



蘑菇书

Easy RL

Qi Wang Yiyuan Yang Ji Jiang

<https://github.com/datawhalechina/easy-rl>

版本: 1.0.3

2022 年 3 月 1 日



前言

李宏毅老师的《深度强化学习》是强化学习领域经典的中文视频之一。李老师幽默风趣的上课风格让晦涩难懂的强化学习理论变得轻松易懂，他会通过很多有趣的例子来讲解强化学习理论。比如老师经常会用玩 Atari 游戏的例子来讲解强化学习算法。此外，为了教程的完整性，我们整理了周博磊老师的《强化学习纲要》、李科尧老师的《世界冠军带你从零实践强化学习》以及多个强化学习的经典资料作为补充。对于想入门强化学习又想看中文讲解的人来说绝对是非常推荐的。

本教程也称为“蘑菇书”，寓意是希望此书能够为读者注入活力，让读者“吃”下这本蘑菇之后，能够饶有兴致地探索强化学习，像马里奥那样愈强大，继而在人工智能领域觅得意外的收获。

使用说明

- 第 4 章到第 11 章为李宏毅《深度强化学习》的部分；
- 第 1 章和第 2 章根据《强化学习纲要》整理而来；
- 第 3 章和第 12 章根据《世界冠军带你从零实践强化学习》整理而来。

在线阅读地址: <https://datawhalechina.github.io/easy-rl/> (内容实时更新)

最新版 PDF 获取地址: <https://github.com/datawhalechina/easy-rl/releases>

编委会

编委: Qi Wang, Yiyuan Yang, Ji Jiang

致谢

特别感谢 Sm1les、LSGOMYP 对本项目的帮助与支持。

扫描下方二维码，然后回复关键词“强化学习”，即可加入“Easy-RL 读者交流群”



Datawhale
一个专注于 AI 领域的开源组织

版权声明

本作品采用知识共享署名-非商业性使用-相同方式共享 4.0 国际许可协议进行许可。

主要符号表

a 标量

\boldsymbol{a} 向量

\boldsymbol{A} 矩阵

\mathbb{R} 实数集

$\arg \max_a f(a)$ $f(a)$ 取最大值时 a 的值

s 状态

a 动作

r 奖励

π 策略

$\pi(s)$ 根据确定性策略 π 在状态 s 选取的动作

γ 折扣因子

τ 轨迹

$V_\pi(s)$ 状态 s 在策略 π 下的价值

$Q_\pi(s, a)$ 状态 s 在策略 π 下采取动作 a 的价值

G_t 时刻 t 时的回报

π_θ 参数 θ 对应的策略

$J(\theta)$ 策略 π_θ 的性能度量

目录

第 1 章 绪论	1
1.1 强化学习概述	1
1.1.1 强化学习与监督学习	1
1.1.2 强化学习的例子	3
1.1.3 强化学习的历史	4
1.1.4 强化学习的应用	6
1.2 序列决策介绍	7
1.2.1 智能体和环境	7
1.2.2 奖励	7
1.2.3 序列决策	7
1.3 动作空间	8
1.4 强化学习智能体的组成成分和类型	9
1.4.1 策略	9
1.4.2 价值函数	9
1.4.3 模型	10
1.4.4 强化学习智能体的类型	11
1.5 学习与规划	13
1.6 探索和利用	14
1.7 强化学习实验	15
1.7.1 Gym	15
1.7.2 MountainCar-v0 例子	18
1.8 关键词	20
1.9 习题	21
1.10 面试题	21
第 2 章 马尔可夫决策过程	22
2.1 马尔可夫过程	22
2.1.1 马尔可夫性质	22
2.1.2 马尔可夫过程/马尔可夫链	22
2.1.3 马尔可夫过程的例子	23
2.2 马尔可夫奖励过程	23
2.2.1 回报与价值函数	24
2.2.2 贝尔曼方程	25
2.2.3 计算马尔可夫奖励过程价值的迭代算法	27
2.2.4 马尔可夫奖励过程的例子	28
2.3 马尔可夫决策过程	29
2.3.1 马尔可夫决策过程中的策略	29
2.3.2 马尔可夫决策过程和马尔可夫过程/马尔可夫奖励过程的区别	29
2.3.3 马尔可夫决策过程中的价值函数	30
2.3.4 贝尔曼期望方程	30
2.3.5 备份图	31
2.3.6 策略评估	33
2.3.7 预测与控制	34

2.3.8 动态规划	35
2.3.9 马尔可夫决策过程中的策略评估	35
2.3.10 马尔可夫决策过程控制	37
2.3.11 策略迭代	38
2.3.12 价值迭代	40
2.3.13 策略迭代与价值迭代的区别	42
2.3.14 马尔可夫决策过程中的预测和控制总结	44
2.4 关键词	44
2.5 习题	45
2.6 面试题	46
第 3 章 表格型方法	47
3.1 马尔可夫决策过程	47
3.1.1 有模型	47
3.1.2 免模型	48
3.1.3 有模型与免模型的区别	48
3.2 Q 表格	49
3.3 免模型预测	52
3.3.1 蒙特卡洛策略评估	52
3.3.2 时序差分	54
3.3.3 动态规划方法、蒙特卡洛方法以及时序差分方法的自举和采样	57
3.4 免模型控制	58
3.4.1 Sarsa: 同策略时序差分控制	61
3.4.2 Q 学习: 异策略时序差分控制	62
3.4.3 同策略与异策略的区别	65
3.5 使用 Q 学习解决悬崖寻路问题	66
3.5.1 CliffWalking-v0 环境简介	66
3.5.2 强化学习基本接口	67
3.5.3 Q 学习算法	68
3.5.4 结果分析	68
3.6 关键词	69
3.7 习题	70
3.8 面试题	70
第 4 章 策略梯度	71
4.1 策略梯度算法	71
4.2 策略梯度实现技巧	76
4.2.1 技巧 1: 添加基线	77
4.2.2 技巧 2: 分配合适的分数	78
4.3 REINFORCE: 蒙特卡洛策略梯度	79
4.4 关键词	82
4.5 习题	83
4.6 面试题	83

第 5 章 近端策略优化	84
5.1 从同策略到异策略	84
5.2 近端策略优化	87
5.2.1 近端策略优化惩罚	88
5.2.2 近端策略优化裁剪	89
5.3 关键词	90
5.4 习题	91
5.5 面试题	91
第 6 章 深度 Q 网络	92
6.1 状态价值函数	92
6.2 动作价值函数	95
6.3 目标网络	98
6.4 探索	100
6.5 经验回放	101
6.6 深度 Q 网络	102
6.7 关键词	103
6.8 习题	103
6.9 面试题	104
第 7 章 深度 Q 网络进阶技巧	105
7.1 双深度 Q 网络	105
7.2 竞争深度 Q 网络	106
7.3 优先级经验回放	108
7.4 在蒙特卡洛方法和时序差分方法中取得平衡	108
7.5 噪声网络	109
7.6 分布式 Q 函数	110
7.7 彩虹	110
7.8 使用深度 Q 网络解决推车杆问题	112
7.8.1 CartPole-v0 简介	112
7.8.2 深度 Q 网络基本接口	113
7.8.3 回放缓冲区	114
7.8.4 Q 网络	115
7.8.5 深度 Q 网络算法	115
7.8.6 结果分析	116
7.9 关键词	116
7.10 习题	118
7.11 面试题	118
第 8 章 针对连续动作的深度 Q 网络	119
8.1 方案 1: 对动作进行采样	119
8.2 方案 2: 梯度上升	119
8.3 方案 3: 设计网络架构	119
8.4 方案 4: 不使用深度 Q 网络	120
8.5 习题	120

第 9 章 演员-评论员算法	121
9.1 策略梯度回顾	121
9.2 深度 Q 网络回顾	122
9.3 演员-评论员算法	122
9.4 优势演员-评论员算法	123
9.5 异步优势演员-评论员算法	124
9.6 路径衍生策略梯度	125
9.7 与生成对抗网络的联系	128
9.8 关键词	128
9.9 习题	128
9.10 面试题	128
第 10 章 稀疏奖励	130
10.1 设计奖励	130
10.2 好奇心	131
10.3 课程学习	132
10.4 分层强化学习	134
10.5 关键词	135
10.6 习题	136
第 11 章 模仿学习	137
11.1 行为克隆	137
11.2 逆强化学习	139
11.3 第三人称视角模仿学习	142
11.4 序列生成和聊天机器人	143
11.5 关键词	143
11.6 习题	144
第 12 章 深度确定性策略梯度	145
12.1 离散动作与连续动作的区别	145
12.2 深度确定性策略梯度	146
12.3 双延迟深度确定性策略梯度	148
12.4 使用深度确定性策略梯度解决倒立摆问题	150
12.4.1 Pendulum-v0 简介	150
12.4.2 深度确定性策略梯度基本接口	150
12.4.3 Ornstein-Uhlenbeck 噪声	151
12.4.4 深度确定性策略梯度算法	152
12.4.5 结果分析	153
12.5 关键词	154
12.6 习题	154
12.7 面试题	154
第 13 章 AlphaStar 论文解读	155
13.1 AlphaStar 以及背景简介	155
13.2 AlphaStar 的模型输入和输出是什么呢? —— 环境设计	155
13.2.1 状态 (网络的输入)	155

13.2.2 动作（网络的输出）	156
13.3 AlphaStar 的计算模型是什么呢？——网络结构	156
13.3.1 输入部分	157
13.3.2 中间过程	157
13.3.3 输出部分	157
13.4 庞大的 AlphaStar 如何训练呢？——学习算法	158
13.4.1 监督学习	158
13.4.2 强化学习	158
13.4.3 模仿学习	159
13.4.4 多智能体学习/自学习	160
13.5 AlphaStar 实验结果如何呢？——实验结果	160
13.5.1 宏观结果	160
13.5.2 其他实验（消融实验）	161
13.6 关于 AlphaStar 的总结	161
附录 A 习题解答	162
附录 B 面试题解答	173

第 1 章 绪论

1.1 强化学习概述

强化学习 (reinforcement learning, RL) 讨论的问题是智能体 (agent) 怎么在复杂、不确定的环境 (environment) 里面去最大化它能获得的奖励。如图 1.1 所示，强化学习由两部分组成：智能体和环境。在强化学习过程中，智能体与环境一直在交互。智能体在环境里面获取某个状态后，它会利用该状态输出一个动作 (action)，这个动作也称为决策 (decision)。然后这个动作会在环境之中被执行，环境会根据智能体采取的动作，输出下一个状态以及当前这个动作带来的奖励。智能体的目的就是尽可能多地从环境中获取奖励。

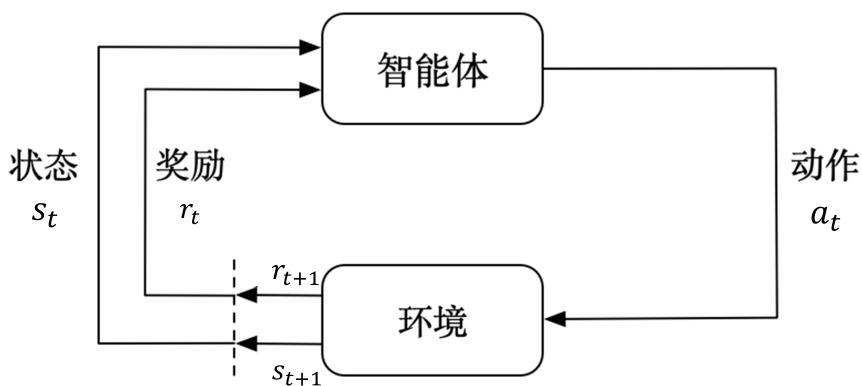


图 1.1 强化学习示意

1.1.1 强化学习与监督学习

我们可以把强化学习与监督学习做一个对比。以图片分类为例，如图 1.2 所示，**监督学习 (supervised learning)** 首先假设我们有大量被标注的数据，比如汽车、飞机、椅子这些被标注的图片，这些图片都要满足独立同分布，即它们之间是没有关联关系的。假设我们训练一个分类器，比如神经网络。为了分辨输入的图片是汽车还是飞机，在训练过程中，我们需要把正确的标签信息传递给神经网络。当神经网络做出错误的预测时，比如现在输入了汽车的图片，它预测出来是飞机，我们就会直接告诉它，该预测是错误的，正确的标签应该是汽车。最后我们根据这个错误写出一个损失函数 (loss function)，通过反向传播 (back propagation) 来训练神经网络。

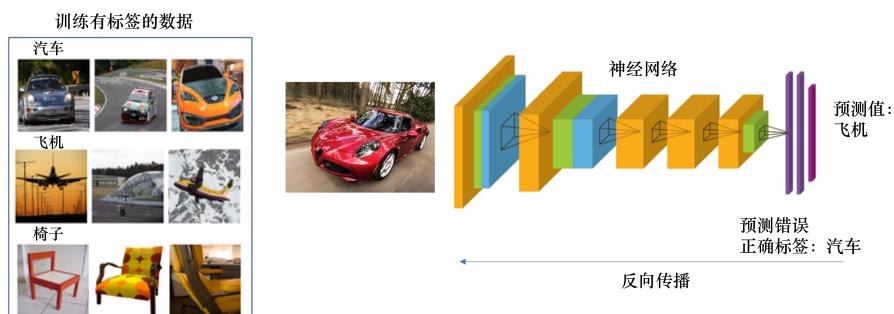


图 1.2 监督学习

通常假设样本空间中全体样本服从一个未知分布，每个样本都是独立地从这个分布上采样获得的，即独立同分布 (independent and identically distributed, i.i.d.)。

