《人工智能》课程系列

强化学习基础*

武汉纺织大学数学与计算机学院

杜小勤

2018/11/16

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: November 16, 2019。

1 概述

强化学习 (Reinforcement Learning, RL) 属于无监督学习，它研究智能体 (Agent) 如何从与环境的交互中，通过获得成功与失败、奖励与惩罚的反馈信息来进行学习。

在监督学习中，智能体无需与环境进行交互，而只需要给它提供带有明确结果的标签数据 (教师)，对它进行训练。例如，在棋类对弈程序的设计中，监督型智能体需要被明确告知每种棋局下的正确走步。这样，经过良好训练的智能体就能学会已知棋局的正确下法，并且也能够学会未知棋局的正确下法 (这意味着智能

体具备了一定的泛化能力[1])。

还是考虑棋类对弈程序，如果缺少监督 (教师) 数据，智能体自己能够通过与环境的交互 (此处指自我对弈，即以 self-play 的方式) 来学会对弈吗？答案是肯定的。在缺少监督数据的情况下，智能体自己可以学会最优策略，可以学会对弈的转移模型，也可以学会预测对手的走步。

但是，有一个非常重要的前提条件，即智能体在与环境交互时，需要环境时时提供反馈信息——强化 (Reinforcement) 信息或奖励 (Reward) 信息[2]，以便让智能体知道哪些行为能够获得正奖励，而哪些行为获得负奖励，并据此调整自己的行为策略。这种学习方式，与动物依靠强化信息建立条件反射的原理非常类似——动物将疼痛与饥饿识别为负奖励，将快乐和食物识别为正奖励，并依此产生相应的行为。强化学习的名称即来源于此。

在一些环境中，奖励出现得比较频繁。例如，在乒乓球比赛中，每次得分或失分都可以看作是获得正奖励或负奖励；在学习爬行时，任何一次向前的运动都是一次成功。而在另一些环境中，奖励出现的次数较少，并且还可能是延时出现的。例如，在诸如围棋、国际象棋、TicTacToe 等棋类游戏中，每次的奖励总是出现在棋局终局时。这个问题被称为延时信度分配问题，它是强化学习面临的一个重要问题。

简而言之，强化学习的任务是利用奖励来学习针对某个环境的最优 (或接近最优) 的策略。

本章将从环境-智能体模型入手，首先介绍环境的 Markov 性质，由此引入环境的 MDP 建模方法，对两种值函数进行了定义，引出 Bellman 方程，然后给出基于效用的理性智能体的最优准则及两种值函数的 Bellman 最优方程。在此基础上，将讨论 MDP 的三种基本方法：动态规划、MC 方法及时序差分方法。在介绍三种方法的时候，始终抓住一条主线：策略迭代和广义的策略迭代方法，并指出三种方法的主要区别在于策略评估以及如何利用广义的策略迭代方法来改进算法效率等问题。最后，通过讨论 MDP 模型的局限性而引入 SMDP 模型，与 MDP 框架一样，将相继给出基于 SMDP 的值函数、最优值函数等定义，并给出基于 SMDP 模型的相关算法。

[1]无论是监督型智能体，还是无监督型智能体，都需要具备泛化能力。在强化学习中，结构信度分配算法用来解决该领域中智能体所需的泛化能力。

[2]在强化学习领域中，“奖励” 是正奖励与负奖励的统称。

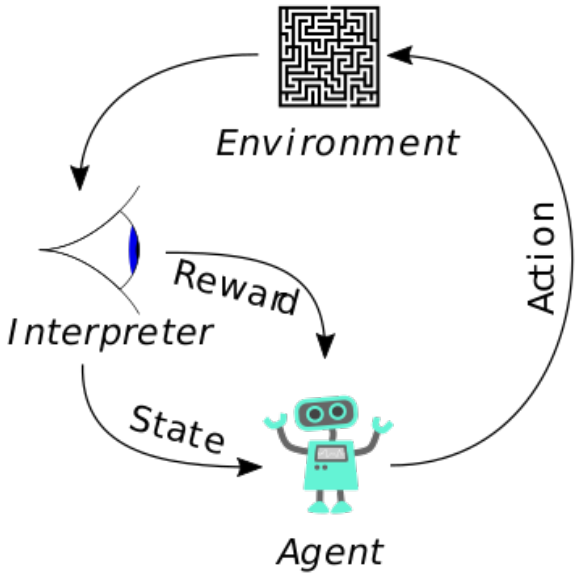

图 2-1: 环境与智能体的交互模型 (EA 模型)

2 环境-智能体交互模型

一个强化学习系统的基本框架主要由两部分组成，即环境和智能体。智能体可以通过传感器（Sensor）感知所处环境，并通过执行器（Actuator）对环境施加影响。从广义上讲，除该智能体之外，凡是与该智能体交互的物体，都可以被称为环境。如图2-1，显示的是环境与智能体进行交互的一个基本框架。环境与智能体的一个基本交互序列是：智能体感知到环境的当前状态，然后进行决策以选择合适的动作去执行，环境为此作出响应：更新状态并产生一个奖励或回报（Reward）。奖励是一个特定的数值信号，用来反映动作执行结果的受欢迎程度。一个基于效用的理性智能体其主要目标是努力地使自己在与环境的交互中积累尽可能多的奖励[3]。实际上，我们可以使用一种非常好的形式化模型——MDP（Markov Decision Problem[4]）来表达上述的环境—智能体（EA）模型。

[3]这是一个非正式的定义。实际上，我们应该定义一个性能函数来准确地反映智能体的目标，见下文。

[4]MDP(Markov Decision Problem) 是一个给性能函数(值函数)定义了最优准则的 MDP(Markov Decision Process)。

3 任务环境

环境的一些属性将会直接影响到智能体的决策及学习行为，我们把这些属性归为一组，并把它定义成一个任务环境 [1]：$T = (P, E, A, S)$，其中 P 表示性能，E 表示环境，A 表示执行器，S 表示传感器（这些元素合起来被称为 $PEAS$ 属性）。性能度量 P 是智能体成功程度标准的具体化，它应该反映出智能体设计者希望智能体从它与环境的交互中能够得到一个什么样的结果，比如智能体设计者希望出租车智能体能够以最短的路径到达目的地，那么就可以选取路径长度作为性能度量标准。实际上，在最优化领域，性能度量被称为效用函数 (Utility Function)，它实现了状态到实数的映射，一个状态的效用（值）反映了智能体对该状态的偏好程度 [1]。对于 MDP 模型来讲，值函数 (Value Function) 就是它的一个效用函数。

一般地，任务环境按某个属性被分成以下几类 [1]：

1. 完全可观察的与部分可观察的。两者的区别在于，智能体是否能够感知环境的完整状态。

2. 确定性的与随机的。两者的区别在于，环境的下一状态是否完全取决于当前的状态和智能体的动作。

3. 片段式的与连续式的。两者的区别在于，智能体的决策是否是阶段性的。

4. 静态的与动态的。两者的区别在于，环境在智能体思考的时候是否会发生变化。

5. 离散的与连续的。两者可以应用于环境的状态、时间的处理方式以及智能体的感知信息与动作，用来表示它们的值是否连续变化。

6. 单智能体与多智能体。两者的区别是显然的。

4 Markov 性质

基于强化学习的智能体是根据环境的状态来进行决策的。状态是指智能体可用到的任何信息，被看作是环境的一部分。在理想情况下，智能体希望当前状态能够包括历史信息，并且希望能一直保持所有的有用信息，因为它认为这些信息对它的决策是有帮助的。但是实际上，这是不可能的，对一些问题来讲，也是不必

要的。比较现实的情况是，智能体应该仅仅希望一个状态是具有 Markov 性质的 [2]：如果该状态保持了所有相关的信息，我们把这个状态称为是 Markov 状态[5]。

例如，一盘棋局被认为是一个 Markov 状态，因为以后棋局的胜负仅仅取决于当前棋局，而与历史棋局无关，虽然那些历史棋局形成了当前棋局。换句话说，当前棋局已经保存了所有关系到以后棋局胜负的信息（当然包括那些历史棋局的有关信息）。这意味着，以后棋局的胜负取决于当前棋局，而与怎样到达当前棋局的历史棋局没有什么关系。

形式地，智能体能够执行的动作用 a 来表示，感知到的环境状态用 s 来表示，获得的奖励用 r 来表示，那么从时刻 0 到时刻 t 的历史事件用 $s_0, a_0, r_1, ..., r_t, s_t, a_t$ 来表示。我们称状态满足 Markov 性质，如果对 $\forall s', r, s_t, a_t, t$，有如下公式成立：

$$\begin{aligned} &\Pr\{s_{t+1} = s', r_{t+1} = r \,|s_t, a_t\} = \\ &\Pr\{s_{t+1} = s', r_{t+1} = r \,|s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1} \ldots, r_1, s_0, a_0\} \end{aligned} \tag{1}$$

公式 (1) 意味着下一状态 s_{t+1} 和下一奖励 r_{t+1} 仅仅依赖于上一状态 s_t 和上一动作 a_t。如果公式 (1) 成立，那么我们称该环境满足 Markov 性质。

如果环境具有 Markov 性质，那么给定当前的状态和动作，使用公式 (1) 就能预测下一个状态和下一个奖励（期望值）。同理，通过迭代公式，我们就可以根据当前的状态和动作来预测将来的所有状态和奖励期望值。

5 Markov 决策过程

5.1 MDP 的定义

一个具有 Markov 性质的任务环境被称为一个 Markov 决策过程（Markov Decision Process, MDP）。如果状态空间和动作空间都是有限的，那么该 MDP 就是一个有限的 MDP。有限 MDP 是强化学习理论的基础与核心，对于正确理解强化学习理论与算法尤为重要。

