强化学习问题时，有时候价值函数可能会在真实价值函数之间来回波动，震荡，甚至无法收敛。而使用策略梯度算法会使收敛过程更加稳定，因为只是朝着优化方向渐进的前进，所以随着策略梯度方法调整参数，至少可以得到一个局部最优解

为策略梯度法一直在向好的方向改善策略

2、 策略梯度算法在高维状态空间或连续动作空间的情况下会更有效率，值函数强化学习算法在巨大的动作空间下（如：高维或连续性动作空间下），需要迭代计算很多次，才能大致确定那些状态动作对相对应的价值函数，来进一步优化策略。而策略梯度算法无需迭代多系计算价值函数，只需将参数向最优化策略的方向移动即可。

3、 策略梯度可以学习到随机策略

缺点

1、 简单的策略梯度方法相比于基于值函数的强化学习方法可能收敛速度或更慢，或方差更高，相对来说效率更低，策略梯度只是每次调整一小步，更加稳定，但同时效率也相对低一些

2、 可能会收敛到局部最优解，而非全局最优解

实际中，策略梯度算法应用更普遍，因为其是一种端对端的学习方法，并且策略改进的方向直接为使得累计期望回报的

7.2 策略目标函数

与值函数近似方法中的随机梯度下降法不同，策略梯度方法没有 loss function,策略梯度法通过一步步优化目标函数，实现策略优化。

针对不同的问题类型，可以选择不同的目标函数类型：

1、 start value: 在能够产生完整 Episode 的环境下使用，如果初始状态总是 s1，或初始状态的概率分布已知，目标函数即初始状态的价值函数

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

2、 average value：在连续的环境下，可以使用 average value 目标函数，将所有状态的价值与其分布概率相乘求和。目的是使得每个时间状态下，目标函数值最大，即保证每一步都得到最优回报

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

考虑所有状态的概率分布

3、 average reward per time-step：同样适用于连续环境

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) \mathcal{R}_s^a$$

三种方法的本质都是策略梯度，且梯度的方向都是向最优解方向。唯一的区别就在于用不同状态的不同概率分布做计算。

7.3 策略梯度定理

7.3.1 似然比

假设策略π_θ是可导的且不等于 0，且我们知道策略梯度的梯度$\nabla_\theta \pi_\theta(s,a)$，则似

然比（Likelihood ratios）为

$$\begin{aligned}\nabla_\theta \pi_\theta(s,a) &= \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \\ &= \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)\end{aligned}$$

其中，$\nabla_\theta log\ \pi_\theta(s,a)$为 score function，指似然比对参数θ变化的敏感性，指示参数调整的方向，使其朝着获得更多奖励的方向前进。

7.3.2one step MDPs

考虑一个非常简单的 one step MDP 问题，假设初始状态的分布为 s~d(s)，只经历一步得到一个即时奖励 r=$R_{s,a}$ 就终止，由于是单步 MDP，前述的三种目标函数均可使用，假设使用 average reward per time-step 目标函数：

$$\begin{aligned}J(\theta) &= \mathbb{E}_{\pi_\theta}[r] \\ &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\mathcal{R}_{s,a}\end{aligned}$$

由前面的似然比可知，对目标函数求梯度时得

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)\mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) r]\end{aligned}$$

由上式可知，由于前两项的 d(s)与π_θ(s,a)为常数，所以策略改善的方向其实为 score 函数与 reward 的乘积方向。

7.3.3 Policy Gradient Theorem

将 7.3.2 的单步 MDP 扩展到多步 MDP，将单步 MDP 策略梯度公式中的即时奖励换为长期价值Q^{π}(s,a),即得多步 MDP 策略梯度公式，对于任何可微的策略π_{θ}(s,a)，对于任何策略的任意三种形式之一的目标函数，其梯度都如下式所示：

$$\nabla_{\theta}J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_{\theta} log\, \pi_{\theta}(\mathrm{s},\mathrm{a}) Q^{\pi\theta}(\mathrm{s},\mathrm{a})]$$

7.3.4 蒙特卡洛策略梯度

针对具有完整 Episode 的情况，使用随机梯度上升来更新参数，使用策略梯度公式，对于公式里的期望，通过采样的形式来替代，即使用 t 时刻的收获（return）作为当前策略下行为价值的无偏估计。