所以在监督学习过程中，有两个假设。第一，输入的数据（标注的数据）都应是没有关联的。因为如果输入的数据有关联，学习器（learner）是不好学习的；第二，我们告诉学习器正确的标签是什么，这样它可以通过正确的标签来修正自己的预测。

在强化学习里面，监督学习的两个假设其实都不满足。以雅达利（Atari）游戏 *Breakout* 为例，如图 1.3 所示，这是一个打砖块的游戏，控制木板左右移动把球反弹到上面来消除砖块。在玩游戏的过程中，我们可以发现智能体得到的观测（observation）不是独立同分布的，上一帧与下一帧间其实有非常强的连续性。我们得到的数据是相关的时间序列数据，不满足独立同分布。另外，我们并没有立刻获得反馈，游戏没有告诉我们哪个动作是正确动作。比如我们现在把木板往右移，这只会使得球往上或者往左去一点儿，我们并不会得到立刻的反馈。因此，强化学习之所以这么困难，是因为智能体不能得到即时的反馈，然而我们依然希望智能体在这个环境里面学习。



图 1.3 雅达利游戏 *Breakout*

如图 1.4 所示，强化学习的训练数据就是一个玩游戏的过程。我们从第 1 步开始，采取一个动作，比如我们把木板往右移，接到球。第 2 步我们又做出动作，得到的训练数据是一个玩游戏的序列。比如现在是在第 3 步，我们把这个序列放进网络，希望网络可以输出一个动作，即在当前的状态应该输出往右移或者往左移。这里有个问题，我们没有标签来说明现在这个动作是正确还是错误的，必须等到游戏结束才可能知道，这个游戏可能 10s 后才结束。现在这个动作到底对最后游戏是否能赢有无帮助，我们其实是不清楚的。这里我们就面临延迟奖励（delayed reward）的问题，延迟奖励使得训练网络非常困难。

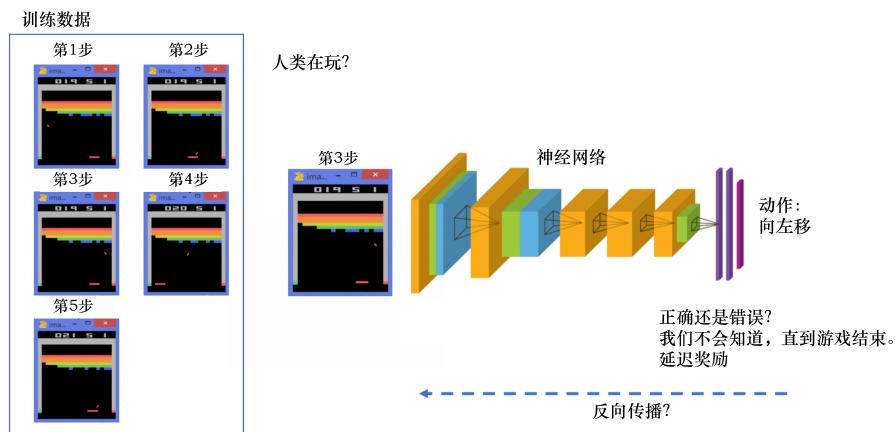


图 1.4 强化学习：玩 *Breakout*

强化学习和监督学习的区别如下。

- (1) 强化学习输入的样本是序列数据，而不像监督学习里面样本都是独立的。

(2) 学习器并没有告诉我们每一步正确的动作应该是什么，学习器需要自己去发现哪些动作可以带来最多的奖励，只能通过不停地尝试来发现最有利的动作。

(3) 智能体获得自己能力的过程，其实是不断地试错探索 (trial-and-error exploration) 的过程。探索 (exploration) 和利用 (exploitation) 是强化学习里面非常核心的问题。其中，探索指尝试一些新的动作，这些新的动作有可能会使我们得到更多的奖励，也有可能使我们“一无所有”；利用指采取已知的可以获得最多奖励的动作，重复执行这个动作，因为我们知道这样做可以获得一定的奖励。因此，我们需要在探索和利用之间进行权衡，这也是在监督学习里面没有的情况。

(4) 在强化学习过程中，没有非常强的监督者 (supervisor)，只有奖励信号 (reward signal)，并且奖励信号是延迟的，即环境会在很久以后告诉我们之前我们采取的动作到底是不是有效的。因为我们没有得到即时反馈，所以智能体使用强化学习来学习就非常困难。当我们采取一个动作后，如果我们使用监督学习，我们就可以立刻获得一个指导，比如，我们现在采取了一个错误的动作，正确的动作应该是什么。而在强化学习里面，环境可能会告诉我们这个动作是错误的，但是它并没有告诉我们正确的动作是什么。而且更困难的是，它可能是在一两分钟过后告诉我们这个动作是错误的。所以这也是强化学习和监督学习不同的地方。

通过与监督学习的比较，我们可以总结出强化学习的一些特征。

(1) 强化学习会试错探索，它通过探索环境来获取对环境的理解。

(2) 强化学习智能体会从环境里面获得延迟的奖励。

(3) 在强化学习的训练过程中，时间非常重要。因为我们得到的是有时间关联的数据 (sequential data)，而不是独立同分布的数据。在机器学习中，如果观测数据有非常强的关联，会使得训练非常不稳定。这也是为什么在监督学习中，我们希望数据尽量满足独立同分布，这样就可以消除数据之间的相关性。

(4) 智能体的动作会影响它随后得到的数据，这一点是非常重要的。在训练智能体的过程中，很多时候我们也是通过正在学习的智能体与环境交互来得到数据的。所以如果在训练过程中，智能体不能保持稳定，就会使我们采集到的数据非常糟糕。我们通过数据来训练智能体，如果数据有问题，整个训练过程就会失败。所以在强化学习里面一个非常重要的问题就是，怎么让智能体的动作一直稳定地提升。

1.1.2 强化学习的例子

为什么我们关注强化学习，其中非常重要的一个原因就是强化学习得到的模型可以有超人类的表现。监督学习获取的监督数据，其实是人来标注的，比如 ImageNet 的图片的标签都是人类标注的。因此我们可以确定监督学习算法的上限 (upper bound) 就是人类的表现，标注结果决定了它的表现永远不可能超越人类。但是对于强化学习，它在环境里面自己探索，有非常大的潜力，它可以得到超越人类的能力的表现，比如 DeepMind 的 AlphaGo 这样一个强化学习的算法可以把人类顶尖的棋手打败。

这里给大家举一些在现实生活中强化学习的例子。

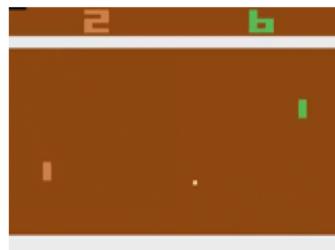
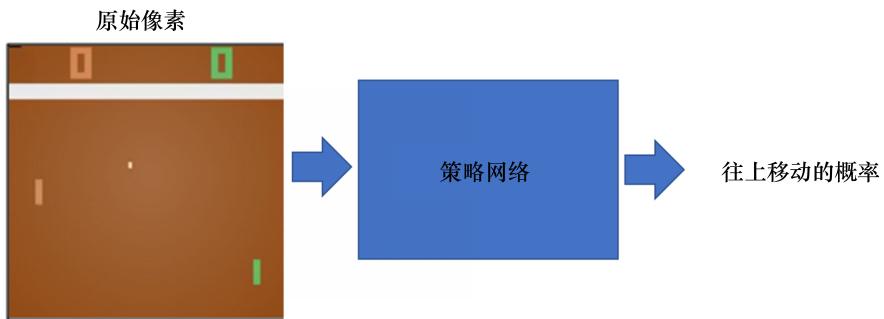
(1) 在自然界中，羚羊其实也在做强化学习。它刚刚出生的时候，可能都不知道怎么站立，然后它通过试错，一段时间后就可以跑得很快，可以适应环境。

(2) 我们也可以把股票交易看成强化学习的过程。我们可以不断地买卖股票，然后根据市场给出的反馈来学会怎么去买卖可以让我们的奖励最大化。

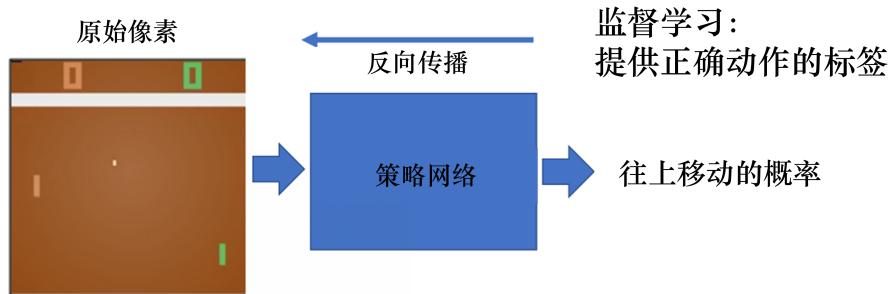
(3) 玩雅达利游戏或者其他电脑游戏，也是一个强化学习的过程，我们可以通过不断试错来知道怎么玩才可以通关。

图 1.5 所示为强化学习的一个经典例子，即雅达利的 *Pong* 游戏。游戏中右边的选手把球拍到左边，然后左边的选手需要把球拍到右边。训练好的强化学习智能体和正常的选手有区别：强化学习的智能体会一直做无意义的振动，而正常的选手不会做出这样的动作。

在 *Pong* 游戏里面，其实只有两个动作：往上或者往下。如图 1.6 所示，如果强化学习通过学习一个策略网络来进行分类，那么策略网络会输入当前帧的图片，输出所有决策的可能性，比如往上移动的概率。

图 1.5 *Pong* 游戏图 1.6 强化学习玩 *Pong*

如图 1.7 所示，对于监督学习，我们可以直接告诉智能体正确动作的标签是什么。但在 *Pong* 游戏中，我们并不知道它的正确动作的标签是什么。

图 1.7 监督学习玩 *Pong*

在强化学习里面，我们让智能体尝试玩 *Pong* 游戏，对动作进行采样，直到游戏结束，然后对每个动作进行惩罚。图 1.8 所示为预演（rollout）的一个过程。预演是指我们从当前帧对动作进行采样，生成很多局游戏。我们将当前的智能体与环境交互，会得到一系列观测。每一个观测可看成一个轨迹（trajectory）。轨迹就是当前帧以及它采取的策略，即状态和动作的序列：

$$\tau = (s_0, a_0, s_1, a_1, \dots) \quad (1.1)$$

最后结束时，我们会知道到底有没有把这个球拍到对方区域，对方有没有接住，我们是赢了还是输了。我们可以通过观测序列以及最终奖励（eventual reward）来训练智能体，使它尽可能地采取可以获得最终奖励的动作。一场游戏称为一个回合（episode）或者试验（trial）。

1.1.3 强化学习的历史

强化学习是有一定的历史的，早期的强化学习，我们称其为标准强化学习。最近业界把强化学习与深度学习结合起来，就形成了深度强化学习（deep reinforcement learning），因此，深度强化学习 = 深度

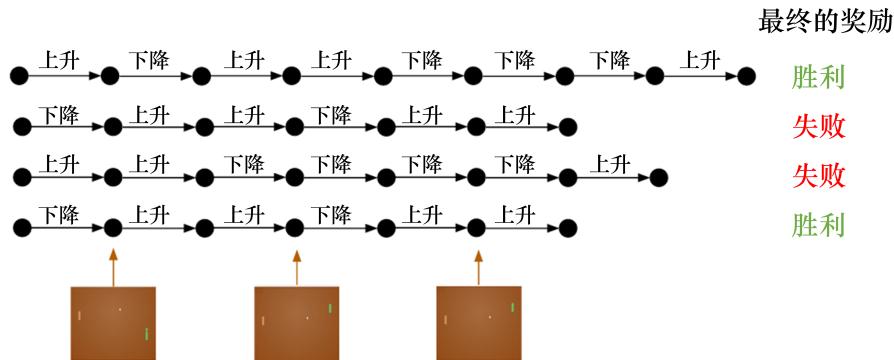


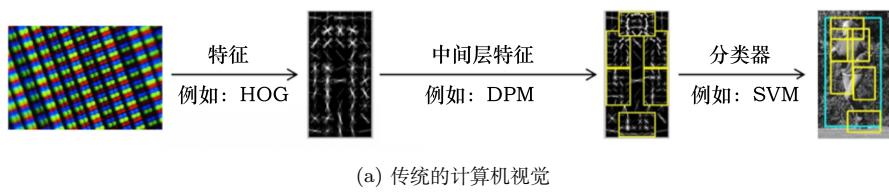
图 1.8 可能的预演序列

学习 + 强化学习。我们可将标准强化学习和深度强化学习类比于传统的计算机视觉和深度计算机视觉。

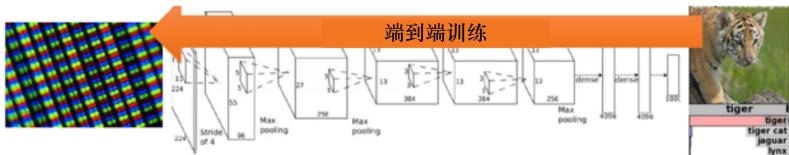
如图 1.9a 所示，传统的计算机视觉由两个过程组成。

(1) 给定一张图片，我们先要提取它的特征，使用一些设计好的特征，比如方向梯度直方图 (histogram of oriental gradient, HOG)、可变现的组件模型 (deformable part model, DPM)。