形式地，一个 MDP 用一个五元组 $M = (S, A, \psi, T, R)$ 来表示。其中，S 表示环境的状态集合，A 表示智能体可执行的动作集合，$\psi \subseteq S \times A$ 是可接受的状态-动作对集合，$T : \psi \times S \to [0, 1]$ 是转移函数，$T(s'|s, a)$(即 $\Pr\{s_{t+1} = s'|s_t = s, a_t = a\}$)

[5]即使状态不具备 Markov 性质，仍然可以使用强化学习。此时的状态被称为是具有近似 Markov 性质的 [2]。

表示在状态 s 下执行动作 a 后转移到状态 s' 的概率，$R:\psi\to\mathbb{R}$ 是奖励函数，$R(s,a)$(即 $E\{r_{t+1}|s_t=s,a_t=a,s_{t+1}=s'\}$) 表示在状态 s 下执行动作 a 后获得的奖励期望值。我们用 A_s 或 $A(s)$(即 $\{a|(s,a)\in\psi\}$) 表示在状态 s 下可执行的动作集合，且 $A_s\subseteq A$。在后面，我们有时也用 $T^a_{ss'}$ 表示转移概率 $T(s'|s,a)$，用 $R^a_{ss'}$ 表示奖励期望值 $R(s,a)$ 。在上述定义中，转移函数与奖励函数都具有 Markov 性质，并且它们都是静态函数，即它们与时间 t 没有关系。从而有下列公式成立：

$$\begin{aligned} T^a_{ss'} &= \Pr\{s'|s,a\} \\ R^a_{ss'} &= E\left\{r\,|s,a,s'\right\} = \sum_r r\cdot\Pr\left\{r\,|s,a,s'\right\} \end{aligned} \tag{2}$$

在前面曾经提到过，基于强化学习的智能体是一种基于效用的理性智能体，这意味着智能体的决策是建立在效用函数的基础上的。在强化学习领域，把这种效用函数叫做值函数。目前，大多数强化学习算法是建立在值函数估算的基础上的。值函数是某个状态（或状态—动作对）的函数，用来表示智能体在该状态（或该状态下执行某个动作）的好坏或偏好程度。好坏的标准是通过将来奖励的期望值、精确值或期望回报值来确定的，也就是说，好坏的标准是针对一定的最优准则而言的。另外，由于奖励值是由智能体在某个状态下所采取的某个动作决定的，所以值函数一定要根据某个策略来确定。从以上描述可以看出，以下两点对智能体的决策至关重要：定义一个最优准则、确定最优值函数或最优策略。

5.2 最优准则

一个基于效用的理性智能体其主要目标是努力地使自己在与环境的交互中积累尽可能多的奖励。对于智能体的这一目标，我们应该采用某种量化形式来正式地定义它。假设在时刻 t 之后，智能体收到的奖励是随机变量序列 $r_{t+1},r_{t+2},r_{t+3},\ldots$，那么对于这个序列来讲，智能体的目标是使期望回报最大化，而期望回报被定义为奖励序列的某个特定函数。智能体有几种可选方式来定义该期望回报，从而以实现期望回报最大化的方式来实现它的目标。

期望回报最简单的方式是采用下面的公式：

$$R_t = r_{t+1}+r_{t+2}+\ldots+r_T \tag{3}$$

这意味着期望回报是时刻 t 之后 T 步的所有奖励之和。如果 T 是智能体与环境交互的最后一个时间步，那么我们就把这种交互方式称为是阶段性的 (Episodic)，称

这样的任务环境是阶段性任务。对于阶段性任务来讲，在一个阶段结束之后，环境被复位到初始状态，另一个阶段又接着开始运行，直到该阶段结束为止，上述过程循环反复。

然而，在有些情况下，智能体与环境之间的交互并没有分割为明确的阶段，而是会无限制地持续下去，我们把这种交互方式称为是连续性的 (Continuous)，称这样的任务环境是连续性任务。在连续性任务中，要使用公式 (3) 来作为期望回报是不可能的，因为它的最终时间步 $T=\infty$，从而导致 $R_t=\infty$。在这种情况下，我们就采用下面的公式：

$$R_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \ldots = \sum_{k=0}^{\infty} \left(\gamma^k \cdot r_{t+k+1}\right) \tag{4}$$

其中，$\gamma(0 \leq \gamma < 1)$ 是一个参数，称为折扣率。折扣率决定着将来的奖励对期望回报的贡献：当 $\gamma = 0$ 时，智能体是缺乏远见的，因为它只考虑使当前的奖励最大化；而当 γ 越接近 1 时，表明智能体是越有远见的，因为智能体越重视将来获得的奖励。从公式 (4) 可以看出，即使 $T=\infty$，只要 $\gamma < 1$ 且 $\{r_t\}$ 是有界的，那么 R_t 就是有限的。在某些情况下，允许 $\gamma = 1$，如果不会导致 R_t 是无限的。如果没有特殊说明，本文将采用这种形式的期望回报模型。然而，其它的期望回报形式也是可以的，如平均奖励模型 [4,5]。

5.3 值函数及最优值函数

前面曾经提到过，值函数一定要根据某个策略来确定。一个 MDP 的策略函数是 $\pi : \psi \to [0,1]$，$\pi(s,a)$ 表示在状态 s 下执行动作 a 的概率。策略可以是静态的，即 $\pi(s,a)$ 依据 s 进行简单的映射；策略也可以是非静态的，即 $\pi(s,a)$ 依据 s 和其它的因素进行动作映射，这些因素可以是智能体在环境中已经执行了的步数，或者是智能体过去经验的某个函数。对于阶段性任务来讲，它的策略（我们称它为有限域策略，Finite-Horizon Policy）可以是非静态的。对于连续性任务来讲，它的策略（我们称它为无限域策略，Infinite-Horizon Policy）一定是静态的 [6]。依据某个最优准则而得到的最优策略用集合 Π^* 来表示，而 MDP 算法的主要任务就是要找到一个策略 $\pi^* \in \Pi^*$ 或者 π^* 的一个近似策略。

下面来定义一个策略 π 上的值函数。假定智能体当前的状态是 s，那么状态 s 的值就是从状态 s 出发，然后一直采用策略 π 时所得到的期望回报。用下面的公

式来表示状态 s 处的值：

$$V^{\pi}(s)=E_{\pi}\{R_t|s_t=s\}=E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s\right\} \tag{5}$$

上式中的 $E_{\pi}\{\}$ 表示在策略 π 下的期望值，我们把 V^{π} 称为策略 π 的状态值函数。同样，我们也可以定义在状态 s 下执行动作 a 并且以后一直采用策略 π 时所得到的期望回报，用下面的公式来表示该值：

$$Q^{\pi}(s,a)=E_{\pi}\{R_t|s_t=s,a_t=a\}=E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s,a_t=a\right\} \tag{6}$$

我们把 Q^{π} 称为策略 π 的状态—动作值函数。

实际上，值函数可以用一种简单的递归关系形式来表示，这就是 Bellman 方程 [7]，它在不同状态的值函数 $V^{\pi}(\cdot)$ 之间建立了联系。具体地，设状态 s 的后继状态为 s'，那么它们的值函数之间存在如下关系：

$$\begin{aligned}
&V^{\pi}(s)=E_{\pi}\{R_t|s_t=s\}=E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s\right\}\\
&=E_{\pi}\left\{r_{t+1}+\gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t=s\right\}\\
&=\sum_a\pi(s,a)\sum_{s'}T^a_{ss'}\left[R^a_{ss'}+\gamma E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s'\right\}\right]\\
&=\sum_a\pi(s,a)\sum_{s'}T^a_{ss'}[R^a_{ss'}+\gamma V^{\pi}(s')].
\end{aligned} \tag{7}$$

类似地，可建立状态 s 的状态—动作值函数与后继状态 s' 的值函数之间的递归关系：

$$\begin{aligned}
&Q^{\pi}(s,a)=E_{\pi}\{R_t|s_t=s,a_t=a\}=E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s,a_t=a\right\}\\
&=E_{\pi}\left\{r_{t+1}+\gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t=s,a_t=a\right\}\\
&=\sum_{s'}T^a_{ss'}\left[R^a_{ss'}+\gamma E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s'\right\}\right]\\
&=\sum_{s'}T^a_{ss'}[R^a_{ss'}+\gamma V^{\pi}(s')].
\end{aligned} \tag{8}$$

在公式 (7) 与公式 (8) 中，$V^{\pi}(\cdot)$ 与 $Q^{\pi}(\cdot)$ 是 Bellman 方程的唯一解。这 2 个公式被称为一步值传递或更新操作，它们构成了强化学习方法中多步值传递或更新操作的基础。实际上，值传递指的是将值从后继的状态传到以前的状态。

有了以上基础，我们就能够回答一个基于强化学习的理性智能体的目标是什么了。大体而言，智能体的目标就是要找到一种策略，使得值函数（期望回报）最大化。一般来说，至少有一个这样的策略要优于或等于其它的策略，我们把这个策略称为最优策略（最优策略可能有几个），记为 π^*。我们称一个策略 π 优于另一个策略 π'，如果对于 $\forall s \in S$，有 $V^{\pi}(s) \geq V^{\pi'}(s)$ 成立。由 Bellman 方程可知，最优策略共有同一个值函数，我们称该值函数为最优值函数，记为 V^*，它被定义为：

$$V^*(s) = \max_{\pi} V^{\pi}(s), \forall s \in S \tag{9}$$

最优策略也共有同一个状态—动作值函数，记为 Q^*，它被定义为：

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \forall s \in S, a \in A \tag{10}$$

下面我们将给出最优策略下的值函数 V^* 与状态—动作值函数 Q^* 的递归表达形式（单步值传递或一步值传递），这两种形式的公式分别被称为 V^* 的 Bellman 最优方程与 Q^* 的 Bellman 最优方程。有两种方式可以推导出它们。第一种方式是采用通用公式 (7) 与公式 (8) 来推导，用 π^* 直接替换 π 可得如下公式：

$$\begin{aligned} V^*(s) &= \sum_a \pi^*(s,a) \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right] = \max_a \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right] \\ Q^*(s,a) &= \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right] = \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s',a') \right] \end{aligned} \tag{11}$$