算法思想为对每个序列进行抽样，结合随机梯度上升思想与策略梯度公式更新参数。蒙特卡洛策略梯度算法描述：

function REINFORCE
Initialise θ arbitrarily
for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
for $t = 1$ to $T - 1$ **do**
$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
end for
end for
return θ
end function

但是，REINFORCE 存在如下三个问题：

① 存在高方差的问题，由于 agent 在一个序列中会采取很多动作，我们很难说哪个动作对最后结果是有用的

② 收敛速度慢，蒙特卡洛特性，只有到达了终止状态的序列才能被采样，调整参数

③ 只有在能够对序列采样的 episodic 环境下使用

7.3.5 Actor critic 策略梯度

Actor 指演员的意思，Critic 是评论家，顾名思义，这种算法就是通过引入一种评价机制来解决高方差的问题。具体是通过引入一个 critic 来估计行为价值函数

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$

Actor critic 算法中包含两组参数：

（1）Critic：更新动作值函数参数ω

（2）Actor：以 Critic 所指导的方向更新策略参数θ

下图为 Actor-critic 算法描述，δ为 critic 价值函数近似的参数，θ为 actor 更新策略目标函数的参数

function QAC
Initialise s, θ
Sample $a \sim \pi_\theta$
for each step **do**
Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
Sample action $a' \sim \pi_\theta(s', a')$
$\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
$w \leftarrow w + \beta \delta \phi(s, a)$
$a \leftarrow a', s \leftarrow s'$
end for
end function

7.4 策略梯度的优化

为了减小方差，引入 baseline 函数，可以直接从策略梯度中减去 baseline 函数，由于 baseline 函数仅与状态有关，而与行为无关，所以不会改变梯度。因此梯度公式中的期望值没有改变，而方差可以改变

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) B(s)\right] &= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s,a) B(s) \\ &= \sum_{s\in\mathcal{S}} d^{\pi_\theta} B(s) \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(s,a) \\ &= 0\end{aligned}$$

Baseline 函数中一个不错的选择是取状态价值函数$\mathrm{B(s)} = V^{\pi\theta}(s)$，所以可以将策略梯度修改一下，引入 Advantage Function：

$$A^{\pi\theta}(\mathrm{s,a}) = Q^{\pi\theta}(s,a) - V^{\pi\theta}(\mathrm{s})$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_\theta \log \pi_\theta(\mathrm{s,a}) A^{\pi_\theta}(\mathrm{s,a})]$$

在这种情况下，需要两个近似函数也就是两套参数，一套用来近似状态价值函数，一套用来近似行为价值函数，以此来计算 Advantage Function 。然而

在实际操作中，由于 TD 误差δ是 Advantage Function 的无偏估计，即

$$\mathbb{E}_{\pi\theta}[\delta^{\pi_\theta}|\mathrm{s},\mathrm{a}] = A^{\pi\theta}(\mathrm{s},\mathrm{a})$$

所以可以直接用近似 TD 误差来计算策略梯度

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$
$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\delta_v]$$

