这一章我们开始学习强化学习（reinforcement learning）和适应性控制（adaptive control）。在监督学习中，算法的输出都在模仿训练集给出的标签 y；这种情况下，对于每个输入特征 x，都有一个对应的标签作为明确的 "正确答案"。与之相反，对于许多数学决策和控制问题，很难提供给学习算法一种**明确的显示监督**（explicit supervision）。例如，假如我们制作了一个 “四腿机器人”，然后要编程让它能走路，但我们并不知道怎么去采取 “正确” 的动作来进行四条腿的行走，所以就不能给它提供一个明确的监督学习算法来进行模仿。在强化学习的框架下，我们将仅为我们的算法提供一个奖励函数（reward function），该函数会告知学习算法何时表现良好，何时表现不佳。在 “四腿机器人” 中，奖励函数会在机器人进步的时候给出正面回馈，即奖励；在机器人退步或摔倒的时候给出负面回馈，即惩罚。然后，学习算法的工作就是判定如何选择动作已得到最大奖励。

强化学习（Reinforcement Learning，RL）已经成功用于多种场景了，例如无人直升机的自主飞行、机器人的腿部运动、手机的网络路由、市场营销的策略筛选、工厂控制、高效率的网页索引等等。我们对强化学习的探索，先从马尔可夫决策过程（Markov decision processes，MDP）开始，这个概念给出了强化学习问题中的常见形式。

1. 马尔可夫决策过程

1.1 马尔可夫决策过程

一个马尔可夫决策过程（Markov decision processes，MDP）由一个元组：$(S, A, \{P_{sa}\}, \gamma, R)$ 组成，其中的元素分别为：

- S 是一个**状态集合**（a set of **states**）。（例如，在无人直升机的案例中，S 就是直升机所有可能的位置和方向的集合。）
- A 是一个**动作集合**（a set of **actions**）。（例如，还以无人直升机为例，A 就是遥控器上能够操作的所有动作的集合）
- P_{sa} 为状态转移概率。对于每个状态 $s \in S$ 和动作 $a \in A$，P_{sa} 是一个在状态空间上的分布。简单地说，P_{sa} 给出的是在状态 s 下进行一个动作 a 之后转移到的**状态的分布**。
- $\gamma \in [0, 1)$ 叫做**折扣因子**（discount factor）。
- $R : S \times A \mapsto R$ 就是**奖励函数**（reward function）。（奖励函数也可以写成仅对状态 S 的函数，这样就可以写成 $R : S \mapsto R$）

马尔可夫决策过程的动态更新如下：在某个起始状态 s_0 启动，然后选择某个动作 $a_0 \in A$ 来执行 MDP。根据所选的动作会有对应的结果，即转移到某个后继状态 s_1，其服从分布 $s_1 \sim P_{s_0a_0}$。然后再选择另外一个动作 a_1，接下来又有对应这个动作的转移状态 $s_2 \sim P_{s_1a_1}$。接着选择一个动作 a_2，就这样进行下去。可以用下面的过程作为表示：

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

通过序列中的所有状态 $s_0, s_1, \ldots$ 和对应的动作 $a_0, a_1, \ldots$，我们就能得到总奖励值，即**总收益函数**（total payoff）为：

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \cdots$$

如果我们把奖励函数作为仅与状态相关的函数，那么就可以简化为：

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

多数情况下，我们都用后面这种仅为"状态" 的函数，虽然扩展到 "状态—动作" 的函数 $R(s, a)$ 也并不难。

强化学习的目标就是找到一组动作，使得总收益函数的期望值最大：

$$E[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots]$$

注意，在时间步长 t 上的奖励函数通过参数 γ^t 进行了衰减。因此，要使得期望最大化，就需要尽早获得奖励、而推迟惩罚的出现。在经济方面中，可以将 R 理解为盈利额，γ 理解为利率，这样可以用一种较为自然解释：今天的一美元就比明天的一美元更有价值。