第二种方式是直接依据 V^* 与 Q^* 的定义来推导，也可得出相同的公式：

$$
\begin{aligned}
&V^*(s) = \max_{\pi} V^{\pi}(s) = \max_{a} Q^*(s,a) = \max_{a} E_{\pi^*}\{R_t \,|s_t = s, a_t = a\} \\
&= \max_{a} E_{\pi^*}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|s_t = s, a_t = a\right\} \\
&= \max_{a} E_{\pi^*}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,|s_t = s, a_t = a\right\} \\
&= \max_{a} E\{r_{t+1} + \gamma V^*(s_{t+1}) \,|s_t = s, a_t = a\} \\
&= \max_{a} \sum_{s'} T_{ss'}^{a} \left[R_{ss'}^{a} + \gamma V^*(s')\right], \\
&Q^*(s,a) = E_{\pi^*}\{R_t \,|s_t = s, a_t = a\} = E_{\pi^*}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|s_t = s, a_t = a\right\} \\
&= E_{\pi^*}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,|s_t = s, a_t = a\right\} \\
&= E_{\pi^*}\{r_{t+1} + \gamma V^*(s_{t+1}) \,|s_t = s, a_t = a\} \\
&= E_{\pi^*}\left\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \,|s_t = s, a_t = a\right\} \\
&= \sum_{s'} T_{ss'}^{a} \left[R_{ss'}^{a} + \gamma \max_{a'} Q^*(s', a')\right].
\end{aligned}
\tag{12}
$$

对于具有有限状态与动作的 MDP，Bellman 最优方程有一个独立于策略的唯一的解 [6]。现在，假定智能体通过某种方式已经获得了 V^* 与 Q^*，那么下一步就是利用它们来获得一个最优策略，以指导智能体的行动并实现智能体的目标了。对于 V^* 来讲，从公式 (11) 可以看出，智能体只要在当前状态下再向前搜索一步就可以选取一个最优动作，该最优动作将能够使智能体获得最大的期望回报。因此，任何一个为最优动作赋予了非零概率的策略都将是一个最优策略。对于 Q^* 来讲，从公式 (11) 可以看出，智能体根本不需要向前搜索一步：对于任何状态，它只需要简单地找出使期望回报最大化的那个动作。因此，函数允许智能体在不知道任何关于环境动力特性的情况下来选择最优动作。

下面，我们将使用一种简洁的记法形式来表达 Bellman 方程 [8,9]。我们使用 $J_{\pi}(s)$ 来表示公式 (7) 右端的单步值传递操作，则公式 (7) 变为如下形式：

$$
V^{\pi}(s) = J_{\pi}(s) V^{\pi} \tag{13}
$$

使用 $J_{\pi^*}(s)$ 或 $J(s)$ 来表示公式 (11) 右端的单步值传递操作，则公式 (11) 变为如

下形式：

$$V^*(s) = J_{\pi^*}(s) V^* = J(s) V^* \tag{14}$$

使用对应的记法 J_π 和 J_{π^*}（或 J）来表示向量上的操作，即对整个状态空间进行操作以求取整个值函数，那么公式 (13) 与 (14) 分别具有如下形式：

$$\begin{aligned} V^\pi &= J_\pi V^\pi \\ V^* &= J_{\pi^*} V^* = J V^* \end{aligned} \tag{15}$$

对于状态—动作值函数 Q 和 Q^*，我们也可以定义非常相似的简写形式，此处不再赘述。

6 MDP 的基本方法

6.1 动态规划方法

动态规划（Dynamic Programming, DP）指的是计算完整 MDP 模型的最优策略的一类算法。Bellman[7] 表明了该方法可以应用于许多问题，而 Minsky[10] 在评论 Samuel 的棋类游戏实验 [11] 时首次将动态规划与强化学习联系了起来；随后 Andreae[12] 正式在强化学习框架内提出了可以将动态规划应用于策略迭代方法中的思想；Werbos 提出了一系列近似动态规划方法，用于解决连续状态问题 [13-18]，这些方法紧密关联下面要介绍的各种动态规划方法；Watkins[19] 提出的 Q-learning 方法成功地将动态规划方法以在线学习的方式应用于强化学习中，形成了所谓的增量式动态规划。在强化学习中，经典动态规划算法的实用性是有限的，一方面是因为它需要完整的 MDP 模型（具有完全的转移函数与奖励函数），另一方面是因为它的计算代价高。但是，经典的动态规划算法在理论上仍然是很重要的，它便于我们理解其它的强化学习算法。本质上，其它的强化学习算法就是要在较少计算需求量和不完全 MDP 模型的条件下，达到与经典的动态规划一样的效果。经典的动态规划算法与 Bellman 方程是密切相关的，前者可以通过将后者转换成赋值的形式（即单步值传递或更新）来不断地（连续地或迭代地）逼近实际值函数，这可以从5.3节提供的 Bellman 方程看出。

6.1.1 策略评估

所谓策略评估指的是计算给定策略 π 的值函数 V^{π}，它有时也被称为预测问题。策略评估是一个基本的问题，因为只有进行策略评估，才能比较两个策略的质量。为方便起见，我们将公式 (7) 重新列出：

$$V^{\pi}(s)=\sum_{a}\pi(s,a)\sum_{s'}T_{ss'}^{a}\left[R_{ss'}^{a}+\gamma V^{\pi}(s')\right],\forall s\in S \tag{16}$$

如果 MDP 的动力特性完全已知，那么我们可以获得 $|S|$ 个形如公式 (16) 的联立线性方程组。理论上，我们可以直接求取该方程组而获得值函数 V^{π}。然而，出于效率和计算简便性等方面的考虑，我们使用迭代算法来获得值函数 V^{π}。该迭代算法的更新规则直接来源于公式 (16)，如下所示：

$$V_{k+1}^{\pi}(s)=\sum_{a}\pi(s,a)\sum_{s'}T_{ss'}^{a}\left[R_{ss'}^{a}+\gamma V_{k}^{\pi}(s')\right],\forall s\in S \tag{17}$$

在上式中，k 是迭代步数。因为 V^{π} 的唯一性，所以在 V^{π} 存在的条件下，$\{V_k\}$ 序列在 $k\to\infty$ 时将收敛于 V^{π}。正因为上述算法的主要操作是迭代，所以这种策略评估算法又被称为迭代策略评估算法，下面给出该算法：

input π, *the policy to be evaluated*

initialize $V(s)=0$, *for all* $s\in S$

repeat

$\Delta\leftarrow 0$

for each $s\in S$:

$v\leftarrow V(s)$

$V(s)\leftarrow\sum_{a}\pi(s,a)\sum_{s'}T_{ss'}^{a}\left[R_{ss'}^{a}+\gamma V(s')\right]$

$\Delta\leftarrow\max(\Delta,|v-V(s)|)$

until $\Delta<\theta$ (*a small positive number*)

output $V\approx V^{\pi}$

6.1.2 策略改进

策略评估可以用于计算给定策略的值函数，其最终目的是为了找到更好的策略。策略改进是一种利用策略评估算法来计算初始策略和改进策略的值函数，并

依据贪心原则来得到改进策略的方法。其基本思路是，在某一状态 s 下选择动作 $a \neq \pi(s)$，此后则一直采用初始策略 π（实际上，这个过程就得到了一个新的策略 π'），如果有 $V^{\pi'}(s) > V^{\pi}(s)$ 成立，则意味着对 $\forall j \in S$，$V^{\pi'}(j) \geq V^{\pi}(j)$ 也成立，那么新策略 π' 要优于初始策略 π。在状态 s 下选择动作 a 的值按如下公式计算：

$$Q^{\pi}(s,a) = E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \,|s_t = s, a_t = a\right\} = \sum_{s'} T^{a}_{ss'}\left[R^{a}_{ss'} + \gamma V^{\pi}(s')\right] \quad (18)$$

下面我们将表明 $V^{\pi'}(j) \geq V^{\pi}(j)$ 为什么是成立的。在公式 (18) 中，$a = \pi'(s)$；而对 $\forall l \in S$ 且 $l \neq s$，有 $\pi'(l) = \pi(l)$ 且 $V^{\pi'}(l) = V^{\pi}(l)$ 成立，把这两种情况综合在一起得到：

$$\begin{cases} Q^{\pi'}(s,\pi'(s)) = V^{\pi'}(s) = \sum_{s'} T^{a}_{ss'}\left[R^{a}_{ss'} + \gamma V^{\pi}(s')\right], \exists s \in S, a = \pi'(s), \\ V^{\pi'}(l) = V^{\pi}(l), \forall l \in S, l \neq s. \end{cases} \quad (19)$$

从公式 (19) 可以看出，由于 $V^{\pi'}(s) > V^{\pi}(s)$ 成立，那么对 $\forall j \in S$，$V^{\pi'}(j) \geq V^{\pi}(j)$ 也成立。

实际上，上面的讨论可以更一般化。设 π 和 π' 是任意两个策略，如果对 $\forall s \in S$，有 $Q^{\pi}(s,\pi'(s)) \geq V^{\pi}(s)$ 成立，那么也有 $V^{\pi'}(s) \geq V^{\pi}(s)$ 成立，或者说策略 π' 至少与策略 π 一样好。下面的公式推导可以证明它：

$$\begin{aligned} V^{\pi}(s) &\leq Q^{\pi}(s,\pi'(s)) = E_{\pi'}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \,|s_t = s\right\} \\ &\leq E_{\pi'}\left\{r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) \,|s_t = s\right\} \\ &= E_{\pi'}\left\{r_{t+1} + \gamma E_{\pi'}\left\{r_{t+2} + \gamma V^{\pi}(s_{t+2})\right\} |s_t = s\right\} \\ &= E_{\pi'}\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}(s_{t+2}) \,|s_t = s\right\} \\ &\leq E_{\pi'}\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^{\pi}(s_{t+3}) \,|s_t = s\right\} \\ &\vdots \\ &\leq E_{\pi'}\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots |s_t = s\right\} \\ &= V^{\pi'}(s). \end{aligned} \quad (20)$$