(2) 提取这些特征后，我们再单独训练一个分类器。这个分类器可以是支持向量机 (support vector machine, SVM) 或 Boosting，然后就可以辨别这张图片是狗还是猫。



(a) 传统的计算机视觉



(b) 深度计算机视觉

图 1.9 传统的计算机视觉与深度计算机视觉的区别

2012 年之后，有了卷积神经网络，大家就把特征提取以及分类这两个过程合并了。我们训练一个神经网络，这个神经网络既可以做特征提取，也可以做分类，它可以实现端到端训练，如图 1.9b 所示，它的参数可以在每一个阶段都得到极大的优化，这是一个非常重要的突破。

我们可以把神经网络放到强化学习里面。

- 标准强化学习：比如 TD-Gammon 玩 Backgammon 游戏的过程，其实就是设计特征，然后训练价值函数的过程，如图 1.10a 所示。标准强化学习先设计很多特征，这些特征可以描述现在整个状态。得到这些特征后，我们就可以通过训练一个分类网络或者分别训练一个价值估计函数来采取动作。
- 深度强化学习：自从我们有了深度学习，有了神经网络，就可以把智能体玩游戏的过程改进成一个端到端训练 (end-to-end training) 的过程，如图 1.10b 所示。我们不需要设计特征，直接输入状态就可以输出动作。我们可以用一个神经网络来拟合价值函数或策略网络，省去特征工程 (feature engineering) 的过程。

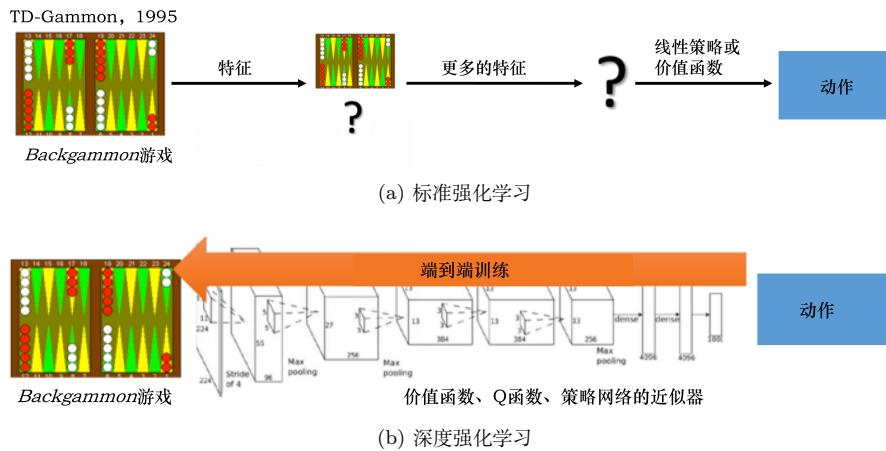


图 1.10 标准强化学习与深度强化学习的区别

1.1.4 强化学习的应用

为什么强化学习在这几年有很多的应用，比如玩游戏以及机器人的一些应用，并且可以击败人类的顶尖棋手呢？这有如下几点原因。首先，我们有了更多的算力（computation power），有了更多的 GPU，可以更快地做更多的试错尝试。其次，通过不同尝试，智能体在环境里面获得了很多信息，然后可以在环境里面取得很大的奖励。最后，我们通过端到端训练把特征提取和价值估计或者决策一起优化，这样就可以得到一个更强的决策网络。

接下来介绍一些强化学习里面比较有意思的例子，如图 1.11 所示。

(1) DeepMind 研发的走路的智能体。这个智能体往前走一步，就会得到一个奖励。这个智能体有不同的形态，可以学到很多有意思的功能。比如，像人一样的智能体学习怎么在曲折的道路上往前走。结果非常有意思，这个智能体会把手举得非常高，因为举手可以让它的身体保持平衡，它就可以更快地在环境里面往前走。而且我们也可以增加环境的难度，加入一些扰动，智能体就会变得更鲁棒。

(2) 机械臂抓取。因为我们把强化学习应用到机械臂自动抓取需要大量的预演，所以我们可以使用多个机械臂进行训练。分布式系统可以让机械臂尝试抓取不同的物体，盘子里面物体的形状是不同的，这样就可以让机械臂学到一个统一的动作，然后针对不同的抓取物都可以使用最优的抓取算法。因为抓取的物体形状的差别很大，所以使用一些传统的抓取算法不能把所有物体都抓起来。传统的抓取算法对每一个物体都需要建模，这样是非常费时的。但通过强化学习，我们可以学到一个统一的抓取算法，其适用于不同的物体。

(3) OpenAI 的机械臂翻魔方。OpenAI 在 2018 年的时候设计了一款带有“手指”的机械臂，它可以通过翻动手指使得手中的木块达到预期的设定。人的手指其实非常灵活，怎么使得机械臂的手指也具有这样灵活的能力一直是个问题。OpenAI 先在一个虚拟环境里面使用强化学习对智能体进行训练，再把它应用到真实的机械臂上。这在强化学习里面是一种比较常用的做法，即我们先在虚拟环境里面得到一个很好的智能体，然后把它应用到真实的机器人中。这是因为真实的机械臂通常非常容易坏，而且非常贵，一般情况下没办法大批量地购买。OpenAI 在 2019 年对其机械臂进行了进一步的改进，这个机械臂在改进后可以玩魔方了。

(4) 穿衣服的智能体。很多时候我们要在电影或者一些动画中实现人穿衣服的场景，通过手写执行命令让机器人穿衣服非常困难，穿衣服也是一种非常精细的操作。我们可以训练强化学习智能体来实现穿衣服功能。我们还可以在里面加入一些扰动，智能体可以抵抗扰动。可能会有失败的情况（failure case）出现，这样智能体就穿不进去衣服。

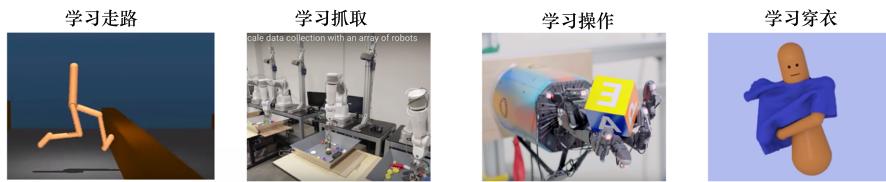


图 1.11 强化学习例子

1.2 序列决策介绍

1.2.1 智能体和环境

接下来我们介绍序列决策 (sequential decision making) 过程。强化学习研究的问题是智能体与环境交互的问题，图 1.12 左边的智能体一直在与图 1.12 右边的环境进行交互。智能体把它的动作输出给环境，环境取得这个动作后会进行下一步，把下一步的观测与这个动作带来的奖励返还给智能体。这样的交互会产生很多观测，智能体的目的是从这些观测之中学到能最大化奖励的策略。

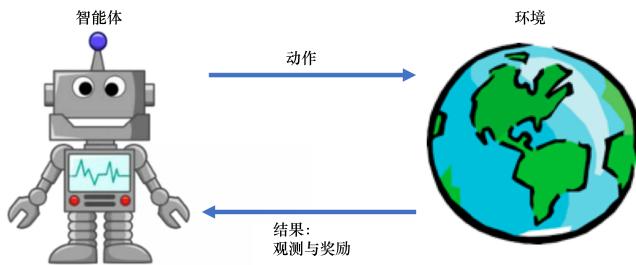


图 1.12 智能体和环境

1.2.2 奖励

奖励是由环境给的一种标量的反馈信号 (scalar feedback signal)，这种信号可显示智能体在某一步采取某个策略的表现如何。强化学习的目的就是最大化智能体可以获得的奖励，智能体在环境里面存在的目的就是最大化它的期望的累积奖励 (expected cumulative reward)。不同的环境中，奖励也是不同的。这里给大家举一些奖励的例子。

- (1) 比如一个象棋选手，他的目的是赢棋，在最后棋局结束的时候，他就会得到一个正奖励（赢）或者负奖励（输）。
- (2) 在股票管理里面，奖励由股票获取的奖励与损失决定。
- (3) 在玩雅达利游戏的时候，奖励就是增加或减少的游戏的分数，奖励本身的稀疏程度决定了游戏的难度。

1.2.3 序列决策

在一个强化学习环境里面，智能体的目的就是选取一系列的动作来最大化奖励，所以这些选取的动作必须有长期的影响。但在这个过程里面，智能体的奖励其实是被延迟了的，就是我们现在选取的某一步动作，可能要等到很久后才知道这一步到底产生了什么样的影响。如图 1.13 所示，在玩雅达利的 Pong 游戏时，我们可能只有到最后游戏结束时，才知道球到底有没有被击打过去。过程中我们采取的上升 (up) 或下降 (down) 动作，并不会直接产生奖励。强化学习里面一个重要的课题就是近期奖励和远期奖励的权衡 (trade-off)，研究怎么让智能体取得更多的远期奖励。

在与环境的交互过程中，智能体会获得很多观测。针对每一个观测，智能体会采取一个动作，也会得

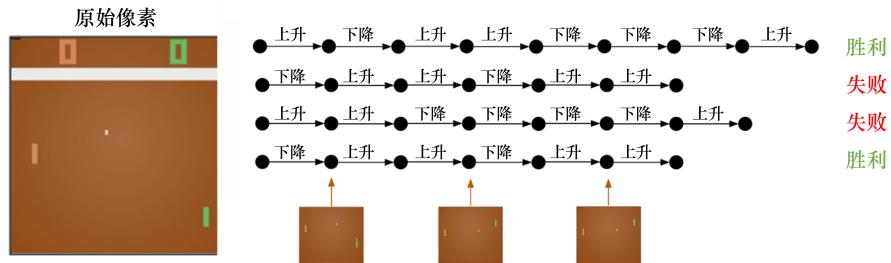


图 1.13 玩 Pong 游戏

到一个奖励。所以历史是观测、动作、奖励的序列：

$$H_t = o_1, a_1, r_1, \dots, o_t, a_t, r_t \quad (1.2)$$

智能体在采取当前动作的时候会依赖于它之前得到的历史，所以我们可以把整个游戏的状态看成关于这个历史的函数：

$$s_t = f(H_t) \quad (1.3)$$

Q：状态和观测有什么关系？

A：状态是对世界的完整描述，不会隐藏世界的信息。观测是对状态的部分描述，可能会遗漏一些信息。在深度强化学习中，我们几乎总是用实值的向量、矩阵或者更高阶的张量来表示状态和观测。例如，我们可以用 RGB 像素值的矩阵来表示一个视觉的观测，可以用机器人关节的角度和速度来表示一个机器人的状态。

环境有自己的函数 $s_t^e = f^e(H_t)$ 来更新状态，在智能体的内部也有一个函数 $s_t^a = f^a(H_t)$ 来更新状态。当智能体的状态与环境的状态等价的时候，即当智能体能够观察到环境的所有状态时，我们称这个环境是完全可观测的（fully observed）。在这种情况下，强化学习通常被建模成一个马尔可夫决策过程（Markov decision process, MDP）的问题。在马尔可夫决策过程中， $o_t = s_t^e = s_t^a$ 。

但是有一种情况是智能体得到的观测并不能包含环境运作的所有状态，因为在强化学习的设定里面，环境的状态才是真正的所有状态。比如智能体在玩 black jack 游戏，它能看到的其实是牌面上的牌。或者在玩雅达利游戏的时候，观测到的只是当前电视上面这一帧的信息，我们并没有得到游戏内部里面所有的运作状态。也就是当智能体只能看到部分的观测，我们就称这个环境是部分可观测的（partially observed）。在这种情况下，强化学习通常被建模成部分可观测马尔可夫决策过程（partially observable Markov decision process, POMDP）的问题。部分可观测马尔可夫决策过程是马尔可夫决策过程的一种泛化。部分可观测马尔可夫决策过程依然具有马尔可夫性质，但是假设智能体无法感知环境的状态，只能知道部分观测值。比如在自动驾驶中，智能体只能感知传感器采集的有限的环境信息。部分可观测马尔可夫决策过程可以用一个七元组描述： $(S, A, T, R, \Omega, O, \gamma)$ 。其中 S 表示状态空间，为隐变量， A 为动作空间， $T(s'|s, a)$ 为状态转移概率， R 为奖励函数， $\Omega(o|s, a)$ 为观测概率， O 为观测空间， γ 为折扣因子。