1.2 策略、值函数

有一种**策略**（policy），是使用任意函数 $\pi : S \mapsto A$，从**状态**到**动作**进行**映射**。如果在状态 s，采取动作 $a = \pi(s)$，就可以说是正在执行策略 π。我们还可以针对策略 π 定义一个**值函数**（value function）：

$$\begin{aligned} V^\pi(s) &= E[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots | s_0 = s, \pi] \\ &= E[\sum_{i=0}^{\infty} \gamma^i R(s_i) | s_0 = s, \pi] \end{aligned}$$

$V^\pi(s)$ 就是从状态 s 开始，根据 π 给出的动作积累的折扣奖励（discounted rewards ）的期望和。

> 事实上我们在期望的表示中使用的记号 π，严格来说不太正确。因为 π 并不是一个随机变量，不过在很多文献里面都这样表示，已经成了某种标准。

如果我们用 s' 表示下一步状态，即 $s' \sim P_{s\pi(s)}$；由于概率分布归一化条件，即 $\sum_{s' \in S} P_{s\pi(s)} = 1$，则 $V^\pi(s)$ 还可以写为：

$$
\begin{aligned}
V^{\pi}(s) &= E[R(s_0)+\gamma R(s_1)+\gamma^2 R(s_2)+\cdots|s_0=s,\pi] && (1)\\
&= E[\sum_{i=0}^{\infty}\gamma^i R(s_i)|s_0=s,\pi] && (2)\\
&= E[R(s_0)+\gamma\sum_{i=1}^{\infty}\gamma^{(i-1)}R(s_i)|s_0=s,\pi] && (3)\\
&= E[R(s_0)+\gamma E[\sum_{i=1}^{\infty}\gamma^{(i-1)}R(s_i)]|s_0=s,\pi] && (4)\\
&= E[R(s_0)+\gamma V^{(\pi)}(s_1)|s_0=s,s_1=s',\pi] && (5)\\
&= \sum_{s'\in S}P_{s\pi(s)}\left(R(s)+\gamma V^{(\pi)}(s')\right) && (6)\\
&= R(s)+\gamma\sum_{s'\in S}P_{s\pi(s)}(s')V^{(\pi)}(s') && (7)
\end{aligned}
$$

其中步骤（3）到步骤（4）根据 **“奖励的期望等于奖励的期望的期望”** 可以推导出来；步骤（5）到步骤（6）相当于按期望的定义展开。针对等式（7），我们有一个专业的说法：给定一个固定的策略 π，其对应的值函数 $V^{\pi}(s)$ 满足贝尔曼等式（Bellman equations），即：

$$
V^{\pi}(s)=R(s)+\gamma\sum_{s'\in S}P_{s\pi(s)}(s')V^{(\pi)}(s')
$$

这也就意味着，从状态 s 开始的折扣奖励（discounted rewards）的期望和即**值函数** $V^{\pi}(s)$ 由两部分组成：1）是在状态 s 的时候立即获得的**奖励函数** $R(s)$，也就是上面式子的第一项；2）后续的折扣奖励的期望和乘以折扣因子。贝尔曼等式可以有效地解出 V^{π}，尤其是一个有限状态的 MDP 过程（$|S|<\infty$）。我们可以把每个状态 s 对应的 $V^{\pi}(s)$ 的方程写出来，这样就得到了一系列的 $|S|$ 个线性方程，有 $|S|$ 个变量，这些 $V^{\pi}(s)$ 都很容易解出来。

1.3 最优值函数、最佳策略

我们还可以定义**最优值函数**（optimal value function）：

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

换句话说，这个值是在所有可能的策略中能得到的最大的折扣奖励的期望和（每一个策略 $\pi(s)$，都会有有个值函数，最优值函数就是所有值函数中的最大的那个）。对于最优值函数，也有一个版本的贝尔曼等式：