另外，如果 $\exists s \in S$，有 $Q^{\pi}(s,\pi'(s)) > V^{\pi}(s)$ 成立，那么就表明策略 π' 一定比策略 π 好。实际上，上述结论就是策略改进定理 [7,20]，把 $Q^{\pi}(s,\pi'(s)) \geq V^{\pi}(s)$ 称为策略改进定理的条件。

可以推测，如果存在一个贪心策略 π'，它在每个状态都选择使得 $Q^{\pi}(s,\pi'(s)) \geq V^{\pi}(s)$ 成立的那个动作，那么该贪心策略 π' 至少与 π 一样好。我们把这种通过使

用贪心方法来改进初始策略以得到贪心策略 π' 的过程称为策略改进。下面是贪心策略 π'（或策略改进）的生成公式：

$$\begin{aligned}\pi'(s) &= \arg\max_a Q^{\pi}(s,a) \\ &= \arg\max_a E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \,|s_t = s, a_t = a\right\} \\ &= \arg\max_a \sum_{s'} T^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right].\end{aligned} \tag{21}$$

实际上，策略改进给出了一个生成最优策略的方法——策略迭代（见下一小节）。其直观的理解是，如果迭代运行策略改进算法，那么贪心策略 π' 与初始策略 π 存在两种关系：π' 与 π 一样好、π' 优于 π，对于前者，有 $V^{\pi'} = V^{\pi}$ 成立。此时，由公式 (18) 和 (21)，对 $\forall s \in S$，可得下列公式：

$$\begin{cases} V^{\pi'}(s) = Q^{\pi}(s,a) = \max\limits_a \sum\limits_{s'} T^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right], \\ V^{\pi'} = V^{\pi}. \end{cases} \\ \Rightarrow V^{\pi'}(s) = \max_a \sum_{s'} T^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi'}(s')\right]. \tag{22}$$

可以看出，公式 (22) 与 Bellman 最优公式 (11) 相同，因此有 $V^{\pi'} = V^{\pi} = V^*$，说明 π' 与 π 都是最优策略；对于后者，π' 将优于 π，表明贪心的策略改进算法总是能够给出一个严格的较优策略，除非 π' 已经是最优策略了。

6.1.3 策略迭代

策略迭代算法合并了策略评估与策略改进算法，可以同时求取最优策略和最优值函数。其执行过程大致如下所示：

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*. \tag{23}$$

其中，π_0 是任意的初始策略，E 表示策略评估，I 表示策略改进。可以看出，策略迭代算法是策略评估与策略改进交替执行的过程。据上一小节的讨论可知，每次的策略评估总能保证产生出一个严格的较优策略，除非已经是最优策略了；并且，由于有限状态与动作的 MDP 只有有限个策略，所以策略迭代算法必然在有

限次迭代后收敛于一个最优策略和最优值函数。下面给出策略迭代算法：

1.*initialization*

$V(s) \in \mathbb{R}$ *and* $\pi(s) \in A(s)$ *arbitrarily for all* $s \in S$

2.*policy evaluation*

repeat

$\Delta \leftarrow 0$

for each $s \in S$

$v \leftarrow V(s)$

$$V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V(s')\right]$$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (*a small positive number*)

3.*policy improvement*

$policy - stable \leftarrow true$

for each $s \in S$

$b \leftarrow \pi(s)$

$$\pi(s) \leftarrow \arg\max_a \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V(s')\right]$$

if $b \neq \pi(s)$, *then* $policy - stable \leftarrow false$

if $policy - stable$, *then stop*; *else goto* 2

6.1.4 值迭代

策略迭代的一个缺陷是在每次迭代中都需要执行完全的策略评估过程，即求取当前策略的值函数 V^π。实际上，策略迭代中的策略评估过程可以不用完整地执行，而不会影响收敛性，我们把这种处理技术称为策略评估的截断 [21]。一种有效的方法是在策略评估执行一次后就停止迭代，然后就执行策略改进。这种算法执行的效果就等同于执行一个特殊的单步值传递操作，该操作（更新规则）如下：

$$\begin{aligned} V_{k+1}(s) &= \max_a E_\pi \left\{r_{t+1} + \gamma V^\pi_k(s_{t+1}) \,|s_t = s, a_t = a\right\} \\ &= \max_a \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s')\right], \forall s \in S. \end{aligned} \tag{24}$$

对任意 V_0，序列 $\{V_k\}$ 在 V^* 存在的情况下必然收敛于 V^*。之所以说该更新操作比较特殊，原因就在于它与 Bellman 最优公式 (11) 是非常相似的。因此，从这个角度看，这种方法是简单地将 Bellman 最优公式作为自己的更新规则，我们把这种算法称为值迭代。

值迭代算法的终止条件与策略评估一样。虽然在形式上值迭代算法与策略评估算法非常相似，但是实际上，值迭代算法在每次遍历状态空间的时候，都有效地将单步策略评估（截断）遍历与策略改进遍历紧密地结合在一起了。一般情况下，可以对值迭代算法进行改进：使用多步策略评估（截断）遍历，这通常可以加快算法的收敛速度。下面给出值迭代的算法：

$$
\begin{aligned}
&initialize\ V(s)=0,\ for\ all\ s\in S\\
&repeat\\
&\quad \Delta \leftarrow 0\\
&\quad for\ each\ s\in S:\\
&\quad\quad v\leftarrow V(s)\\
&\quad\quad V(s)\leftarrow \max_a \sum_{s'} T^a_{ss'}\left[R^a_{ss'}+\gamma V(s')\right]\\
&\quad\quad \Delta \leftarrow \max\left(\Delta, |v-V(s)|\right)\\
&until\ \Delta<\theta\,(a\ small\ positive\ number)\\
&output\ V\approx V^*\\
&output\ a\ deterministic\ policy\ \pi^*,\ such\ that\\
&\pi(s)\leftarrow \arg\max_a \sum_{s'} T^a_{ss'}\left[R^a_{ss'}+\gamma V(s')\right]
\end{aligned}
$$

6.2 Monte Carlo 方法

Monte Carlo(MC) 方法是一种使用样本平均回报值来估计值函数并发现最优策略的在线学习算法。在强化学习领域，Michie 等人 [22] 首次在平衡摆实验中使用 MC 方法来估算状态—动作值函数；Barto 等人 [23] 研究了基于 MC 方法的策略迭代问题，用来求解线性方程；Singh 等人 [24] 比较了两种基于 MC 的策略评估方法：EV 和 FV（6.2.1-策略评估）。与动态规划算法不同，MC 方法不需要完整的环境模型，而是通过在线交互的方式来获得状态、动作和奖励的样本序列，然后估计值函数并进行策略改进，上述过程反复迭代并最终获得最优策略。MC 方法

的基本思想是，因为状态（和状态—动作）值是回报的数学期望，所以可以通过对被访问的状态所获得的回报求平均值的方式来估算该值，而如果能够无限地访问每个状态，那么它们的回报平均值就应该收敛于期望值了。一般而言，MC 方法只适用于阶段性任务，并只在一个阶段结束后更新值函数和策略，所以 MC 方法是以阶段为基础进行渐进学习的，而不是象 TD（Temporal Difference，后面将介绍）方法那样，在每个时间步进行学习。

与动态规划方法一样，MC 方法首先要考虑策略评估，即计算给定策略的值函数，然后是进行策略改进，最后是利用策略迭代来获得最优策略。

6.2.1 策略评估

基于 MC 方法的策略评估是指智能体在给定策略 π 的控制下，与环境交互一系列阶段，并采用求取平均回报值的方式来估算值函数 V^π 和 Q^π。在每个阶段的采样中每出现一次状态 s，我们就称为对 s 的一次访问。在每个阶段的采样中状态 s 的第一次出现，我们就称为对 s 的首次访问。有两种方法用来进行策略评估，一种方法是 EV(Every-Visit) 方法，另一种是 FV(First-Visit) 方法。其中，EV 方法是计算所有阶段的对状态 s 的所有访问的平均回报值，而 FV 方法是计算所有阶段的对状态 s 的首次访问的平均回报值。这两种方法非常相似，但理论特性上有一些细微差别，我们将采用 FV 方法来进行策略评估。无论是 EV 方法，还是 FV 方法，它们都随着对状态 s 的访问次数趋向无穷而收敛于 $V^\pi(s)$。它们的算法形式比较简单而直接，此处不再赘述。

一个值得注意的问题是，在 MC 方法中，由于环境的模型是未知的，所以估算状态—动作值函数 Q^π 对我们来说更加有用。主要原因在于，Q^π 函数可以在不知道任何关于环境动力特性的情况下指导智能体如何行动，而 V^π 需要环境模型才能实现这一功能，见5.3节。然而，在利用 EV 方法或 FV 方法估算 Q^π 时，将会碰到一个问题：怎样保证所有的状态—动作对能够被无限次地访问到，因为只有保证每个状态—动作对被无限次地访问到，才能确保它们的平均回报值收敛于 Q^π。这个问题确实存在，因为假如 π 是一个确定性策略（与随机性策略相对应），每个状态下的被执行动作将只能是某个固定的动作，那么所计算的平均回报值也只能是该状态—动作对的，而其它的状态—动作对的平均回报值将不会随着经验的积累而得到更新。这是一个严重的问题，特别是，我们估算 Q^π 的主要目标是要获得每个状态下的最优动作——最优策略，而上述的确定性策略却阻止我们去