1.3 动作空间

不同的环境允许不同种类的动作。在给定的环境中，有效动作的集合经常被称为动作空间（action space）。像雅达利游戏和围棋（Go）这样的环境有离散动作空间（discrete action space），在这个动作空间里，智能体的动作数量是有限的。在其他环境，比如在物理世界中控制一个智能体，在这个环境中就有连续动作空间（continuous action space）。在连续动作空间中，动作是实值的向量。

例如，走迷宫机器人如果只有往东、往南、往西、往北这 4 种移动方式，则其动作空间为离散动作空间；如果机器人可以向 360° 中的任意角度进行移动，则其动作空间为连续动作空间。

1.4 强化学习智能体的组成成分和类型

对于一个强化学习智能体，它可能有一个或多个如下的组成成分。

- **策略 (policy)**。智能体会用策略来选取下一步的动作。
- **价值函数 (value function)**。我们用价值函数来对当前状态进行评估。价值函数用于评估智能体进入某个状态后，可以对后面的奖励带来多大的影响。价值函数值越大，说明智能体进入这个状态越有利。
- **模型 (model)**。模型表示智能体对环境的状态进行理解，它决定了环境中世界的运行方式。

下面我们深入了解这 3 个组成部分的细节。

1.4.1 策略

策略是智能体的动作模型，它决定了智能体的动作。它其实是一个函数，用于把输入的状态变成动作。策略可分为两种：随机性策略和确定性策略。

随机性策略 (stochastic policy) 就是 π 函数，即 $\pi(a|s) = p(a_t = a|s_t = s)$ 。输入一个状态 s ，输出一个概率。这个概率是智能体所有动作的概率，然后对这个概率分布进行采样，可得到智能体将采取的动作。比如可能是有 0.7 的概率往左，0.3 的概率往右，那么通过采样就可以得到智能体将采取的动作。

确定性策略 (deterministic policy) 就是智能体直接采取最有可能的动作，即 $a^* = \arg \max_a \pi(a | s)$ 。

如图 1.14 所示，从雅达利游戏来看，策略函数的输入就是游戏的一帧，它的输出决定智能体向左移动或者向右移动。

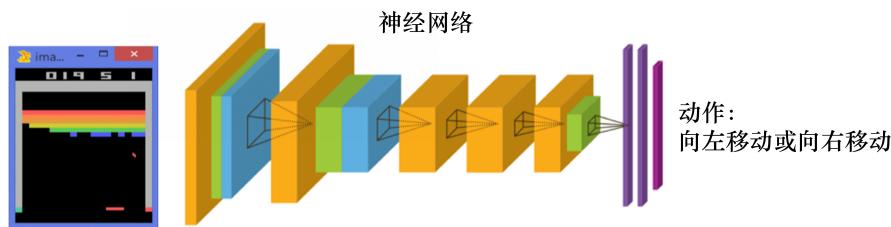


图 1.14 策略函数

通常情况下，强化学习一般使用随机性策略，随机性策略有很多优点。比如，在学习时可以通过引入一定的随机性来更好地探索环境；随机性策略的动作具有多样性，这一点在多个智能体博弈时非常重要。采用确定性策略的智能体总是对同样的状态采取相同动作，这会导致它的策略很容易被对手预测。

1.4.2 价值函数

价值函数的值是对未来奖励的预测，我们用它来评估状态的好坏。价值函数里面有一个**折扣因子 (discount factor)**，我们希望在尽可能短的时间里面得到尽可能多的奖励。比如现在给我们两个选择：10 天后给我们 100 块钱或者现在给我们 100 块钱。我们肯定更希望现在就给我们 100 块钱，因为我们可以把这 100 块钱存在银行里面，这样就会有一些利息。因此，我们可以把折扣因子放到价值函数的定义里面，价值函数的定义为

$$V_\pi(s) \doteq \mathbb{E}_\pi [G_t | s_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right], \text{ 对于所有的 } s \in S \quad (1.4)$$

期望 \mathbb{E}_π 的下标是 π 函数， π 函数的值可反映在我们使用策略 π 的时候，到底可以得到多少奖励。

我们还有一种价值函数：Q 函数。Q 函数里面包含两个变量：状态和动作。其定义为

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi [G_t | s_t = s, a_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right] \quad (1.5)$$

所以我们未来可以获得奖励的期望取决于当前的状态和当前的动作。Q 函数是强化学习算法里面要学习的一个函数。因为当我们得到 Q 函数后，进入某个状态要采取的最优动作可以通过 Q 函数得到。

1.4.3 模型

第 3 个组成部分是模型，模型决定了下一步的状态。下一步的状态取决于当前的状态以及当前采取的动作。它由状态转移概率和奖励函数两个部分组成。状态转移概率即

$$p_{ss'}^a = p(s_{t+1} = s' | s_t = s, a_t = a) \quad (1.6)$$

奖励函数是指我们在当前状态采取了某个动作，可以得到多大的奖励，即

$$R(s, a) = \mathbb{E}[r_{t+1} | s_t = s, a_t = a] \quad (1.7)$$

当我们有了策略、价值函数和模型 3 个组成部分后，就形成了一个马尔可夫决策过程（Markov decision process）。如图 1.15 所示，这个决策过程可视化了状态之间的转移以及采取的动作。

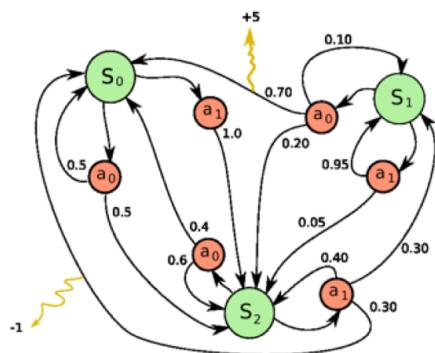


图 1.15 马尔可夫决策过程

我们来看一个走迷宫的例子。如图 1.16 所示，要求智能体从起点 (start) 开始，然后到达终点 (goal) 的位置。每走一步，我们就会得到 -1 的奖励。我们可以采取的动作是往上、下、左、右走。我们用现在智能体所在的位置来描述当前状态。

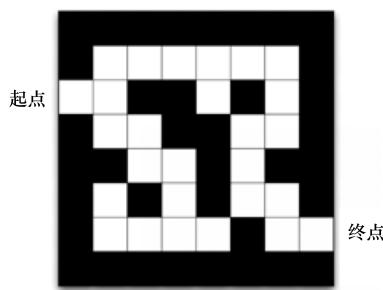


图 1.16 走迷宫的例子

我们可以用不同的强化学习方法来解这个环境。如果我们采取基于策略的强化学习 (policy-based RL) 方法，当学习好了这个环境后，在每一个状态，我们都会得到一个最佳的动作。如图 1.17 所示，比如我们现在在起点位置，我们知道最佳动作是往右走；在第二格的时候，得到的最佳动作是往上走；第三格是往右走……通过最佳的策略，我们可以最快地到达终点。

如果换成基于价值的强化学习 (value-based RL) 方法，利用价值函数作为导向，我们就会得到另外一种表征，每一个状态会返回一个价值。如图 1.18 所示，比如我们在起点位置的时候，价值是 -16 ，因为我们最快可以 16 步到达终点。因为每走一步会减 1，所以这里的价值是 -16 。当我们快接近终点的时候，

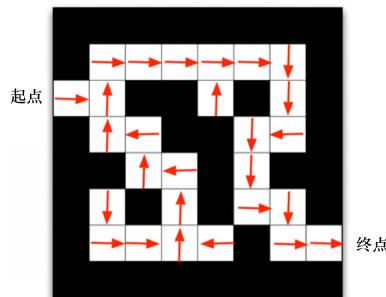


图 1.17 使用基于策略的强化学习方法得到的结果

这个数字变得越来越大。在拐角的时候，比如现在在第二格，价值是 -15 ，智能体会看上、下两格，它看到上面格子的价值变大了，变成 -14 了，下面格子的价值是 -16 ，那么智能体就会采取一个往上走的动作。所以通过学习的价值的不同，我们可以抽取出现在最佳的策略。

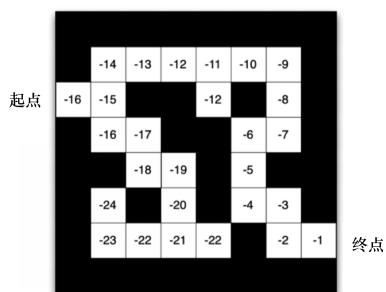


图 1.18 使用基于价值的强化学习方法得到的结果

1.4.4 强化学习智能体的类型

1. 基于价值的智能体与基于策略的智能体

根据智能体学习的事物不同，我们可以把智能体进行归类。基于价值的智能体（value-based agent）显式地学习价值函数，隐式地学习它的策略。策略是其从学到的价值函数里面推算出来的。基于策略的智能体（policy-based agent）直接学习策略，我们给它一个状态，它就会输出对应动作的概率。基于策略的智能体并没有学习价值函数。把基于价值的智能体和基于策略的智能体结合起来就有了演员-评论员智能体（actor-critic agent）。这一类智能体把策略和价值函数都学习了，然后通过两者的交互得到最佳的动作。

Q: 基于策略和基于价值的强化学习方法有什么区别？

A: 对于一个状态转移概率已知的马尔可夫决策过程，我们可以使用动态规划算法来求解。从决策方式来看，强化学习又可以划分为基于策略的方法和基于价值的方法。决策方式是智能体在给定状态下从动作集合中选择一个动作的依据，它是静态的，不随状态变化而变化。在基于策略的强化学习方法中，智能体会制定一套动作策略（确定在给定状态下需要采取何种动作），并根据这个策略进行操作。强化学习算法直接对策略进行优化，使制定的策略能够获得最大的奖励。而在基于价值的强化学习方法中，智能体不需要制定显式的策略，它维护一个价值表格或价值函数，并通过这个价值表格或价值函数来选取价值最大的动作。基于价值迭代的方法只能应用在不连续的、离散的环境下（如围棋或某些游戏领域），对于动作集合规模庞大、动作连续的场景（如机器人控制领域），其很难学习到较好的结果（此时基于策略迭代的方法能够根据设定的策略来选择连续的动作）。基于价值的强化学习算法有 Q 学习（Q-learning）、Sarsa 等，而基于策略的强化学习算法有策略梯度算法等。此外，演员-评论员算法同时使用策略和价值评估来做

出决策。其中，智能体会根据策略做出动作，而价值函数会对做出的动作给出价值，这样可以在原有的策略梯度算法的基础上加速学习过程，取得更好的效果。

2. 有模型强化学习智能体与免模型强化学习智能体

另外，我们可以通过智能体到底有没有学习环境模型来对智能体进行分类。**有模型 (model-based)** 强化学习智能体，它通过学习状态的转移来采取动作。**免模型 (model-free)** 强化学习智能体，它没有去直接估计状态的转移，也没有得到环境的具体转移变量，它通过学习价值函数和策略函数进行决策。免模型强化学习智能体的模型里面没有环境转移的模型。

我们可以用马尔可夫决策过程来定义强化学习任务，并将其表示为四元组 $\langle S, A, P, R \rangle$ ，即状态集合、动作集合、状态转移函数和奖励函数。如果这个四元组中所有元素均已知，且状态集合和动作集合在有限步数内是有限集，则智能体可以对真实环境进行建模，构建一个虚拟世界来模拟真实环境中的状态和交互反应。具体来说，当智能体知道状态转移函数 $P(s_{t+1}|s_t, a_t)$ 和奖励函数 $R(s_t, a_t)$ 后，它就能知道在某一状态下执行某一动作后能带来的奖励和环境的下一状态，这样智能体就不需要在真实环境中采取动作，直接在虚拟世界中学习和规划策略即可。这种学习方法称为**有模型强化学习**。有模型强化学习的流程如图 1.19 所示。

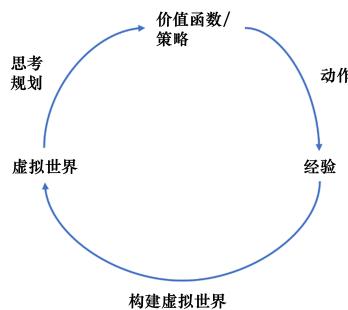


图 1.19 有模型强化学习流程

然而在实际应用中，智能体并不是那么容易就能知晓马尔可夫决策过程中的所有元素的。通常情况下，状态转移函数和奖励函数很难估计，甚至连环境中的状态都可能是未知的，这时就需要采用免模型强化学习。免模型强化学习没有对真实环境进行建模，智能体只能在真实环境中通过一定的策略来执行动作，等待奖励和状态迁移，然后根据这些反馈信息来更新动作策略，这样反复迭代直到学习到最优策略。

Q：有模型强化学习和免模型强化学习有什么区别？

A：针对是否需要对真实环境建模，强化学习可以分为有模型强化学习和免模型强化学习。有模型强化学习是指根据环境中的经验，构建一个虚拟世界，同时在真实环境和虚拟世界中学习；免模型强化学习是指不对环境进行建模，直接与真实环境进行交互来学习到最优策略。

总之，有模型强化学习相比免模型强化学习仅仅多出一个步骤，即对真实环境进行建模。因此，一些有模型的强化学习方法，也可以在免模型的强化学习方法中使用。在实际应用中，如果不清楚该用有模型强化学习还是免模型强化学习，可以先思考在智能体执行动作前，是否能对下一步的状态和奖励进行预测，如果能，就能够对环境进行建模，从而采用有模型学习。