$$
V^{*}(s)=R(s)+\max_{a\in A}\sum_{s'\in S}P_{sa}(s')V^{*}(s')
$$

上面这个等式中的第一项，还是跟之前的一样，还是即时奖励函数 $R(s)$；第二项是在所有的可能的动作中，后续的值函数的最大值。

另外还可以定义**最佳策略**：$\pi^{*}(s):S\mapsto A$

$$
\pi^{*}(s)=\arg\max_{a\in A}\sum_{s'\in S}P_{sa}(s')V^{*}(s')
$$

这里的 $\pi^{*}(s)$ 给出的动作 a 都能使贝尔曼等式的第二项取得最大值（这也是称为最佳策略的原因）。对于每个状态 s 和每个策略 π，都有：

$$V^*(s) = V^{\pi^*}(s) \geq V^{\pi}(s)$$重 点 ： 上 面 的 第 一 个 等 式 表 明 ： 最 佳 策 略

重点：上面的第一个等式表明：最佳策略 π^* 的值函数 $V^{\pi^*}(s)$ 等于它的最优值函数 $V^*(s)$。右边的不等式表明：最佳策略 π^* 生成的值函数是所有策略 π 的最大值，即最佳策略。

注意，这个最佳策略 π^* 有一个有趣的特性，它是所有状态 s 的最佳策略（个人理解：因为最佳策略的递归中有下一个状态 s'，所以解得的最佳策略不依赖于起始状态。我们不妨把问题简单化，即每个动作 a 之后得到的状态是确定的，而不是一个状态空间上的分布，那么我们总能找到一组最佳的动作，生成这组最佳动作的就是最佳策略 π^*。）具体来说，就是最佳策略 π^* 不会因为起始状态的不同而变化。这也就意味着无论马尔可夫决策过程的初始状态如何，都可以使用同样的策略函数 π^*。

2. 值迭代和策略迭代

现在我们开始讲解两种不同的算法，都能有效地解决有限状态的马尔可夫决策问题（finite-state MDPs）。目前为止，我们只考虑有限状态和动作空间的马尔可夫决策过程，即状态和动作的个数都是有限的，$|S| < \infty, |A| < \infty$。

2.1 值迭代（value iteration）

第一种算法，称为值迭代（value iteration），过程如下所述：

1. 对每个状态 s，初始化 $V(s) := 0$;
2. 重复直到收敛 {

   对每个状态 s，更新规则为：$V(s) := R(s) + \max_{a \in A} \gamma \sum_{s'} P_{sa}(s')V(s')$

   }

这个算法可以理解成，利用贝尔曼等式重复估计更新**值函数**。

在上述的算法的内部循环体中，有两种进行更新的方法。第一种，我们首先计算每个状态 s 的 $V(s)$ 的新值，然后统一用所有新值覆盖旧值。这也叫做**同步更新**（synchronous update）。在这种情况下，可以将该算法视为实现了一个 “贝尔曼备份” 操作符，该操作符接收值函数的当前估计值，并将其映射到新的估计值。第二种方法，就是按照某种次序遍历所有的状态，每计算一个状态，就更新该状态的值函数。这种方法也称为**异步更新**（asynchronous update）。值迭代算法收敛后的策略就是最佳策略。

2.2 策略迭代（policy iteration)

除了值迭代算法外，还有另外一种标准算法可以用来在马尔可夫决策过程（MDP）中寻找一个最佳策略，称为策略迭代（policy iteration），算法过程如下：

1. 随机初始化 π;
2. 重复直到收敛 {
   - 令 $V := V^{\pi}$;
   - 对每个状态 s，令 $\pi(s) = \arg\max_{a \in A} \sum_{s'} P_{sa}(s')V(s')$;

   }

因此，在循环体内就重复计算对于当前策略的值函数；然后使用当前的值函数进行计算，选择值最大的动作来更新策略。在上面的算法迭代了某个最大迭代次数后，V 将会收敛到 V^*，而 π 将会收敛到 π^*。

值迭代和策略迭代都是结局马尔可夫决策过程问题的标准算法，但目前对于这两个算法哪个更好，还没有一个统一的意见。对小规模的 MDPs（即状态空间较少）来说，策略迭代通常很快，很少的迭代次数就会收敛。然而，对于大规模状态空间的 MDPs，确切求解 V^π 常常涉及到求解一个非常大的线性方程组，可能非常困难。对于这种问题，就更倾向于选择值迭代。因此，在实际使用中，值迭代通常比策略迭代更加常用。