综上，在更新 Critic 时，可以用 MC, TD(0), 前向 TD(λ), 后向 TD(λ) ，同理在更新 Actor 时，也可以使用以上方法。

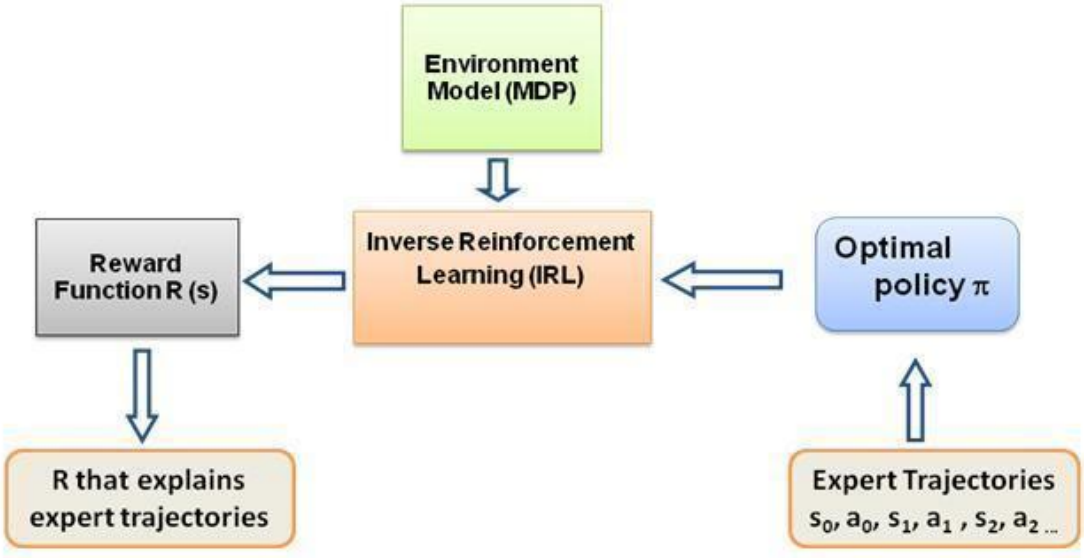
第八章 强化学习前沿

8.1 逆强化学习

8.1.1 简介

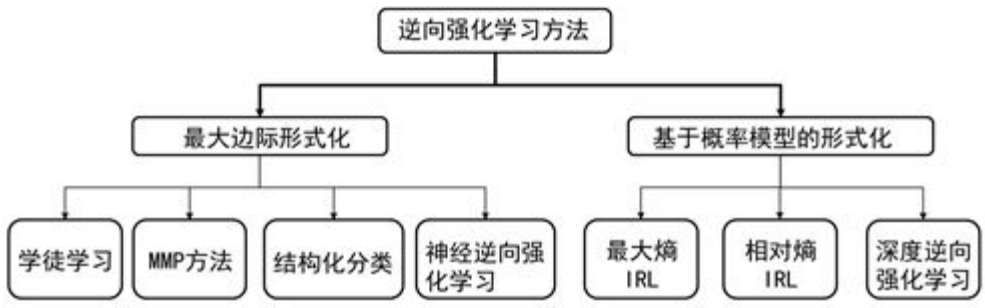
之前的强化学习算法中，奖励大多是人为制定或者是环境给出。然而，在很多复杂的任务中，奖励很难指定，例如，环境在 agent 终止行为之前奖励稀疏。此外，人为设计奖励函数具有很大的主观性，很难制定合适的奖励函数。在很多实际任务中存在一些专家完成任务的序列被认为获取了比较高的累积奖励。人类专家在完成复杂任务时,可能未考虑奖励函数。但是,这并不是说人类专家在完成任务时就没有奖励函数。从某种程度上来讲,人类专家在完成具体任务时有潜在的奖励函数。Ng 等人提出,专家在完成某项任务时,其决策往往是最优或接近最优的,可以假设,当所有的策略所产生的累积奖励期望都不比专家策略所产生的累积奖励期望大时,所对应的奖励函数就是根据示例学到的奖励函数。

因此,IRL 可以定义为从专家示例中学到奖励函数,即 IRL 考虑的情况是在 MDP 中,奖励函数未知。而专家在完成在此 MDP 环境下的某项任务时，其策略往往是最优或接近最优的。假设，当所有的策略所产生的累积回报期望都不比专家策略所产生的累积回报期望大时，强化学习所对应的奖励函数就是根据专家示例学到的回报函数。下图为逆强化学习的流程图，有一个由专家演示轨迹组成的集合（Expert trajectories），逆向强化学习通过学习专家演示轨迹学习到专家策略，进而按照专家策略生成该 MDP 下的奖励函数。



逆强化学习的流程图

8.1.2 分类



如果将最开始逆强化学习的思想用数学的形式表示出来，那么这个问题可以归结为最大边际化问题。如上图所示，这是逆强化学习最早的思想。根据这个思想发展起来的算法包括：学徒学习（Apprenticeship learning）,MMP 方法（Maximum Margin Planning）,结构化分类（SCIRL）和神经逆向强化学习（NIRL)。

最大边际形式化的最大缺点是很多时候不存在单独的回报函数使得专家示例行为既是最优的又比其他任何行为好很多，或者有很多不同的回报函数会导致相同的专家策略，也就是说这种方法无法解决歧义的问题。