免模型强化学习通常属于数据驱动型方法，需要大量的采样来估计状态、动作及奖励函数，从而优化动作策略。例如，在雅达利平台上的《太空侵略者》游戏中，免模型的深度强化学习需要大约两亿帧游戏画面才能学到比较理想的效果。相比之下，有模型的深度强化学习可以在一定程度上缓解训练数据匮乏的问题，因为智能体可以在虚拟世界中进行训练。免模型学习的泛化性要优于有模型强化学习，原因是免模型强化学习算法需要对真实环境进行建模，并且虚拟世界与真实环境之间可能还有差异，这限制了有模型强化学习算法的泛化性。有模型的强化学习方法可以对环境建模，使得该类方法具有独特魅力，即“想象能

力”。在免模型强化学习中，智能体只能一步一步地采取策略，等待真实环境的反馈；有模型强化学习可以在虚拟世界中预测出将要发生的事，并采取对自己最有利的策略。

目前，大部分深度强化学习方法都采用了免模型强化学习，这是因为：免模型强化学习更为简单、直观且有丰富的开源资料，如 AlphaGo 系列都采用免模型强化学习；在目前的强化学习研究中，大部分情况下环境都是静态的、可描述的，智能体的状态是离散的、可观察的（如雅达利游戏平台），这种相对简单、确定的问题并不需要评估状态转移函数和奖励函数，可直接采用免模型强化学习，使用大量的样本进行训练就能获得较好的效果^[1]。

如图 1.20 所示，我们可以把几类模型放到同一个图里面。图 1.20 有 3 个组成成分：价值函数、策略和模型。按一个智能体具有三者中的三者、两者或一者的情况可以把它分成很多类。

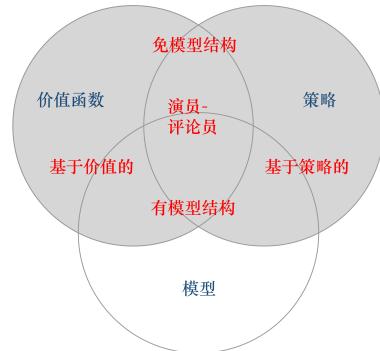


图 1.20 强化学习智能体的类型

1.5 学习与规划

学习 (learning) 和规划 (planning) 是序列决策的两个基本问题。如图 1.21 所示，在强化学习中，环境初始时是未知的，智能体不知道环境如何工作，它通过不断地与环境交互，逐渐改进策略。

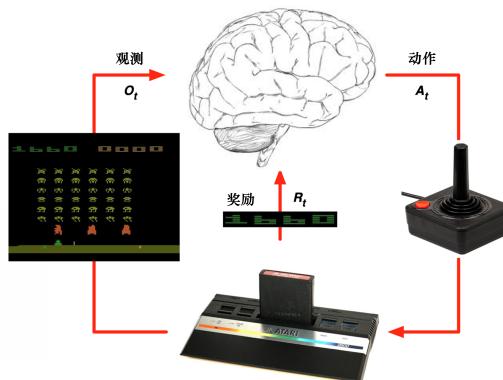


图 1.21 学习

如图 1.22 所示，在规划中，环境是已知的，智能体被告知了整个环境的运作规则的详细信息。智能体能够计算出一个完美的模型，并且在不需要与环境进行任何交互的时候进行计算。智能体不需要实时地与环境交互就能知道未来环境，只需要知道当前的状态，就能够开始思考，来寻找最优解。

在图 1.22 所示的游戏中，规则是确定的，我们知道选择左之后环境将会产生什么变化。我们完全可以通过已知的规则，来在内部模拟整个决策过程，无需与环境交互。一个常用的强化学习问题解决思路是，先学习环境如何工作，也就是了解环境工作的方式，即学习得到一个模型，然后利用这个模型进行规划。

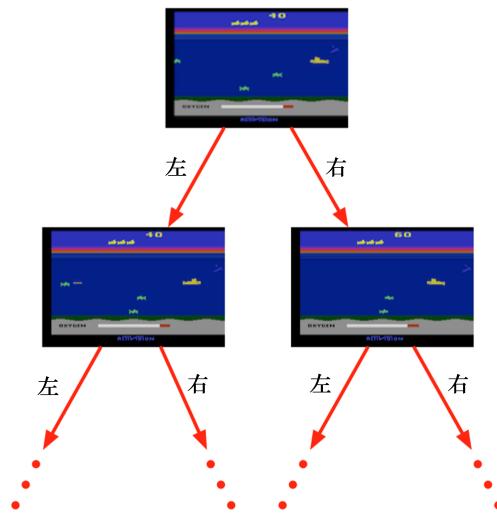


图 1.22 规划

1.6 探索和利用

在强化学习里面，探索和利用是两个很核心的问题。探索即我们去探索环境，通过尝试不同的动作来得到最佳的策略（带来最大奖励的策略）。利用即我们不去尝试新的动作，而是采取已知的可以带来很大奖励的动作。在刚开始的时候，强化学习智能体不知道它采取了某个动作后会发什么，所以它只能通过试错去探索，所以探索就是通过试错来理解采取的动作到底可不可以带来好的奖励。利用是指我们直接采取已知的可以带来很好奖励的动作。所以这里就面临一个权衡问题，即怎么通过牺牲一些短期的奖励来理解动作，从而学习到更好的策略。

下面举一些探索和利用的例子。以选择餐馆为例，利用是指我们直接去我们最喜欢的餐馆，因为我们去过这个餐馆很多次了，所以我们知道这里面的菜都非常可口。探索是指我们用手机搜索一个新的餐馆，然后去尝试它的菜到底好不好吃。我们有可能对这个新的餐馆感到非常不满意，这样钱就浪费了。以做广告为例，利用是指我们直接采取最优的广告策略。探索是指我们换一种广告策略，看看这个新的广告策略可不可以得到更好的效果。以挖油为例，利用是指我们直接在已知的地方挖油，这样可以确保挖到油。探索是指我们在一个新的地方挖油，这样就有很大的概率可能不能发现油田，但也可能有比较小的概率可以发现一个非常大的油田。以玩游戏为例，利用是指我们总是采取某一种策略。比如，我们玩《街头霸王》游戏的时候，采取的策略可能是蹲在角落，然后一直出脚。这个策略很可能可以奏效，但可能遇到特定的对手就会失效。探索是指我们可能尝试一些新的招式，有可能我们会放出“大招”来，这样就可能“一招毙命”。

与监督学习任务不同，强化学习任务的最终奖励在多步动作之后才能观察到，这里我们不妨先考虑比较简单的情形：最大化单步奖励，即仅考虑一步动作。需注意的是，即便在这样的简单情形下，强化学习仍与监督学习有显著不同，因为智能体需通过试错来发现各个动作产生的结果，而没有训练数据告诉智能体应当采取哪个动作。

想要最大化单步奖励需考虑两个方面：一是需知道每个动作带来的奖励，二是要执行奖励最大的动作。若每个动作对应的奖励是一个确定值，那么尝试遍所有的动作便能找出奖励最大的动作。然而，更一般的情形是，一个动作的奖励值是来自一个概率分布，仅通过一次尝试并不能确切地获得平均奖励值。

实际上，单步强化学习任务对应于一个理论模型，即 **K-臂赌博机 (K-armed bandit)**。K-臂赌博机也被称为**多臂赌博机 (multi-armed bandit)**。如图 1.23 所示，K-臂赌博机有 K 个摇臂，赌徒在投入一个硬币后可选择按下其中一个摇臂，每个摇臂以一定的概率吐出硬币，但这个概率赌徒并不知道。赌徒的目标是通过一定的策略最大化自己的奖励，即获得最多的硬币^[2]。若仅为获知每个摇臂的期望奖励，则可采用**仅探索 (exploration-only)** 法：将所有的尝试机会平均分配给每个摇臂（即轮流按下每个摇臂），

最后以每个摇臂各自的平均吐币概率作为其奖励期望的近似估计。若仅为执行奖励最大的动作，则可采用仅利用（exploitation-only）法：按下目前最优的（即到目前为止平均奖励最大的）摇臂，若多个摇臂同为最优，则从中随机选取一个。

显然，仅探索法能很好地估计每个摇臂的奖励，却会失去很多选择最优摇臂的机会；仅利用法则相反，它没有很好地估计摇臂期望奖励，很可能经常选不到最优摇臂。因此，这两种方法都难以使最终的累积奖励最大化。

事实上，探索（估计摇臂的优劣）和利用（选择当前最优摇臂）这两者是矛盾的，因为尝试次数（总投币数）有限，加强了一方则自然会削弱另一方，这就是强化学习所面临的探索-利用窘境（exploration-exploitation dilemma）。显然，想要累积奖励最大，则必须在探索与利用之间达成较好的折中。

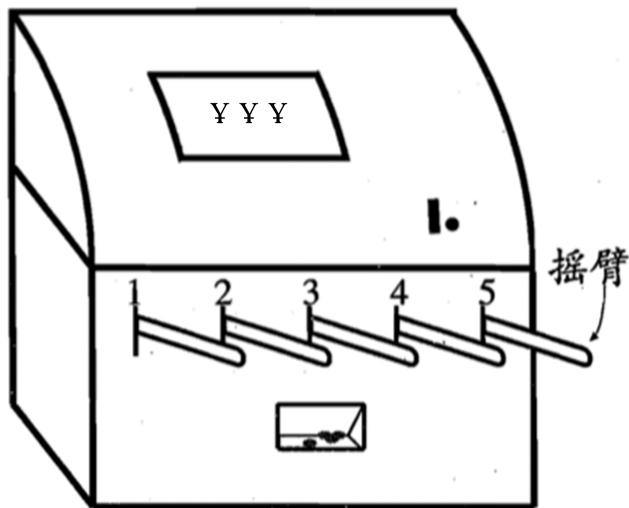


图 1.23 K -臂赌博机图示^[2]

1.7 强化学习实验

强化学习是一个理论与实践相结合的机器学习分支，我们不仅要理解它算法背后的一些数学原理，还要通过上机实践实现算法。在很多实验环境里面去探索算法能不能得到预期效果也是一个非常重要的过程。我们可以使用 Python 和深度学习的一些包来实现强化学习算法。现在有很多深度学习的包可以使用，比如 PyTorch、TensorFlow、Keras，熟练使用其中的两三种，就可以实现非常多的功能。所以我们并不需要从头去“造轮子”。

1.7.1 Gym

OpenAI 是一家非营利性的人工智能研究公司，其公布了非常多的学习资源以及算法资源。其之所以叫作 OpenAI，他们把所有开发的算法都进行了开源。如图 1.24 所示，OpenAI 的 **Gym** 库是一个环境仿真库，里面包含很多现有的环境。针对不同的场景，我们可以选择不同的环境。离散控制场景（输出的动作是可数的，比如 Pong 游戏中输出的向上或向下动作）一般使用雅达利环境评估；连续控制场景（输出的动作是不可数的，比如机器人走路时不仅有方向，还有角度，角度就是不可数的，是一个连续的量）一般使用 MuJoCo 环境评估。**Gym Retro** 是对 Gym 环境的进一步扩展，包含更多的游戏。

我们可以通过 pip 来安装 Gym 库：

```
pip install gym
```

在 Python 环境中导入 Gym 库，如果不报错，就可以认为 Gym 库安装成功。

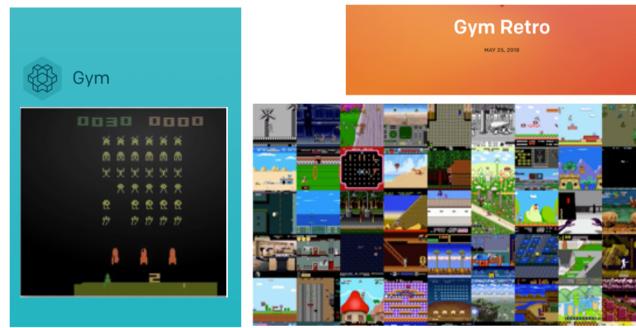


图 1.24 OpenAI 的 Gym 库

```
$python
>>>import gym
```

比如我们现在安装了 Gym 库，就可以直接调入 Taxi-v3 的环境。初始化这个环境后，我们就可以进行交互了。智能体得到某个观测后，它就会输出一个动作。这个动作会被环境拿去执行某个步骤，然后环境就会往前走一步，返回新的观测、奖励以及一个 flag 变量 done，done 决定这个游戏是不是结束了。我们通过几行代码就可实现强化学习的框架：

```
import gym
env = gym.make("Taxi-v3")
observation = env.reset()
agent = load_agent()
for step in range(100):
    action = agent(observation)
    observation, reward, done, info = env.step(action)
```

如图 1.25 所示，Gym 库里面有很多经典的控制类游戏。比如 *Acrobot* 需要让一个双连杆机器人立起来；*CartPole* 需要通过控制一辆小车，让杆立起来；*MountainCar* 需要通过前后移动车，让它到达旗帜的位置。在刚开始测试强化学习的时候，我们可以选择这些简单环境，因为强化学习在这些环境中可以在一两分钟之内见到效果。