估算其它的状态—动作对的值，从而阻止了最优状态—动作对的发现。

实际上，上面讨论的这个问题涉及到一个概念——维持探索（Maintaining Exploration, ME），即为确保对 Q^{π} 的估算是有效和正确的，必须确保智能体在环境中执行的动作具有持续的探索性质，而不是具有贪婪性质。前者意味着智能体总是尝试着去运行那些就目前而言并不是最优的那些动作，而后者意味着智能体总是执行就目前而言是最优的动作[6]。有两种方法可以维持探索，一种是所谓的探索式启动（Exploring Starts, ES），即每个阶段的开始所有的状态—动作对都有机会被选中运行；另外一种方法是智能体持续地引入随机性动作选择机制给待评估策略（如 ϵ-greedy 策略等）或直接使用随机性策略（如 soft 策略等）来与环境交互，前者形成了所谓的 On-Policy 方法，后者形成了所谓的 Off-Policy 方法。

6.2.2 策略改进与策略迭代——MC 控制

本小节将讨论 MC 的控制问题，即智能体如何学会最优控制。它的基本思路与动态规划一样，交替执行策略评估与策略改进——策略迭代，其目标是获得一个（近似）最优策略。更一般地，根据需要，我们可以使用一个广义的策略迭代（Generalized Policy Iteration，GPI）模型 [2]。类似地，我们建立一个基于 MC 方法的策略迭代模型，如下所示：

$$\pi_0 \xrightarrow{E} Q^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} Q^{\pi_1} \xrightarrow{I} \pi_2 \cdots \xrightarrow{I} \pi^* \xrightarrow{E} Q^* \tag{25}$$

其中，π_0 是任意的初始策略，E 表示策略评估，I 表示策略改进。依照上一小节的讨论，策略评估阶段必须维持探索且要保证“无限次”的阶段运行次数，才能保证策略评估的准确性。此处，我们假定智能体使用探索式启动（ES）的方式来维持探索。策略改进也采用相对于当前值函数的贪心方法来实现，即对某个状态 s，确定性地选择能够导致该状态获取最大回报值的那个动作 a，并且 Q^{π_k} 不需要环境模型就能够构造这个贪心策略 $Q^{\pi_{k+1}}$：

$$\pi_{k+1}(s) = \arg\max_{a} Q^{\pi_k}(s, a) \tag{26}$$

[6]这个问题也被称为“利用 (Exploitation) 与探索 (Exploration)”问题，在强化学习等领域是一个非常经典的问题，得到了广泛的研究。

该策略改进是有效的，因为它符合策略改进定理（6.1.2节）：

$$\begin{aligned} & V^{\pi_{k+1}}(s) = Q^{\pi_{k+1}}(s, \pi_{k+1}(s)) = Q^{\pi_k}(s, \pi_{k+1}(s)) \\ & = Q^{\pi_k}\left(s, \arg\max_a Q^{\pi_k}(s,a)\right) \\ & = \max_a Q^{\pi_k}(s,a) \geq Q^{\pi_k}(s, \pi_k(s)) \\ & = V^{\pi_k}(s). \end{aligned} \tag{27}$$

正如前面所讨论的，该定理确保了 π_{k+1} 总是要好于 π_k，除非两者已经相等，而在这种情况下，两者都已经是最优策略了。因此，从上面的讨论可以看出，基于 MC 方法的策略迭代能够只通过阶段采样的渐进学习方式来进行策略评估与策略改进，并最终获得最优策略与最优值函数，在整个过程中，它并不需要环境的模型。

然而，上述针对 MC 策略迭代模型 (公式25) 的讨论中，存在着两个前提假设：维持探索假设和无限次阶段运行假设。对于这两个假设，我们都能够找到比较合适的方式来设法满足它，或简单地避免它。本小节将讨论无限次阶段运行假设的问题，至于维持探索假设问题，我们将在后面的两小节中予以讨论。

实际上，在迭代策略评估算法（6.1.1节）中已经碰到了无限次运行假设问题。解决措施之一是，如迭代策略评估算法那样设置一个较小的数，用来表明策略的值函数已经获得了某种程度的近似。然而，这种方式是缺乏效率的，也是不必要的，因为广义的策略迭代（GPI）模型 [2] 建议我们在进行策略改进之前，可以不需要进行完整的策略评估。由此，我们便得到了第二个解决措施，这便是采取广义的策略迭代模型，而不是经典的策略迭代模型（6.1.3节及本小节开头所示的迭代模型）。值迭代算法（6.1.4节）就是一个极端的例子，它在两个策略改进过程之间只进行单步策略评估。也许，对于基于 MC 方法的值迭代方法来说就更为特殊了，因为它在两个策略改进过程之间（实际上是两个阶段之间）只进行被访问状态的值与策略的更新。下面将给出一个实用的基于 MC 方法的值迭代算法——Monte Carlo with Exploring Starts(MC ES) 方法。虽然该算法象其它的策略迭代算法那样，将不可避免地收敛于最优策略和最优值函数，但是它的收敛性证明仍

然是强化学习领域中几个最基本的已知问题之一。MC ES 算法如下：

$$
\begin{aligned}
&initialize,\ for\ all\ s \in S, a \in A(s):\\
&\quad Q(s,a) \leftarrow arbitrary\\
&\quad \pi(s) \leftarrow arbitrary\\
&\quad returns(s,a) \leftarrow empty\ list\\
&repeat\ forever:\\
&\quad (a)\ generate\ an\ episode\ using\ exploring\ starts\ and\ \pi\\
&\quad (b)\ for\ each\ pair\ s,\ a\ appearing\ in\ the\ episode:\\
&\qquad\quad r \leftarrow return\ following\ the\ first\ occurrence\ of\ s,\ a\\
&\qquad\quad append\ r\ to\ returns(s,a)\\
&\qquad\quad Q(s,a) \leftarrow average(returns(s,a))\\
&\quad (c)\ for\ each\ s\ in\ the\ episode:\\
&\qquad\quad \pi(s) \leftarrow \arg\max_a Q(s,a)
\end{aligned}
\tag{28}
$$

6.2.3 On-Policy MC 控制

本小节及下一小节讨论维持探索问题。在前面的两节中，我们使用了探索式启动 (ES) 来解决维持探索问题，然而，这种方式并不适用于那些与环境存在实际交互的学习，或者说不是很实用。本节介绍的 On-Policy 方法通过在给定的行为策略中引入随机性来维持探索，能够不断地改进行为策略并最终获得一个最优的行为策略和最优值函数，这种行为策略既用于产生智能体的行为，或者说用于控制智能体，或者说是智能体的决策策略，同时又是待评估和改善的策略。

我们首先介绍软策略（Soft Policy）的概念，一个策略是软的，如果 $\forall s \in S, a \in A(s)$ 有 $\pi(s,a) > 0$。On-Policy 方法存在许多的变体，本节介绍的方法引入 ϵ-greedy 概念给行为策略，从而形成了所谓的 ϵ-greedy 策略，它的含义是：以大概率 $1-\epsilon$ 选择一个最优动作执行，而以小概率 ϵ 随机地选择 $A(s)$ 中的一个动作去执行，其中 ϵ 是一个小的正数。可以看出，ϵ-greedy 策略是软的，并且是 ϵ-soft 策略的一种。所谓 ϵ-soft 策略指的是 $\forall s \in S, a \in A(s)$ 有 $\pi(s,a) \geq \epsilon/|A(s)|$。在所有的 ϵ-soft 策略中，ϵ-greedy 策略是最贪婪的一种。

从总体上看，On-Policy MC 方法是一个广义的策略迭代，它使用 FV 方法去

进行策略评估，使用 ϵ-greedy 方法进行策略改进。注意，由于没有使用探索式启动技术来维持探索，所以 On-Policy MC 方法不能象 MC ES 方法那样使用贪心的策略改进方法，而是使用 ϵ-greedy 的策略改进方法，这对于广义的策略迭代来说是允许的。下面的讨论将表明策略改进定理对基于 ϵ-greedy 的策略改进是成立的：设 π 是任意一个 ϵ-soft 策略，那么任意一个相对于 Q^π 的基于 ϵ-greedy 方法的改进策略 π' 至少与 π 一样好。对于 $\forall s \in S$，有下面的公式成立：

$$\begin{aligned}
&V^{\pi'}(s) = Q^{\pi'}(s, \pi'(s)) = Q^{\pi}(s, \pi'(s)) = \sum_a \pi'(s,a) Q^\pi(s,a) \\
&= \frac{\varepsilon}{|A(s)|} \sum_a Q^\pi(s,a) + (1-\varepsilon) \max_a Q^\pi(s,a) \\
&\geq \frac{\varepsilon}{|A(s)|} \sum_a Q^\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(s,a) - \varepsilon/|A(s)|}{1-\varepsilon} Q^\pi(s,a) \\
&= \frac{\varepsilon}{|A(s)|} \sum_a Q^\pi(s,a) - \frac{\varepsilon}{|A(s)|} \sum_a Q^\pi(s,a) + \sum_a \pi(s,a) Q^\pi(s,a) \\
&= V^\pi(s).
\end{aligned} \tag{29}$$

其中，第三步之所以成立是因为 π 是一个 ϵ-soft 策略，从而对 $\forall s \in S, a \in A(s)$ 有 $\pi(s,a) \geq \epsilon/|A(s)|$；而 π' 是一个 ϵ-greedy 策略，从而对 $\forall s \in S, a \in A(s)$ 且 $a \neq \arg\max_b Q^{\pi'}(s,b)$ 有 $\pi'(s,a) = \epsilon/|A(s)|$，而对 $\forall s \in S$ 且 $a = \arg\max_b Q^{\pi'}(s,b)$ 有 $\pi'(s,a) = 1 - \epsilon + \epsilon/|A(s)|$ 成立。公式 (29) 表明，基于 ϵ-greedy 的策略改进符合策略改进定理，它的每一步都产生了改进，除非最优策略已经被找到了。On-Policy MC 算法类似于 MC ES 算法，只是第 (c) 步要改成 ϵ-greedy 策略，此处不再赘述。