基于概率模型的方法可以解决歧义性的问题。研究者们利用概率模型又发展出了很多逆向强化学习算法，如最大熵逆向强化学习、相对熵逆向强化学习、最大熵深度逆向强化学习，基于策略最优的逆向强化学习等等。

8.2 分层强化学习

传统强化学习方法面临着维度灾难，即当环境较为复杂或者任务较为困难时，agent 的状态空间过大，会导致需要学习的参数以及所需的存储空间急速增长，传统强化学习方法难以取得理想的效果。 为了解决维度灾难的问题，研究者提出了分层强化学习（Hierarchical Reinforcement Learning)。分层强化学习的主要目标是将复杂的问题分解成多个小问题，分别解决小问题从而达到解决原问题的目的。近些年来，人们认为分层强化学习基本可以解决强化学习的维度灾难问题，转而将研究方向转向如何将复杂的问题抽象成不同的层级，从而更好地解

决这些问题。

分层强化学习是将复杂的强化学习问题分解成一些容易解决的子问题，通过分别解决这些子问题，最终解决原本的强化学习问题。

常见的分层强化学习方法可以大致分为四大类：基于选项（option）的强化学习、基于分层抽象机（Hierarchical of abstract machines） 的分层强化学习、 基于 MaxQ 函数分解（ MaxQ value function decomposition） 的分层强化学习，以及端到端的（end to end）的分层强化学习。

8.3 多 agent 强化学习

8.3.1 背景

多 agent 系统由一群有自主性的，可互相交互的 agent 组成，它们共享一个环境，各自感知环境并各自采取行动。多智能体在现实生活中已有应用，比如说，多个机器人的控制，语言的交流，多玩家的游戏，以及社会问题分析等。

多 agent 强化学习就是将强化学习的算法与方法论运用在真实复杂的多智能体环境中来解决最优的决策的问题。如果使用传统的强化学习方法，如基于值函数的方法，由于在训练过程中，各 agent 的行为都在变化，因此每个 agent 的角度来看，环境并不稳定。给训练稳定性带来了挑战，并且直接阻碍了利用 experience replay，因此无法直接使用 DQN 算法。如果使用策略梯度方法，因为 agent 的 reward 通常取决于许多 agent 的动作，agent 自己动作的 reward 表现出更多的可变性，从而增加了其梯度的方差。所以会面临相比于单 agent 更大的方差问题，导致算法难以收敛。

8.3.2 算法分类

Fully cooperative	
Static	**Dynamic**
JAL [29] *FMQ* [59]	*Team-Q* [70] *Distributed-Q* [67] *OAL* [136]

Fully competitive
Minimax-Q [69]

Mixed	
Static	**Dynamic**
Fictitious Play [19] *MetaStrategy* [93] *IGA* [109] *WoLF-IGA* [17] *GIGA* [144] *GIGA-WoLF* [14] *AWESOME* [31] *Hyper-Q* [124]	*Single-agent RL* [32, 75, 105] *Nash-Q* [49] *CE-Q* [42] *Asymmetric-Q* [64] *NSCP* [138] *WoLF-PHC* [17] *PD-WoLF* [5] *EXORL* [116]

可以从不同维度来分类 MARL 算法，如果从任务的类型来分的话，MARL 算法可以分成如下三大类：完全合作，完全竞争，混合任务

（1） 完全合作

完全合作是指：各个 agent 齐心协力，共同朝着总体 reward 最高的方向努力。

如果在随机博弈中，存在一个中心控制者，可以控制其他 Agent 的行动。那么就能求得最佳的联合动作值，随机博弈就退化成马尔科夫决策过程，并且可以用 Q-learning 算法求解。

$$Q_{k+1}(x_k, \boldsymbol{u}_k) = Q_k(x_k, \boldsymbol{u}_k) + \alpha\left[r_{k+1} + \gamma \max_{\boldsymbol{u}'} Q_k(x_{k+1}, \boldsymbol{u}') - Q_k(x_k, \boldsymbol{u}_k)\right]$$

但是大部分系统是不存在中心控制者，那么是否可以假定其他 Agent 都是选择当前状态下最优的动作，在这种假定下，再选择对自己最优的动作，即

$$\overline{h}_i^*(x) = \arg\max_{u_i} \max_{u_1,\ldots,u_{i-1},u_{i+1},\ldots,u_n} Q^*(x, \boldsymbol{u})$$