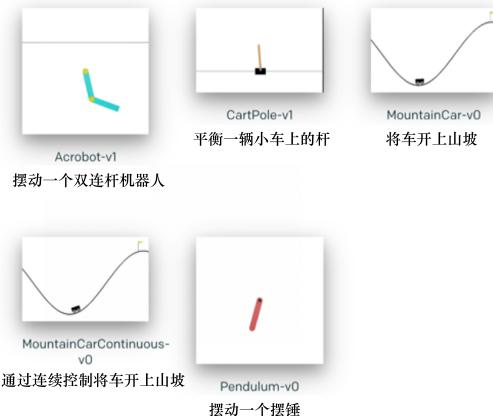


图 1.25 经典控制问题

如图 1.26 所示，*CartPole-v0* 环境有两个动作：将小车向左移动和将小车向右移动。我们还可以得到观测：小车当前的位置，小车当前往左、往右移的速度，杆的角度以及杆的最高点（顶端）的速度。观测越详细，我们就可以更好地描述当前所有的状态。这里有奖励的定义，如果能多走一步，我们就会得到一个奖励（奖励值为 1），所以我们需要存活尽可能多的时间来得到更多的奖励。当杆的角度大于某一个角

度（没能保持平衡），或者小车的中心到达图形界面窗口的边缘，或者累积步数大于 200，游戏就结束了，我们就输了。所以智能体的目的是控制杆，让它尽可能地保持平衡以及尽可能保持在环境的中央。

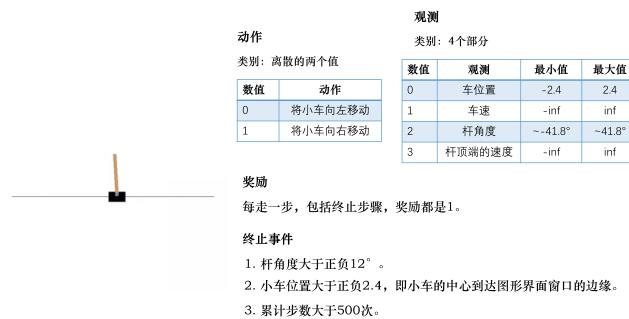


图 1.26 CartPole-v0 的例子

```
import gym # 导入 Gym 的 Python 接口环境包
env = gym.make('CartPole-v0') # 构建实验环境
env.reset() # 重置一个回合
for _ in range(1000):
    env.render() # 显示图形界面
    action = env.action_space.sample() # 从动作空间中随机选取一个动作
    env.step(action) # 用于提交动作，括号内是具体的动作
env.close() # 关闭环境
```

注意：如果绘制了实验的图形界面窗口，那么关闭该窗口的最佳方式是调用 `env.close()`。试图直接关闭图形界面窗口可能会导致内存不能释放，甚至会导致死机。

当我们执行这段代码时，机器人会驾驶着小车朝某个方向一直跑，直到我们看不见，这是因为我们还没开始训练机器人。

Gym 库中的大部分小游戏都可以用一个普通的实数或者向量来表示动作。输出 `env.action_space.sample()` 的返回值，能看到输出为 1 或者 0。`env.action_space.sample()` 的含义是，在该游戏的所有动作空间里随机选择一个作为输出。在这个例子中，动作只有两个：0 和 1，一左一右。`env.step()` 方法有 4 个返回值：`observation`、`reward`、`done`、`info`。

`observation` 是状态信息，是在游戏中观测到的屏幕像素值或者盘面状态描述信息。`reward` 是奖励值，即动作提交以后能够获得的奖励值。这个奖励值因游戏的不同而不同，但总体原则是，对完成游戏有帮助的动作会获得比较高的奖励值。`done` 表示游戏是否已经完成，如果完成了，就需要重置游戏并开始一个新的回合。`info` 是一些比较原始的用于诊断和调试的信息，或许对训练有帮助。不过，OpenAI 在评价我们提交的机器人时，是不允许使用这些信息的。

在每个训练中都要使用的返回值有 `observation`、`reward`、`done`。但 `observation` 的结构会由于游戏的不同而发生变化。以 CartPole-v0 为例，我们对代码进行修改：

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    print(observation)
env.close()
```

输出：

```
[ 0.01653398 0.19114579 0.02013859 -0.28050058]
[ 0.0203569 -0.00425755 0.01452858 0.01846535]
[ 0.02027175 -0.19958481 0.01489789 0.31569658]
.....
```

从输出可以看出这是一个四维的观测。在其他游戏中会有维度更多的情况出现。

`env.step()` 完成了一个完整的 $S \rightarrow A \rightarrow R \rightarrow S'$ 过程。我们只要不断观测这样的过程，并让智能体在其中用相应的算法完成训练，就能得到一个高质量的强化学习模型。

我们想要查看当前 Gym 库已经注册了哪些环境，可以使用以下代码：

```
from gym import envs
env_specs = envs.registry.all()
envs_ids = [env_spec.id for env_spec in env_specs]
print(envs_ids)
```

每个环境都定义了自己的观测空间和动作空间。环境 `env` 的观测空间用 `env.observation_space` 表示，动作空间用 `env.action_space` 表示。观测空间和动作空间既可以是离散空间（取值是有限个离散的值），也可以是连续空间（取值是连续的值）。在 Gym 库中，一般离散空间用 `gym.spaces.Discrete` 类表示，连续空间用 `gym.spaces.Box` 类表示。

例如，环境 `MountainCar-v0` 的观测空间是 `Box(2,)`，表示观测可以用 2 个 float 值表示；环境 `MountainCar-v0` 的动作空间是 `Discrete(3)`，表示动作取值自 0,1,2。对于离散空间，`Discrete` 类实例的成员 `n` 表示有几个可能的取值；对于连续空间，`Box` 类实例的成员 `low` 和 `high` 表示每个浮点数的取值范围。

1.7.2 MountainCar-v0 例子

接下来，我们通过一个例子来学习如何与 Gym 库进行交互。我们选取**小车上山 (MountainCar-v0)** 作为例子。

首先我们来看看这个任务的观测空间和动作空间：

```
import gym
env = gym.make('MountainCar-v0')
print('观测空间 = {}'.format(env.observation_space))
print('动作空间 = {}'.format(env.action_space))
print('观测范围 = {} ~ {}'.format(env.observation_space.low,
    env.observation_space.high))
print('动作数 = {}'.format(env.action_space.n))
```

输出：

```
观测空间 = Box(2,)
动作空间 = Discrete(3)
观测范围 = [-1.2 -0.07] ~ [0.6 0.07]
动作数 = 3
```

由输出可知，观测空间是形状为 (2,) 的浮点型 np.array，动作空间是取 0,1,2 的 int 型数值。

接下来考虑智能体。智能体往往是我们自己实现的。我们可以实现一个智能体类——`BespokeAgent` 类，代码如下：

```
class BespokeAgent:
    def __init__(self, env):
        pass
```

```

def decide(self, observation): # 决策
    position, velocity = observation
    lb = min(-0.09 * (position + 0.25) ** 2 + 0.03,
              0.3 * (position + 0.9) ** 4 - 0.008)
    ub = -0.07 * (position + 0.38) ** 2 + 0.07
    if lb < velocity < ub:
        action = 2
    else:
        action = 0
    return action # 返回动作

def learn(self, *args): # 学习
    pass

agent = BespokeAgent(env)

```

智能体的 `decide()` 方法实现了决策功能，而 `learn()` 方法实现了学习功能。`BespokeAgent` 类是一个比较简单的类，它只能根据给定的数学表达式进行决策，不能有效学习，所以它并不是一个真正意义上的强化学习智能体类。但是，它用于演示智能体和环境的交互已经足够了。

接下来我们试图让智能体与环境交互，代码如下。

```

def play_montecarlo(env, agent, render=False, train=False):
    episode_reward = 0. # 记录回合总奖励，初始化为0
    observation = env.reset() # 重置游戏环境，开始新回合
    while True: # 不断循环，直到回合结束
        if render: # 判断是否显示
            env.render() # 显示图形界面，图形界面可以用 env.close() 语句关闭
        action = agent.decide(observation)
        next_observation, reward, done, _ = env.step(action) # 执行动作
        episode_reward += reward # 收集回合奖励
        if train: # 判断是否训练智能体
            agent.learn(observation, action, reward, done) # 学习
        if done: # 回合结束，跳出循环
            break
        observation = next_observation
    return episode_reward # 返回回合总奖励

```

上面代码中的 `play_montecarlo()` 函数可以让智能体和环境交互一个回合，这个函数有 4 个参数。`env` 是环境类。`agent` 是智能体类。`render` 是 `bool` 型变量，指示在运行过程中是否要图形化显示，如果函数参数 `render` 为 `True`，那么在交互过程中会调用 `env.render()` 以显示图形界面，而这个界面可以通过调用 `env.close()` 关闭。`train` 是 `bool` 型的变量，指示在运行过程中是否训练智能体，在训练过程中应当设置为 `True`，以调用 `agent.learn()` 函数；在测试过程中应当设置为 `False`，使得智能体不变。这个函数有一个返回值 `episode_reward`，是 `float` 型的数值，表示智能体与环境交互一个回合的回合总奖励。

接下来，我们使用下面的代码让智能体和环境交互一个回合，并在交互过程中进行图形化显示，可用 `env.close()` 语句关闭图形界面。

```

env.seed(0) # 设置随机数种子，只是为了让结果可以精确复现，一般情况下可删去
episode_reward = play_montecarlo(env, agent, render=True)
print('回合奖励 = {}'.format(episode_reward))
env.close() # 此语句可关闭图形界面

```

输出：

```
回合奖励 = -105.0
```

为了系统评估智能体的性能，下列代码求出了连续交互 100 回合的平均回合奖励。

```
episode_rewards = [play_montecarlo(env, agent) for _ in range(100)]
print('平均回合奖励 = {}'.format(np.mean(episode_rewards)))
```

输出：