6.2.4 Off-Policy MC 控制

On-Policy 方法的行为策略同时也是待改进的策略，而在 Off-Policy 方法中，行为策略与待改进策略是分开的，后者被称为估计策略。Off-Policy 方法的特点在于这两个策略可以是不相关的，估计策略可以是确定性的策略（例如贪心的改进策略等），而行为策略仍然是软策略，以确保维持探索条件成立。这种方法的一个潜在的问题是，它从一个阶段的尾部开始学习，即从最后一个非贪心动作后面开始。如果非贪心动作频繁出现，那么学习将变得很慢，特别是对在长阶段的早期就出现这种情况的状态来说，情况就更糟糕了。然而，目前还没有充分的经验来评估该问题的严重性。关于 Off-Policy MC 算法，它的主要部分是策略评估过程，这涉及到使用行为策略来产生阶段样本并估算估计策略的值函数这样一类技术；而

策略改进是简单的，它采用贪心的策略改进方法，算法的具体描述可参见文献 [2]。

6.3 时序差分方法

时序差分（Temporal-Difference, TD）学习方法是一种解决强化学习系统中时间信度分配问题的方法，它被认为是强化学习的核心算法之一。所谓时间信度分配问题就是指对涉及到的每个动作与状态赋予信任与责备 [25]。在很多实际问题中，一个动作的成功或失败需要一段时间以后才能知道，所以强化信号往往是以前的某个动作所引起的响应，或者说强化信号是与一个状态—动作序列相关的，这种情况被称为延时强化学习问题。因此，强化学习系统要能够对该状态—动作序列上的每个状态或状态—动作对赋予奖励或惩罚，以改进智能体的决策[7]。

早期的时序差分方法研究主要由 Samuel 和 Klopf[11,26,27] 做出；Holland[28,29] 对时序差分学习方法的研究做出了重要贡献，他研究的学习方法导致了一大批相关系统的产生，其中包括 Booker[30] 和 Holland[31] 等实现的系统，而这些相关于一个重要的算法——Sarsa（6.3.2节）。时序差分方法可以看成是 MC 方法与动态规划方法的结合。与 MC 方法一样，时序差分方法也不需要环境的模型；与动态规划方法一样，其值函数的更新方式也是步步为营的（Bootstrapping），即基于其它的已经学习到的估算值来更新，而不象 MC 方法那样，一定要等到一个阶段结束并产生了最终的回报才进行更新。与前面的两种方法一样，对时序差分方法的讨论也是从策略评估开始的，然后讨论它的策略改进及策略迭代。实际上，动态规划、MC 及时序差分方法的主要区别在于它们具有不同的策略评估方法。

6.3.1 基于时序差分的策略评估

本小节仅讨论单步 TD 评估（预测）方法，即 $TD(0)$。我们先来看看基于 MC 的 EV 评估方法，该方法需要等到一个阶段访问结束并得到完全回报后，再用该完全回报来更新值函数，公式如下：

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T \\ V(s_t) &\leftarrow V(s_t) + \alpha [R_t - V(s_t)] \end{aligned} \tag{30}$$

[7]还存在另外一种信度分配问题，即结构信度分配问题 [25]，它是指根据已知的状态信度来推测未知的状态信度，即泛化问题。特别是当问题的空间太大而不能完全搜索时，智能体必须具备泛化能力。

其中，T 是一个阶段的最后一个时间步，R_t 是自时刻 t 以来所获得的实际回报（被称为目标值），α 是一个常量步长参数，上述方法被称为常量—α MC 方法。可以看出，预测值 $V(s_t)$ 朝目标值的方向进行了调整，调整幅度由 α 来确定。与常量—α MC 方法不同，$TD(0)$ 方法不用等到一个阶段的结束，而只需等到下一个时间步结束，其相关的更新公式如下：

$$\begin{aligned} R_t &= r_{t+1} + \gamma V\left(s_{t+1}\right) \\ V\left(s_t\right) &\leftarrow V\left(s_t\right) + \alpha\left[R_t - V\left(s_t\right)\right] \end{aligned} \tag{31}$$

将公式 (30) 与公式 (31) 对比，可以看出，$TD(0)$ 方法用已经得到的估算 $\gamma V(s_{t+1})$ 取代了常量—α MC 方法中的 $\gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T$ 后续实际回报。因此，从某种意义上说，$TD(0)$ 方法是常量—α MC 方法的一个特例。特别地，公式 (31) 的处理方式可以自然地扩展到 n 步，从而可以形成 $TD(\lambda)$ 方法 [2]。

粗略地讲，时序差分方法合并了 MC 方法与动态规划方法，但它不需要环境模型，也不需要等到一个阶段结束之后才更新值函数。时序差分方法的一个优点在于它可以完全在线地执行，因为它在每个时间步都可以学习与预测。这个优点是十分明显的，因为许多实际应用具有较长的运行阶段，如果等到阶段结束之后才能进行预测，那么学习速度将会很慢；另外，还有一些应用是连续性任务，并不能划分成阶段。

下面给出 Tabular TD(0) 策略评估算法：

Initialize $V(s)$ *arbitrarily,* π *to the policy to be evaluated*

Repeat $(for\ each\ episode)$:

  Initialize s

  Repeat $(for\ each\ step\ of\ episode)$:

    $a \leftarrow$ *action given by* π for s

    Take action a; *observe reward* r, *and next state* s'

    $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$

    $s \leftarrow s'$

  until s *is terminal*

6.3.2 Sarsa: On-Policy TD 控制

Sarsa 方法是一种基于 TD 的广义策略迭代方法，它在形式上与 Q-learning 方法非常相似。Sarsa 方法由 Rummery 等人提出 [32]，它是一种 On-Policy TD 方法，它的行为策略与估计策略是一样的；而 Q-learning 是一种 Off-policy TD 方法，其行为策略是一个软策略，而估计策略是一个贪心策略。Sarsa 方法的策略评估使用下面的更新规则：

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha\left[r_{t+1} + \gamma Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right)\right] \tag{32}$$

其中，$r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ 是更新目标，α 是一个学习参数。上述更新规则使用了转移函数与奖励函数的所有元素：$s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$，Sarsa 方法由此而得名 [33]。Sarsa 方法的策略改进采用 ϵ-greedy 方法，为便于最后获得最优策略，可以让 ϵ 随时间逐步减小，可设 $\epsilon = 1/t$。事实上，对于 Q-learning 方法难以解决的问题，可以考虑使用 Sarsa 方法，并且在许多情况下，On-Policy 方法可能会优于 Off-Policy 方法 [2,25]。

下面给出 Sarsa(On-Policy TD) 策略控制算法：

Initialize $Q(s,a)$ *arbitrarily*

Repeat $(for\ each\ episode)$:

  Initialize s

  Choose a $from$ s *using policy derived from* $Q(e.g.,\ \epsilon - greedy)$

  Repeat $(for\ each\ step\ of\ episode)$:

    Take action a; *observe reward* r, *and next state* s'

    Choose a' $from$ s' *using policy derived from* $Q(e.g.,\ \epsilon - greedy)$

    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$

    $s \leftarrow s';\ a \leftarrow a';$

  until s *is terminal*

6.3.3 Q-learning: Off-Policy TD 控制

Q-learning 方法是强化学习领域的一个重大突破，它是一种基于 Off-Policy TD 的广义策略迭代方法，由 Watkins 于 1989 年提出 [19]，并证明了 Q-learning

方法的收敛性。其策略评估使用下面的更新规则：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{33}$$

其中，$r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ 是更新目标，α 是一个学习参数。从该公式可以看出，估计策略是一个贪心策略。Q-learning 方法的策略改进则可以采用任何软策略。

Watkins 和 Dayan[19,34] 证明了 Q-learning 方法在一定条件下的收敛性：

1. 环境具备 Markov 性质；
2. 用 Lookup 表来表示状态—动作值函数；
3. 每个状态—动作对被无限地访问；
4. 学习参数的正确选择；

而更加通用的收敛性证明由 Jaakkola 等人 [35] 及 Tsitsiklis[36] 给出。

下面给出 Q-learning(Off-Policy TD) 策略控制算法：

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* $Q(e.g.,\ \epsilon - greedy)$

Take action a; *observe reward* r, *and next state* 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*

7 Semi-Markov 决策过程

前面所建立起来的 MDP 框架实际上隐含了一个假设：所有的动作有一个均匀的执行时间，且必须在一个时间步之内完成。然而，对于大多数问题来讲，这显然是不合理的。从决策的角度来讲，单个时间步的粒度显然是太精细了，不利于智能体进行高层决策。例如，前锋在射门的时候是不会考虑大腿肌肉如何收缩

这一低级动作的。Semi-Markov 决策过程（Semi-Markov Decision Process, SMDP）[20,37] 扩展了 MDP 框架，它允许动作不必在一个时间步内完成，使得 SMDP 成为建模时间扩展动作（Temporally Extended Actions, TEA）的一个得力工具 [38]。这种建模特点非常有利于智能体形成一个合理的决策层次，使得智能体在进行高层决策时可以忽略低层动作，而在执行时可以让低层动作为其服务。实际上，这正是层次强化学习的一个基本思想。