在特定状态下，一些组合动作才可能取得最优结果(此时 Agent 不一定都是取自己最优的动作)。这时就需要协同机制，来协调 Agent 的动作。

（2）完全竞争

完全竞争环境是指：每个 agent 都只关心自己的 reward，想要最大化自己的 reward，并不考虑自己想要最大化自己的 reward 的 action 对于他人的影响。

在完全竞争的随机博弈中，可以应用最小最大化(minimax)原则：在假定对手会采取使自己收益最小化的动作的情况下，采取使自己收益最大的动作（即以最坏的恶意揣度对手)。 minimax-Q 算法基于最小最大化原则，使用下式来更新 Agent 1 的策略函数和 Q 函数。

$$h_{1,k}(x_k, \cdot) = argm_1(Q_k, x_k)$$

$$Q_{k+1}(x_k, u_{1,k}, u_{2,k}) = Q_k(x_k, u_{1,k}, u_{2,k}) + \alpha[r_{k+1} + \gamma m_1(Q_k, x_{k+1}) - Q_k(x_k, u_{1,k}, u_{2,k})]$$

其中 m1 是 Agent 1 的最小最大值

$$m_1(Q, x) = max_{h_1(x,\cdot)} min_{u2} \sum_{u1} h_1(x, u_1) Q(x, u_1, u_2)$$

minmax-Q 算法是对手独立的，不管对手如何选择，总能取得不差于 minmax 函数回报值。但如果对手不是采取最优策略(即使自己的收益最小化)，

而且能对对手建模，那么就可以取得更优的动作。

（3）混合任务

在混合任务中，又分为静态任务和动态任务。大部分算法是针对静态任务的，而且主要是关注适应性。其中最简单的是直接应用单 Agent 算法。即每个 Agent 的 Q 函数都只跟自己相关，对对手无感知。在特定的博弈中，单 Agent 算法是可以收敛至协调均衡的。但是在其他情况下，会存在不稳定的循环震荡。

8.4 深度强化学习

近年来,深度学习(Deep Learning,DL)作为机器学习领域一个重要的研究热点,已经在图像分析、语音识别、自然语言处理等领域取得了令人瞩目的成功.DL 的基本思想是通过多层的网络结构和非线性变换,组合低层特征,形成抽象的、易于区分的高层表示,以发现数据的分布式特征表示.因此 DL 方法侧重于对事物的感知和表达。强化学习(Reinforcement Learning,RL)作为机器学习领域另一个研究热点,已经广泛应用于工业制造、仿真模拟、机器人控制、优化与调度、游戏博弈等领域。 RL 的基本思想是通过最大化智能体(agent)从环境中获得的累计奖赏值,以学习到完成目标的最优策略).因此 RL 方法更加侧重于学习解决问题的策略.随着人类社会的飞速发展,在越来越多复杂的现实场景任务中,需要利用 DL 来自动学习大规模输入数据的抽象表征,并以此表征为依据进行自我激励的 RL,优化解决问题的策略.由此,谷歌的人工智能研究团队 DeepMind 创新性地将具有感知能力的 DL 和具有决策能力的 RL 相结合，形成了深度强化学习。

分为基于值函数的深度强化学习，基于策略梯度的深度强化学习，基于搜索和监督的深度强化学习，分层深度强化学习，多任务迁移深度强化学习，多 agent 深度强化学习，基于记忆与推理的深度强化学习。

参考文献

《Reinforcement Learning: An Introduction (2nd Edition)》

《Algorithms for Reinforcement Learning》

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

https://zhuanlan.zhihu.com/sharerl

https://zhuanlan.zhihu.com/p/34747205

《Large-Scale Machine Learning with Stochastic Gradient Descent》

《强化学习研究综述》

《分层强化学习综述》

《human-level control through deep reinforcement learning》

《deep reinforcement learning with double Q-learning》

《Dueling Network Architectures for Deep Reinforcement Learning》

《Transfer Learning for Reinforcement Learning Domains: A Survey》

《深度强化学习综述》

《deep reinforcement learning：overview》

《ReinforcementLearning_A Survey》