```
平均回合奖励 = -102.61
```

小车上山环境有一个参考的回合奖励值 -110 ，如果连续 100 个回合的平均回合奖励大于 -110 ，则认为这个任务被解决了。BespokeAgent 类对应的策略的平均回合奖励就在 -110 左右。

测试智能体在 Gym 库中某个任务的性能时，学术界一般最关心 100 个回合的平均回合奖励。至于为什么是 100 个回合而不是其他回合数（比如 128 个回合），完全是习惯使然，没有什么特别的原因^[3]。对于有些任务，还会指定一个参考的回合奖励值，当连续 100 个回合的奖励大于指定的值时，就认为这个任务被解决了。但是，并不是所有的任务都指定了这样的值。对于没有指定值的任务，就无所谓任务被解决了或者没有被解决。

我们对 Gym 库的用法进行总结：使用 `env=gym.make(环境名)` 取出环境，使用 `env.reset()` 初始化环境，使用 `env.step(动作)` 执行一步环境，使用 `env.render()` 显示环境，使用 `env.close()` 关闭环境。Gym 库有对应的官方文档 (<https://gym.openai.com/docs/>)，读者可以阅读文档来学习 Gym 库。

1.8 关键词

强化学习 (reinforcement learning, RL)：智能体可以在与复杂且不确定的环境进行交互时，尝试使所获得的奖励最大化的算法。

动作 (action)：环境接收到的智能体基于当前状态的输出。

状态 (state)：智能体从环境中获取的状态。

奖励 (reward)：智能体从环境中获取的反馈信号，这个信号指定了智能体在某一步采取了某个策略以后是否得到奖励，以及奖励的大小。

探索 (exploration)：在当前的情况下，继续尝试新的动作。其有可能得到更高的奖励，也有可能一无所有。

开发 (exploitation)：在当前的情况下，继续尝试已知的可以获得最大奖励的过程，即选择重复执行当前动作。

深度强化学习 (deep reinforcement learning)：不需要手动设计特征，仅需要输入状态就可以让系统直接输出动作的一个端到端 (end-to-end) 的强化学习方法。通常使用神经网络来拟合价值函数 (value function) 或者策略网络 (policy network)。

全部可观测 (full observability)、完全可观测 (fully observed) 和部分可观测 (partially observed)：当智能体的状态与环境的状态等价时，我们就称这个环境是全部可观测的；当智能体能够观察到环境的所有状态时，我们称这个环境是完全可观测的；一般智能体不能观察到环境的所有状态时，我们称这个环境是部分可观测的。

部分可观测马尔可夫决策过程 (partially observable Markov decision process, POMDP)：即马尔可夫决策过程的泛化。部分可观测马尔可夫决策过程依然具有马尔可夫性质，但是其假设智能体无法感知环境的状态，只能知道部分观测值。

动作空间 (action space)、离散动作空间 (discrete action space) 和连续动作空间 (continuous action space)：在给定的环境中，有效动作的集合被称为动作空间，智能体的动作数量有限的动作空间称为离散动作空间，反之，则被称为连续动作空间。

基于策略的 (policy-based)：智能体会制定一套动作策略，即确定在给定状态下需要采取何种动作，并根据这个策略进行操作。强化学习算法直接对策略进行优化，使制定的策略能够获得最大的奖励。

基于价值的 (valued-based)：智能体不需要制定显式的策略，它维护一个价值表格或者价值函数，并通过这个价值表格或价值函数来执行使得价值最大化的动作。

有模型 (model-based) 结构：智能体通过学习状态的转移来进行决策。

免模型 (model-free) 结构：智能体没有直接估计状态的转移，也没有得到环境的具体转移变量，它通过学习价值函数或者策略网络进行决策。

1.9 习题

1-1 强化学习的基本结构是什么？

1-2 强化学习相对于监督学习为什么训练过程会更加困难？

1-3 强化学习的基本特征有哪些？

1-4 近几年强化学习发展迅速的原因有哪些？

1-5 状态和观测有什么关系？

1-6 一个强化学习智能体由什么组成？

1-7 根据强化学习智能体的不同，我们可以将其分为哪几类？

1-8 基于策略迭代和基于价值迭代的强化学习方法有什么区别？

1-9 有模型学习和免模型学习有什么区别？

1-10 如何通俗理解强化学习？

1.10 面试题

1-1 友善的面试官：看来你对于强化学习还是有一定了解的呀，那么可以用一句话谈一下你对于强化学习的认识吗？

1-2 友善的面试官：请问，你认为强化学习、监督学习和无监督学习三者有什么区别呢？

1-3 友善的面试官：根据你的理解，你认为强化学习的使用场景有哪些呢？

1-4 友善的面试官：请问强化学习中所谓的损失函数与深度学习中的损失函数有什么区别呢？

1-5 友善的面试官：你了解有模型和免模型吗？两者具体有什么区别呢？

参考文献

[1] 诸葛越, 江云胜, 葫芦娃. 百面深度学习[M]. 北京：人民邮电出版社, 2020.

[2] 周志华. 机器学习[M]. 北京：清华大学出版社, 2016.

[3] 肖智清. 强化学习：原理与 Python 实现[M]. 北京：机械工业出版社, 2019.

第 2 章 马尔可夫决策过程

图 2.1 介绍了强化学习里面智能体与环境之间的交互，智能体得到环境的状态后，它会采取动作，并把这个采取的动作返还给环境。环境得到智能体的动作后，它会进入下一个状态，把下一个状态传给智能体。在强化学习中，智能体与环境就是这样进行交互的，这个交互过程可以通过马尔可夫决策过程来表示，所以马尔可夫决策过程是强化学习的基本框架。

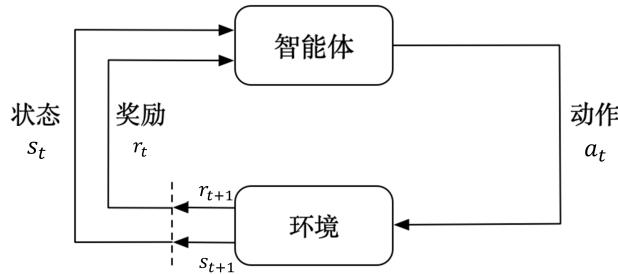


图 2.1 智能体与环境之间的交互

本章将介绍马尔可夫决策过程。在介绍马尔可夫决策过程之前，我们先介绍它的简化版本：马尔可夫过程（Markov process, MP）以及马尔可夫奖励过程（Markov reward process, MRP）。通过与这两种过程的比较，我们可以更容易理解马尔可夫决策过程。其次，我们会介绍马尔可夫决策过程中的策略评估（policy evaluation），就是当给定决策后，我们怎么去计算它的价值函数。最后，我们会介绍马尔可夫决策过程的控制，具体有策略迭代（policy iteration）和价值迭代（value iteration）两种算法。在马尔可夫决策过程中，它的环境是全部可观测的。但是很多时候环境里面有些量是不可观测的，但是这个部分可观测的问题也可以转换成马尔可夫决策过程的问题。

2.1 马尔可夫过程

2.1.1 马尔可夫性质

在随机过程中，**马尔可夫性质**（Markov property）是指一个随机过程在给定现在状态及所有过去状态情况下，其未来状态的条件概率分布仅依赖于当前状态。以离散随机过程为例，假设随机变量 X_0, X_1, \dots, X_T 构成一个随机过程。这些随机变量的所有可能取值的集合被称为状态空间（state space）。如果 X_{t+1} 对于过去状态的条件概率分布仅是 X_t 的一个函数，则

$$p(X_{t+1} = x_{t+1} | X_{0:t} = x_{0:t}) = p(X_{t+1} = x_{t+1} | X_t = x_t) \quad (2.1)$$

其中， $X_{0:t}$ 表示变量集合 X_0, X_1, \dots, X_t ， $x_{0:t}$ 为在状态空间中的状态序列 x_0, x_1, \dots, x_t 。

马尔可夫性质也可以描述为给定当前状态时，将来状态与过去状态是条件独立的^[1]。如果某一个过程满足**马尔可夫性质**，那么未来的转移与过去的是独立的，它只取决于现在。马尔可夫性质是所有马尔可夫过程的基础。

2.1.2 马尔可夫过程/马尔可夫链

马尔可夫过程是一组具有马尔可夫性质的随机变量序列 s_1, \dots, s_t ，其中下一个时刻的状态 s_{t+1} 只取决于当前状态 s_t 。我们设状态的历史为 $h_t = \{s_1, s_2, s_3, \dots, s_t\}$ (h_t 包含了之前的所有状态)，则马尔可夫过程满足条件：

$$p(s_{t+1} | s_t) = p(s_{t+1} | h_t) \quad (2.2)$$

从当前 s_t 转移到 s_{t+1} ，它是直接就等于它之前所有的状态转移到 s_{t+1} 。

离散时间的马尔可夫过程也称为**马尔可夫链 (Markov chain)**。例如，图 2.2 里面有 4 个状态，这 4 个状态在 s_1, s_2, s_3, s_4 之间互相转移。比如从 s_1 开始， s_1 有 0.1 的概率继续停留在 s_1 状态，有 0.2 的概率转移到 s_2 ，有 0.7 的概率转移到 s_4 。如果 s_4 是我们的当前状态，它有 0.3 的概率转移到 s_2 ，有 0.2 的概率转移到 s_3 ，有 0.5 的概率留在当前状态。

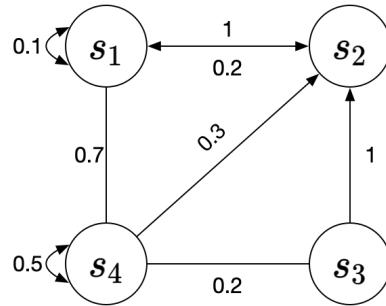


图 2.2 马尔可夫链示例

我们可以用**状态转移矩阵 (state transition matrix) P** 来描述状态转移 $p(s_{t+1} = s' | s_t = s)$:

$$P = \begin{pmatrix} p(s_1 | s_1) & p(s_2 | s_1) & \dots & p(s_N | s_1) \\ p(s_1 | s_2) & p(s_2 | s_2) & \dots & p(s_N | s_2) \\ \vdots & \vdots & \ddots & \vdots \\ p(s_1 | s_N) & p(s_2 | s_N) & \dots & p(s_N | s_N) \end{pmatrix} \quad (2.3)$$

状态转移矩阵类似于条件概率 (conditional probability)，它表示当我们知道当前我们在状态 s_t 时，到达下面所有状态的概率。所以它的每一行描述的是从一个节点到达所有其他节点的概率。

2.1.3 马尔可夫过程的例子

图 2.3 所示为一个马尔可夫过程的例子，这里有七个状态。比如从 s_1 开始，它有 0.4 的概率到 s_2 ，有 0.6 的概率留在当前的状态。 s_2 有 0.4 的概率到 s_1 ，有 0.4 的概率到 s_3 ，另外有 0.2 的概率留在当前状态。所以给定状态转移的马尔可夫链后，我们可以对这个链进行采样，这样就会得到一串轨迹。例如，假设我们从状态 s_3 开始，可以得到 3 个轨迹：

- s_3, s_4, s_5, s_6, s_6 ；
- s_3, s_2, s_3, s_2, s_1 ；
- s_3, s_4, s_4, s_5, s_5 。

通过对状态的采样，我们可以生成很多这样的轨迹。

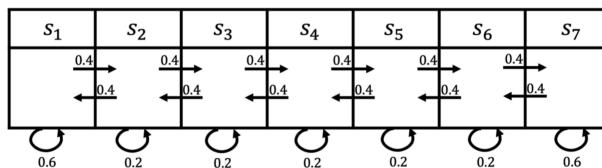


图 2.3 马尔可夫过程的例子

2.2 马尔可夫奖励过程

马尔可夫奖励过程 (Markov reward process, MRP) 是马尔可夫链加上奖励函数。在马尔可夫奖励过程中，状态转移矩阵和状态都与马尔可夫链一样，只是多了**奖励函数 (reward function)**。奖励函

数 R 是一个期望，表示当我们到达某一个状态的时候，可以获得多大的奖励。这里另外定义了折扣因子 γ 。如果状态数是有限的，那么 R 可以是一个向量。

2.2.1 回报与价值函数

这里我们进一步定义一些概念。范围（horizon）是指一个回合的长度（每个回合最大的时间步数），它是由有限个步数决定的。回报（return）是指把奖励进行折扣后所获得的奖励。回报可以定义为奖励的逐步叠加，即

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots + \gamma^{T-t-1} r_T \quad (2.4)$$

这里有一个折扣因子，越往后得到的奖励，折扣越多。这说明我们更希望得到现有的奖励，对未来的奖励要打折扣。当我们有了回报之后，就可以定义状态的价值了，就是状态价值函数（state-value function）。对于马尔可夫奖励过程，状态价值函数被定义成回报的期望，即

$$\begin{aligned} V^t(s) &= \mathbb{E}[G_t | s_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots + \gamma^{T-t-1} r_T | s_t = s] \end{aligned} \quad (2.5)$$

其中， G_t 是之前定义的折扣回报（discounted return）。我们对 G_t 取了一个期望，期望就是从这个状态开始，我们可能获得多大的价值。所以期望也可以看成未来可能获得奖励的当前价值的表现，就是当我们进入某一个状态后，我们现在有多大的价值。

我们使用折扣因子的原因如下。第一，有些马尔可夫过程是带环的，它并不会终结，我们想避免无穷的奖励。第二，我们并不能建立完美的模拟环境的模型，我们对未来的评估不一定是准确的，我们不一定完全信任模型，因为这种不确定性，所以我们对未来的评估增加一个折扣。我们想把这个不确定性表示出来，希望尽可能快地得到奖励，而不是在未来某一个点得到奖励。第三，如果奖励是有实际价值的，我们可能更希望立刻就得到奖励，而不是后面再得到奖励（现在的钱比以后的钱更有价值）。最后，我们也更想得到即时奖励。有些时候可以把折扣因子设为 0 ($\gamma = 0$)，我们就只关注当前的奖励。我们也可以把折扣因子设为 1 ($\gamma = 1$)，对未来的奖励并没有打折扣，未来获得的奖励与当前获得的奖励是一样的。折扣因子可以作为强化学习智能体的一个超参数（hyperparameter）来进行调整，通过调整折扣因子，我们可以得到不同动作的智能体。

在马尔可夫奖励过程里面，我们如何计算价值呢？如图 2.4 所示，马尔可夫奖励过程依旧是状态转移，其奖励函数可以定义为：智能体进入第一个状态 s_1 的时候会得到 5 的奖励，进入第七个状态 s_7 的时候会得到 10 的奖励，进入其他状态都没有奖励。我们可以用向量来表示奖励函数，即

$$\mathbf{R} = [5, 0, 0, 0, 0, 0, 10] \quad (2.6)$$

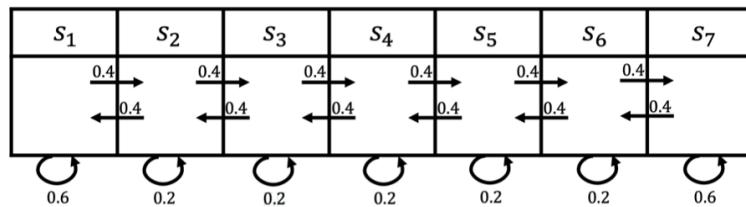


图 2.4 马尔可夫奖励过程的例子

我们对 4 步的回合 ($\gamma = 0.5$) 来采样回报 G 。

- (1) s_4, s_5, s_6, s_7 的回报： $0 + 0.5 \times 0 + 0.25 \times 0 + 0.125 \times 10 = 1.25$
- (2) s_4, s_3, s_2, s_1 的回报： $0 + 0.5 \times 0 + 0.25 \times 0 + 0.125 \times 5 = 0.625$

(3) s_4, s_5, s_6, s_7 的回报： $0 + 0.5 \times 0 + 0.25 \times 0 + 0.125 \times 0 = 0$

我们现在可以计算每一个轨迹得到的奖励，比如我们对轨迹 s_4, s_5, s_6, s_7 的奖励进行计算，这里折扣因子是 0.5。在 s_4 的时候，奖励为 0。下一个状态 s_5 的时候，因为我们已经到了下一步，所以要把 s_5 进行折扣， s_5 的奖励也是 0。然后是 s_6 ，奖励也是 0，折扣因子应该是 0.25。到达 s_7 后，我们获得了一个奖励，但是因为状态 s_7 的奖励是未来才获得的奖励，所以我们要对之进行 3 次折扣。最终这个轨迹的回报就是 1.25。类似地，我们可以得到其他轨迹的回报。

这里就引出了一个问题，当我们有了一些轨迹的实际回报时，怎么计算它的价值函数呢？比如我们想知道 s_4 的价值，即当我们进入 s_4 后，它的价值到底如何？一个可行的做法就是我们可以生成很多轨迹，然后把轨迹都叠加起来。比如我们可以从 s_4 开始，采样生成很多轨迹，把这些轨迹的回报都计算出来，然后将其取平均值作为我们进入 s_4 的价值。这其实是一种计算价值函数的办法，也就是通过蒙特卡洛（Monte Carlo, MC）采样的方法计算 s_4 的价值。

2.2.2 贝尔曼方程

但是这里我们采取了另外一种计算方法，从价值函数里面推导出**贝尔曼方程** (Bellman equation)：

$$V(s) = \underbrace{R(s)}_{\text{即时奖励}} + \gamma \underbrace{\sum_{s' \in S} p(s' | s) V(s')}_{\text{未来奖励的折扣总和}} \quad (2.7)$$

其中， s' 可以看成未来的所有状态， $p(s'|s)$ 是指从当前状态转移到未来状态的概率。 $V(s')$ 代表的是未来某一个状态的价值。我们从当前状态开始，有一定的概率去到未来的所有状态，所以我们要把 $p(s' | s)$ 写上去。我们得到了未来状态后，乘一个 γ ，这样就可以把未来的奖励打折扣。 $\gamma \sum_{s' \in S} p(s' | s) V(s')$ 可以看成未来奖励的折扣总和 (discounted sum of future reward)。

贝尔曼方程定义了当前状态与未来状态之间的关系。未来奖励的折扣总和加上即时奖励，就组成了贝尔曼方程。

1. 全期望公式

在推导贝尔曼方程之前，我们先仿照全期望公式 (law of total expectation) 的证明过程来证明：

$$\mathbb{E}[V(s_{t+1})|s_t] = \mathbb{E}[\mathbb{E}[G_{t+1}|s_{t+1}]|s_t] = \mathbb{E}[G_{t+1}|s_t] \quad (2.8)$$

全期望公式也被称为叠期望公式 (law of iterated expectations, LIE)。如果 A_i 是样本空间的有限或可数的划分 (partition)，则全期望公式可定义为

$$\mathbb{E}[X] = \sum_i \mathbb{E}[X | A_i] p(A_i)$$

证明：为了符号简洁并且易读，我们去掉下标，令 $s = s_t$, $g' = G_{t+1}$, $s' = s_{t+1}$ 。我们可以根据条件期望的定义来重写回报的期望为

$$\begin{aligned} \mathbb{E}[G_{t+1} | s_{t+1}] &= \mathbb{E}[g' | s'] \\ &= \sum_{g'} g' p(g' | s') \end{aligned} \quad (2.9)$$

如果 X 和 Y 都是离散型随机变量，则条件期望 (conditional expectation) $\mathbb{E}[X|Y=y]$ 定义为

$$\mathbb{E}[X | Y=y] = \sum_x x p(X=x | Y=y)$$

令 $s_t = s$, 我们对式 (2.9) 求期望可得

$$\begin{aligned}
 \mathbb{E}[\mathbb{E}[G_{t+1} | s_{t+1}] | s_t] &= \mathbb{E}[\mathbb{E}[g' | s'] | s] \\
 &= \mathbb{E}\left[\sum_{g'} g' p(g' | s') | s\right] \\
 &= \sum_{s'} \sum_{g'} g' p(g' | s', s) p(s' | s) \\
 &= \sum_{s'} \sum_{g'} \frac{g' p(g' | s', s) p(s' | s) p(s)}{p(s)} \\
 &= \sum_{s'} \sum_{g'} \frac{g' p(g' | s', s) p(s', s)}{p(s)} \\
 &= \sum_{s'} \sum_{g'} \frac{g' p(g', s' | s)}{p(s)} \\
 &= \sum_{g'} \sum_{s'} g' p(g', s' | s) \\
 &= \sum_{g'} g' p(g' | s) \\
 &= \mathbb{E}[g' | s] = \mathbb{E}[G_{t+1} | s_t]
 \end{aligned} \tag{2.10}$$

2. 贝尔曼方程推导

贝尔曼方程的推导过程如下:

$$\begin{aligned}
 V(s) &= \mathbb{E}[G_t | s_t = s] \\
 &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots | s_t = s] \\
 &= \mathbb{E}[r_{t+1} | s_t = s] + \gamma \mathbb{E}[r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \dots | s_t = s] \\
 &= R(s) + \gamma \mathbb{E}[G_{t+1} | s_t = s] \\
 &= R(s) + \gamma \mathbb{E}[V(s_{t+1}) | s_t = s] \\
 &= R(s) + \gamma \sum_{s' \in S} p(s' | s) V(s')
 \end{aligned} \tag{2.11}$$

贝尔曼方程就是当前状态与未来状态的迭代关系, 表示当前状态的价值函数可以通过下个状态的价值函数来计算。贝尔曼方程因其提出者、动态规划创始人理查德·贝尔曼 (Richard Bellman) 而得名, 也叫作“动态规划方程”。

贝尔曼方程定义了状态之间的迭代关系, 即

$$V(s) = R(s) + \gamma \sum_{s' \in S} p(s' | s) V(s') \tag{2.12}$$

假设有一个马尔可夫链如图 2.5a 所示, 贝尔曼方程描述的就是当前状态到未来状态的一个转移。如图 2.5b 所示, 假设我们当前在 s_1 , 那么它只可能去到 3 个未来的状态: 有 0.1 的概率留在它当前位置, 有 0.2 的概率去到 s_2 状态, 有 0.7 的概率去到 s_4 状态。所以我们把状态转移概率乘它未来的状态的价值, 再加上它的即时奖励 (immediate reward), 就会得到它当前状态的价值。贝尔曼方程定义的就是当前状态与未来状态的迭代关系。

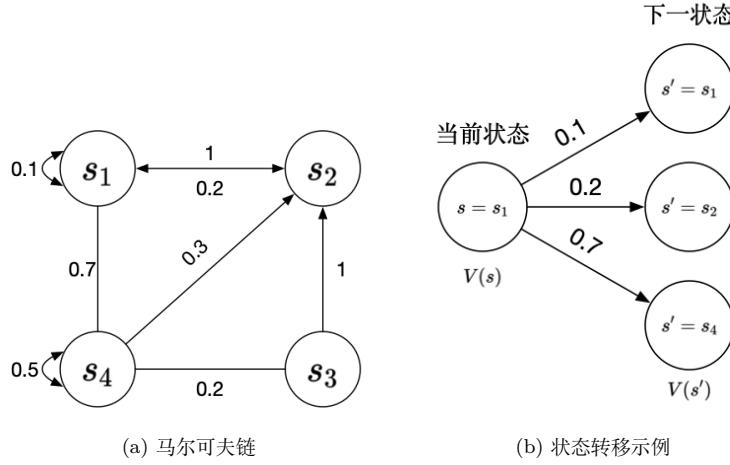


图 2.5 状态转移

我们可以把贝尔曼方程写成矩阵的形式：

$$\begin{pmatrix} V(s_1) \\ V(s_2) \\ \vdots \\ V(s_N) \end{pmatrix} = \begin{pmatrix} R(s_1) \\ R(s_2) \\ \vdots \\ R(s_N) \end{pmatrix} + \gamma \begin{pmatrix} p(s_1 | s_1) & p(s_2 | s_1) & \dots & p(s_N | s_1) \\ p(s_1 | s_2) & p(s_2 | s_2) & \dots & p(s_N | s_2) \\ \vdots & \vdots & \ddots & \vdots \\ p(s_1 | s_N) & p(s_2 | s_N) & \dots & p(s_N | s_N) \end{pmatrix} \begin{pmatrix} V(s_1) \\ V(s_2) \\ \vdots \\ V(s_N) \end{pmatrix} \quad (2.13)$$

我们当前的状态是向量 $[V(s_1), V(s_2), \dots, V(s_N)]^T$ 。每一行来看，向量 \mathbf{V} 乘状态转移矩阵里面的某一行，再加上它当前可以得到的奖励，就会得到它当前的价值。

当我们把贝尔曼方程写成矩阵形式后，可以直接求解：

$$\begin{aligned} \mathbf{V} &= \mathbf{R} + \gamma \mathbf{P} \mathbf{V} \\ \mathbf{I} \mathbf{V} &= \mathbf{R} + \gamma \mathbf{P} \mathbf{V} \\ (\mathbf{I} - \gamma \mathbf{P}) \mathbf{V} &= \mathbf{R} \\ \mathbf{V} &= (\mathbf{I} - \gamma \mathbf{P})^{-1} \mathbf{R} \end{aligned} \quad (2.14)$$

我们可以直接得到解析解 (analytic solution)：

$$\mathbf{V} = (\mathbf{I} - \gamma \mathbf{P})^{-1} \mathbf{R} \quad (2.15)$$

我们可以通过矩阵求逆把 \mathbf{V} 的价值直接求出来。但是一个问题是这个矩阵求逆的过程的复杂度是 $O(N^3)$ 。所以当状态非常多的时候，比如从 10 个状态到 1000 个状态，或者到 100 万个状态，当我们有 100 万个状态的时候，状态转移矩阵就会是一个 100 乘 100 的矩阵，对这样一个大矩阵求逆是非常困难的。所以这种通过解析解去求解的方法只适用于很小量的马尔可夫奖励过程。

2.2.3 计算马尔可夫奖励过程价值的迭代算法

我们可以将迭代的方法应用于状态非常多的马尔可夫奖励过程 (large MRP)，比如：动态规划的方法，蒙特卡洛的方法（通过采样的办法计算它），时序差分学习 (temporal-difference learning, TD learning) 的方法（时序差分学习是动态规划和蒙特卡洛方法的一个结合）。

首先我们用蒙特卡洛方法来计算价值。如图 2.6 所示，蒙特卡洛方法就是当得到一个马尔可夫奖励过程后，我们可以从某个状态开始，把小船放到状态转移矩阵里面，让它“随波逐流”，这样就会产生一个轨迹。产生一个轨迹之后，就会得到一个奖励，那么直接把折扣的奖励即回报 g 算出来。算出来之后将它积