7.1 SMDP 的定义

形式地，一个 SMDP 用一个五元组 $M = (S, A, \psi, T, R)$ 来表示。其中，S 表示环境状态集合，A 表示动作集合，$\psi \subseteq S \times A$ 是可接受的状态—动作对集合；$T : S \times A \times S \times N \to [0, 1]$ 是转移函数，可简写成 $T(s', N|s, a)$ 或 $T_{ss'}^{Na}(= \Pr\{s_{t+N} = s' \,|s_t = s, a_t = a\})$，它表示在状态 s 执行动作 a 并且在 N 个时间步转移到状态 s' 的概率；$R : S \times A \times N \to \mathbb{R}$ 是奖励函数，可简写成 $R(s, a, N)$ 或 $R_{ss'}^{Na}(= E\left\{\sum_{n=1}^{N} \gamma^{n-1} r_{t+n} \,|s_t = s, a_t = a, s_{t+N} = s'\right\})$，它表示在状态 s 执行动作 a 并且在 N 个时间步完成所获得的期望奖励值。此处，我们采用了 Dietterich[39] 的联合概率分布的建模方法，而传统的建模方法 [20] 则使用了两个概率分布来描述转移函数。

我们仍然用 A_s 或 $A(s)$（$= \{a|(s, a) \in \psi\}$）表示在状态 s 下可执行的动作集合，且 $A_s \subseteq A$。在上述定义中，转移函数与奖励函数都具有 Markov 性质，并且它们都是静态函数，即它们与时间 t 没有关系。对于转移函数来讲，有下面的公式成立：

$$\sum_{s' \in S, N \in \mathbb{N}} T\left(s', N \,|s, a\right) = 1.0 \tag{34}$$

SMDP 与 MDP 框架的主要差别在于转移函数。本质上，SMDP 框架建模了系统在关键决策点处的转移特性，而 MDP 框架则描述了系统所有时刻的转移特性 [5]，MDP 的建模粒度要比 SMDP 小。奖励函数也与 MDP 框架不同，一个动作在执行期间所获得的奖励可以累计起来并一直持续到该动作执行结束，期间所获得的全部奖励就作为该动作的回报。

7.2 值函数与 SMDP Bellman 方程

SMDP 的策略及不同形式的最优准则等概念与 MDP 框架中的一样。对于无限域 SMDP 来说，我们的目标仍然是要找到一个策略，它能够最大化期望的折扣总体回报。但是对于 SMDP 来说，由于动作可以持续多个时间步，所以值函数的定义必须要把这个因素考虑进去。

与 MDP 框架类似，首先需要定义历史事件概念。我们定义一个从时刻 0 到时刻 t 的历史事件为：$s_0, a_0, R_0, N_0, s_1, a_1, R_1, N_1, \ldots, s_t, a_t, R_t, N_t$，其中 s_i, a_i, R_i, N_i 分别为第 i 时刻的状态、动作、回报及持续的时间步数，把这四个元素合起来称为智能体的一个 SMDP 决策步。对于无限域 SMDP，我们采用下面的公式作为期望回报：

$$\Re_0 = R_0 + \gamma^{N_0} R_1 + \gamma^{N_0+N_1} R_2 + \gamma^{N_0+N_1+N_2} R_3 + \ldots = \sum_{t=0}^{\infty} \left(\gamma\left(t\right) \cdot R_t\right) \tag{35}$$

其中，$\gamma\left(t\right) = \prod_{i=0}^{t-1} \gamma^{N_i}$，并定义 $\gamma\left(0\right) = 1$。有了上面的定义，我们就可以定义策略 π 下状态 s 的值如下：

$$V^{\pi}\left(s\right) = E_{\pi}\left\{\Re_0 \left| s_0 = s \right.\right\} = E_{\pi}\left\{\sum_{t=0}^{\infty} \gamma\left(t\right) R_t \left| s_0 = s \right.\right\} \tag{36}$$

类似地，可以定义策略 π 下在状态 s 处执行动作 a 的值如下：

$$Q^{\pi}\left(s, a\right) = E_{\pi}\left\{\Re_0 \left| s_0 = s\ , a_0 = a \right.\right\} = E_{\pi}\left\{\sum_{t=0}^{\infty} \gamma\left(t\right) R_t \left| s_0 = s\ , a_0 = a \right.\right\} \tag{37}$$

现在，与 MDP 框架一样，我们也能够使用 Bellman 方程形式来表达 SMDP 中的

状态值函数与状态—动作对值函数了，公式如下：

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{\Re_0 | s_0 = s\} = E_{\pi}\left\{\sum_{t=0}^{\infty}\gamma(t) R_t | s_0 = s\right\} \\
&= E_{\pi}\left\{R_0 + \gamma(0)\sum_{t=1}^{\infty}\frac{\gamma(t)}{\gamma(0)} R_t | s_0 = s\right\} \\
&= \sum_a \pi(s,a) E_{\pi}\left\{R_0 + \gamma(0)\sum_{t=1}^{\infty}\frac{\gamma(t)}{\gamma(0)} R_t | s_0 = s, a_0 = a\right\} \\
&= \sum_a \pi(s,a)\sum_N E_{\pi}\left\{R_0 + \gamma^N \sum_{t=1}^{\infty}\frac{\gamma(t)}{\gamma^N} R_t | s_0 = s, a_0 = a, N_0 = N\right\} \\
&= \sum_a \pi(s,a)\sum_{s'N} T_{ss'}^{Na}\left(R_{ss'}^{Na} + \gamma^N E_{\pi}\left\{\sum_{t=1}^{\infty}\frac{\gamma(t)}{\gamma^N} R_t | s_1 = s'\right\}\right) \\
&= \sum_a \pi(s,a)\sum_{s'N} T_{ss'}^{Na}\left(R_{ss'}^{Na} + \gamma^N V^{\pi}(s')\right).
\end{aligned} \tag{38}$$

同理可得：

$$Q^{\pi}(s,a) = \sum_{s'N} T_{ss'}^{Na}\left(R_{ss'}^{Na} + \gamma^N V^{\pi}(s')\right) \tag{39}$$

与 MDP 框架类似，它们的 Bellman 最优方程如下：

$$\begin{aligned}
V^*(s) &= \max_a \sum_{s'N} T_{ss'}^{Na}\left(R_{ss'}^{Na} + \gamma^N V^*(s')\right) \\
Q^*(s,a) &= \sum_{s'N} T_{ss'}^{Na}\left(R_{ss'}^{Na} + \gamma^N \max_{a'} Q^*(s',a')\right)
\end{aligned} \tag{40}$$

7.3 SMDP 的基本方法

从上一小节可以看出，SMDP 的值函数与 MDP 的值函数极为相似。实际上，MDP 的基本方法也能够很容易地扩展到 SMDP 中。对于基于 SMDP 的动态规划方法，这种扩展是直接的，即直接使用 SMDP 值函数即可，并且 White[40] 与 Puterman[37] 已经表明这些算法是收敛的。对于在线学习算法，Bradtke 等人 [41] 提出了基于 SMDP 的 Q-learning 和时序差分算法，虽然他们没有能够证明其收敛性；而在最近，Parr[9] 提出了一种基于 SMDP 的 Q-learning 算法，并且证明了在满足一定条件的情况下该算法是收敛的，而这些条件基本上同常规 Q-learning 是一样的。其更新规则如下：

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha_t(s_t,a_t)\left[R_t + \gamma^N \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right] \tag{41}$$

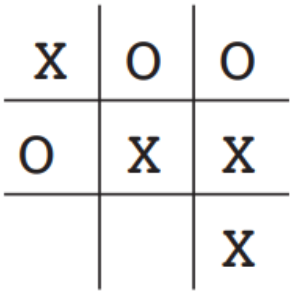

图 8-2: Tic-Tac-Toe 游戏

上面的公式与公式 (33) 稍有不同：R_t 表示智能体在状态 s_t 下执行动作 a_t N 步并转移到 s_{t+1} 的过程中所获得的回报；学习参数 a_t 已经被扩展到依赖于当前的状态 s_t 和动作 a_t；折扣率 γ 扩展成了 γ^N。

8 应用

8.1 Tic-Tac-Toe

Tic-Tac-Toe 是一种简单的双人棋类游戏，如图8-2所示。双方 (X 方与 O 方) 轮流下棋，X 方先下。如果一方在水平、垂直或对角线方向占据 3 个方格，则该方获胜，图8-2表明 X 方获胜。如果没有一方能够占据水平、垂直或对角线上的 3 个方格而棋盘被填满，则棋局为平局 (Draw)。

解决 Tic-Tac-Toe 棋类游戏的方法有很多，此处考虑使用强化学习的基础算法——On-Policy TD(0) 来解决。在算法实现时，需要为棋局终局定义奖励值。我们规定：

- 从智能体 X 方的角度定义胜负平得分，其奖励值分别为：X 方胜，奖励值为 1；X 方输，奖励值为 0；双方平，奖励值各为 0.5；

无论是 X 方，还是 O 方，均按照图8-3所示的方式更新状态值函数 $V(s)$。其中，实线表示棋局的实际走向；黑色实心圆点表示棋局，并在旁边标有棋局编号，带有 “*” 的编号表示当前的最佳选择；在棋局 b，智能体选取了最佳走步，棋局转换到 c^*，在棋局 d，智能体没有选取最佳走步，而是进行了随机探索，棋局转换到 e，在棋局 f，智能体再次选取最佳走步，棋局转换到 g^*；带有箭头的曲线表示值