3. 马尔可夫决策过程的自学习模型

目前为止，我们讲的 MDPs 以及用于 MDPs 的一些算法，都是基于一个假设，即状态转移概率 P_{sa} 和奖励函数 R 已知。然而在很多显示问题中，却未必知道这两样，而是必须从数据中对其进行估计。（通常 S, A, γ 都是已知的）

例如，对于倒立摆问题，我们在这个 MDP 问题中进行了若干次实验，实验过程如下：

$$
\begin{gathered}
s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} s_3^{(1)} \xrightarrow{a_3^{(1)}} \ldots \\
s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} s_3^{(2)} \xrightarrow{a_3^{(2)}} \ldots \\
\ldots
\end{gathered}
$$

其中，$s_i^{(j)}$ 表示的是第 j 次实验中第 i 次的状态，而 $a_i^{(j)}$ 是该状态下的动作。在实践中，每个实验都会运行到 MDP 过程停止，或者会运行到某个大但有限的步数。

有了在 MDP 中一系列实验得到的 "经验"，就可以对状态转移概率推导出最大似然估计了：

$$
\begin{aligned}
P_{(sa)}(s') &= \frac{\#\text{times we took action a in state s and go to s'}}{\#\text{times we took action a in state s}} \\
&= \frac{\text{在状态 s 执行动作 a 而到达状态 s' 的次数}}{\text{在状态 s 执行动作 a 的总次数}}
\end{aligned}
$$

如果，上面的分式出现了 $0/0$ 的情况，对应的现实就是在状态 s 从没有进行过动作 a，这样我们就可以将状态转移概率简单估计为 $P_{sa}(s') = \frac{1}{|S|}$（即把下一状态 s' 的状态空间分布估计为均匀分布）。

值得注意的是，如果在 MDP 过程中我们能获得更多经验信息（更多的观察次数），就能利用新经验来更新估计的状态转移概率，这样很有效率。具体来说，如果我们保存上式的分子和分母的计数，那么观察到更多实验的时候，就可以很简单地累加这些计数值；然后计算比例，就能更新对 P_{sa} 的估计。利用类似的程序，如果奖励函数 R 未知，我们也可以选择在状态 s 下的期望即时奖励函数 $R(s)$ 来当做是在状态 s 观测到的平均奖励函数。

构建了一个马尔可夫决策过程的自学习模型后，我们可以采用值迭代或者策略迭代的方法，利用估计的状态转移概率和奖励函数，来求解这个 MDP 问题。例如，结合模型学习和值迭代，就可以在未知状态转移概率的情况下对 MDP 进行学习，下面就是一种可行的算法：

1. 随机初始化 π；
2. 重复 {
    - 在 MDP 中执行 π 作为若干次实验。
    - 利用上面在 MDP 积累的经验，更新对状态转移概率 P_{sa} 的估计（如果可以的话，也对奖励函数 R 进行更新）；
    - 利用估计的状态转移概率和奖励函数，应用值迭代，得到一个新的估计的值函数 V；
    - 更新 π 为与 V 对应的贪婪策略。

   }

我们注意到， 对于这个特定的算法，有一种简单的优化方法可以让该算法运行得更快。具体来说，在我们应用值迭代的内部循环中，如果不使用 $V = 0$ 初始化迭代，而是使用在我们的算法的前一次迭代中找到的值来初始化它，那么就提供了一个更好的迭代起点，能让算法更快收敛。

4. 备注

事实上，在机器学习中，根据马尔可夫性（即无后效性），有多种马尔可夫子模型，整理如下：

	不考虑动作	**考虑动作**
状态完全可见	马尔可夫链（MC）	马尔可夫决策过程（MDP）
状态部分可见	隐马尔可夫模型（HMM）	部分可观察马尔可夫决策过程（POMDP）