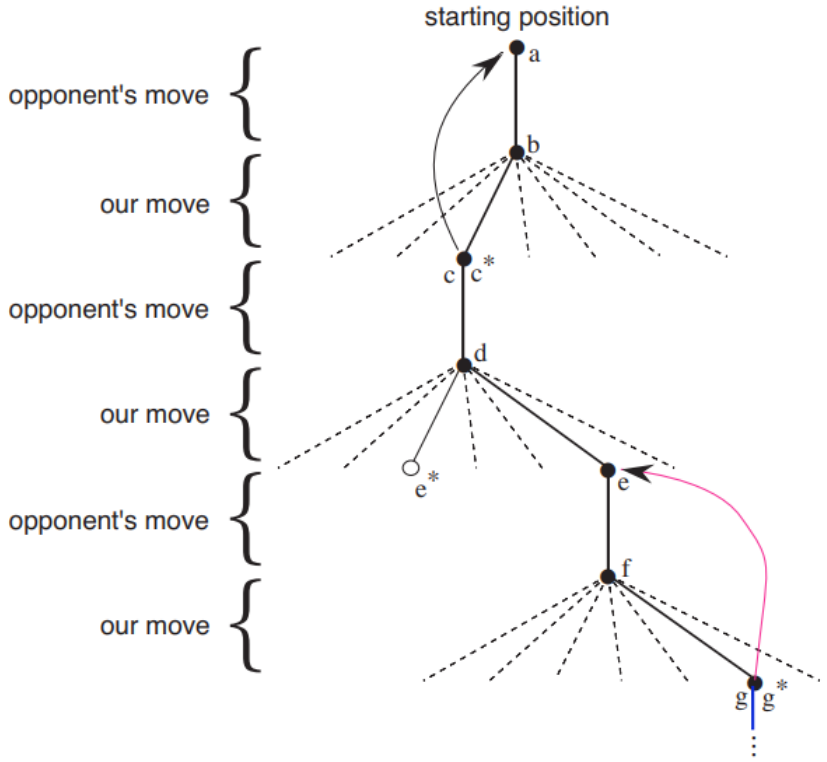

图 8-3: Tic-Tac-Toe 状态值函数的更新方式

更新的方向。注意，在探索阶段，棋局的状态值不会被更新。在利用阶段，棋局的状态值按如下公式更新：

$$V(s) \leftarrow V(s) + \alpha[V(s') - V(s)] \tag{42}$$

其中，α 为学习率或学习步长。初始时，所有的 $V(s)$ 初始化为 0.5，表示获胜概率为 50%。

下面给出 Tic-Tac-Toe 的 On-Policy TD(0)（ϵ-greedy 控制策略）算法：

```
Input:
    root: the root game state;
    V: all V(s) are initialized with 0.5;
    alpha: 0.5
    epsilon: 0.1
    learning_time: 10000
Output:
    best move (with V changed)
```

```
def TDLearning(root, V, alpha, epsilon, learning_time)
    seed()
    for i in range(learning_time):
        node = root.clone()
        parent = None
        result = node.isGameOver()
        while result == None:#node is nonterminal
            if random() < epsilon:
                Choose a move of node randomly
                node.play(move)
                result = node.isGameOver()
                if result != None:#node is terminal
                    UpdateTerminalNode(node, result)
                    if parent != None:
                        Update V(parent) with V(node) & alpha, by
                        ↪ TD(0) learning rule
                    parent = None
                else:
                    parent = node.clone()
            else:
                move = BestMove(node)
                node.play(move)
                result = node.isGameOver()
                if result != None:#node is terminal
                    UpdateTerminalNode(node, result)
                if parent != None:
                    Update V(parent) with V(node) & alpha, by TD(0)
                    ↪ learning rule
                if result != None:#node is terminal
                    parent = None
                else:
```

```
                parent = node.clone()
        if result == None:#node is nonterminal:
            Choose a move randomly for the opponent
            node.play(move)
            result = node.isGameOver()
    UpdateTerminalNode(node, result)
    if parent != None:
        Update V(parent) with V(node) & alpha, by TD(0) learning
        ↪ rule
return BestMove(root)

def UpdateTerminalNode(node, result):
    if V(node) exists:
        return
    if result is X win:
        V(node) = 1
    elif result is O win:
        V(node) = 0
    elif result is draw:
        V(node) = 0.5

#Exploring Start: in the beginning, all children should be
↪ selected uniformly, because each V(child) is 0.5
def BestMove(node):
    if node is X player:
        return the move to the child of node with the MAX
        ↪ V(child)
    else:
        return the move to the child of node with the MIN
        ↪ V(child)
```

具体的实现程序，请参阅本课程的相关文档。

9 练习

1. 实现 TicTacToe 的 TD(0) 对弈程序；

10 参考文献

1. Stuart J. Russell, Peter Norvig, 殷建平等译。《人工智能：一种现代的方法》，第 3 版，清华大学出版社。

2. Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. A Bradford Book, The MIT Press, Cambridge, Massachusetts, London, England. 1998.

3. Wikipedia: Reinforcement Learning.

4. Mahadevan, S. Average reward reinforcement learning: Foundations, algorithms, and empirical results. Machine Learning, 1996, pp. 159-196.

5. Mohammad Ghavamzadeh. Hierarchical Reinforcement Learning in Continuous State and Multi-Agent Environments. Ph.D. thesis. Department of Computer Science, University of Massachusetts Amherst, 2006.

6. Blackwell D. Discrete dynamic programming. Annals of Mathematical Statistics, 1962, 33, 719-726.

7. Bellman R. Dynamic Programming. Princeton University Press, 1957.

8. Bertsekas D. C., Tsitsiklis J. N. Parallel and Distributed Computation: Numerical Methods. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

9. Parr, R. Hierarchical Control and Learning for Markov Decision Processes. Ph.D. thesis, University of California, Berkeley, California, 1998.

10. Minsky M. L. Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ, 1967.

11. Samuel A. L. Some studies in machine learning using the game of checkers. IBM Journal on Research and Development, 1959, pages 210-229.

12. Andreae J. H. Learning machines—a unified view. Encyclopedia of Information, Linguistics, and Control, 1969, pages 261-270. Pergamon, Oxford.

13. Werbos, P. J. Advanced forecasting methods for global crisis warning and models of intelligence.General Systems Yearbook, 1977, 22:25-38.

14. Werbos, P. J. Applications of advances in nonlinear sensitivity analysis. In Drenick, R. F. and Kosin, F., editors, System Modeling an Optimization. Springer-Verlag. In Proceedings of the Tenth IFIP Conference, New York, 1981.

15. Werbos, P. J. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. IEEE Transactions on Systems, Man, and Cybernetics, 1987, pages 7-20.

16. Werbos, P. J. Generalization of back propagation with applications to a recurrent gas market model. Neural Networks, 1988, 1:339-356.

17. Werbos, P. J. Neural networks for control and system identification. In Proceedings of the 28th Conference on Decision and Control, 1989, pages 260-265, Tampa, Florida.

18. Werbos, P. Approximate dynamic programming for real-time control and neural modeling. In White, D. A. and Sofge, D. A., editors, Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches, 1992, pages 493-525. Van Nostrand Reinhold, New York.

19. Watkins C.J.C.H. Learning from Delayed Rewards. Ph.D. thesis, Cambridge University, Cambridge, England, 1989.

20. Howard, R. Dynamic Programming and Markov Processes. MIT Press, Cambridge, MA, 1960.

21. Puterman, M. L. and Shin, M. C. Modified policy iteration algorithms for discounted Markov decision problems. Management Science, 1978, 24:1127-1137.

22. Michie, D. and Chambers, R. A. BOXES: An experiment in adaptive control. In Dale, E. and Michie, D., editors, Machine Intelligence 1968, 2, pages 137-152. Oliver and Boyd.

23. Barto, A. G. and Duff, M. Monte carlo matrix inversion and reinforcement learning. In Cohen, J. D., Tsar, G., and Alspector, J., editors, Advances in Neural Information Processing Systems: In Proceedings of the 1993 Conference, pages 687-694, San Francisco, CA. Morgan Kaufmann.

24. Singh, S. P. and Sutton, R. S. Reinforcement learning with replacing eligibility traces. Machine Learning, 1996, 22:123-158.

25. 张汝波编著. 强化学习理论及应用. 哈尔滨：哈尔滨工程大学出版社, 2001 年 4 月第 1 版。

26. Samuel, A. L. Some studies in machine learning using the game of checkers. II–Recent progress. IBM Journal on Research and Development, 1967, pages 601-617.

27. Klopf, A. H. Brain function and adaptive systems–A heterostatic theory. Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA. A summary appears in Proceedings of the International Conference on Systems, Man, and Cybernetics, 1974, IEEE Systems, Man, and Cybernetics Society, Dallas, TX.

28. Holland, J. H. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.

29. Holland, J. H. Adaptation. In Rosen, R. and Snell, F. M., editors, Progress in Theoretical Biology, 1976, volume 4, pages 263-293. Academic Press, NY.

30. Booker, L. B. Intelligent Behavior as an Adaptation to the Task Environment. PhD thesis, University of Michigan, Ann Arbor, MI, 1982.

31. Holland, J. H. Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems. In Michalski, R. S., Carbonell, J.

G., and Mitchell, T. M., editors, Machine Learning: An Artificial Intelligence Approach, Volume II, 1986, pages 593-623. Morgan Kaufmann, San Mateo, CA.

32. Rummery, G. A. and Niranjan, M. On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

33. Richard S. Sutton. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. Advances in Neural Information Processing Systems. 1996, 8, pp.1038-1044, MIT Press.

34. Watkins, C. J. C. H. and Dayan, P. Q-learning. Machine Learning, 1992, 8:279-292.

35. Jaakkola, T., Jordan, M. I., and Singh, S. P. On the convergence of stochastic iterative dynamic programming algorithms. Neural Computation, 1994, 6.

36. Tsitsiklis, J. N. Asynchronous stochastic approximation and q-learning. Machine Learning, 1994, 16:185-202.

37. Puterman, M. Markov Decision Processes. Wiley Interscience, 1994.

38. Mahadevan S., Khaleeli N., and Marchalleck N. Designing agent controllers using discrete-event Markov models. In Proceedings of the AAAI Fall Symposium on Model-Directed Autonomous Systems, 1997.

39. T. G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. Journal of Artificial Intelligence Research, 2000, Volume 13, pages 227-303.

40. White, D. J. Markov Decision Processes. Wiley, New York, 1993.

41. Bradtke S. J., Duff M. O. Reinforcement learning methods for continuous-time Markov decision problems. In Advances in Neural Information Processing Systems, 1995, 7: In Proceedings of the 1994 Conference Denver, Colorado. MIT Press.