```

1:  $i \leftarrow 0, G_t \leftarrow 0$ 
2: 当  $i \neq N$  时, 执行:
3:   生成一个回合的轨迹, 从状态  $s$  和时刻  $t$  开始
4:   使用生成的轨迹计算回报  $g = \sum_{i=t}^{H-1} \gamma^{i-t} r_i$ 
5:    $G_t \leftarrow G_t + g, i \leftarrow i + 1$ 
6: 结束循环
7:  $V_t(s) \leftarrow G_t/N$ 

```

图 2.6 计算马尔可夫奖励过程价值的蒙特卡洛方法

累起来, 得到回报 G_t 。当积累了一定数量的轨迹之后, 我们直接用 G_t 除以轨迹数量, 就会得到某个状态的价值。

比如我们要计算 s_4 状态的价值, 可以从 s_4 状态开始, 随机产生很多轨迹。把小船放到状态转移矩阵里面, 然后它就会“随波逐流”, 产生轨迹。每个轨迹都会得到一个回报, 我们得到大量的回报, 比如 100 个、1000 个回报, 然后直接取平均值, 就可以等价于现在 s_4 的价值, 因为 s_4 的价值 $V(s_4)$ 定义了我们未来可能得到多少的奖励。这就是蒙特卡洛采样的方法。

如图 2.7 所示, 我们也可以用动态规划的方法, 一直迭代贝尔曼方程, 直到价值函数收敛, 我们就可以得到某个状态的价值。我们通过**自举 (bootstrapping)** 的方法不停地迭代贝尔曼方程, 当最后更新的状态与我们上一个状态的区别并不大的时候, 更新就可以停止, 我们就可以输出最新的 $V'(s)$ 作为它当前的状态的价值。这里就是把贝尔曼方程变成一个贝尔曼更新 (Bellman update), 这样就可以得到状态的价值。

动态规划的方法基于后继状态价值的估计来更新现在状态价值的估计 (如图 2.7 所示算法中的第 3 行用 V' 来更新 V)。根据其他估算值来更新估算值的思想, 我们称其为自举。

```

1: 对于所有状态  $s \in S, V'(s) \leftarrow 0, V(s) \leftarrow \infty$ 
2: 当  $\|V - V'\| > \epsilon$  执行
3:    $V \leftarrow V'$ 
4:   对于所有状态  $s \in S, V'(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s)V(s')$ 
5: 结束循环
6: 返回  $V'(s)$  对于所有状态  $s \in S$ 

```

图 2.7 计算马尔可夫奖励过程价值的动态规划算法

bootstrap 的本意是“解靴带”。这里使用了德国文学作品《吹牛大王历险记》中解靴带自助（拔靴自助）的典故, 因此将其译为“自举”。^[2]

2.2.4 马尔可夫奖励过程的例子

如图 2.8 所示, 如果我们在马尔可夫链上加上奖励, 那么到达每个状态, 我们都会获得一个奖励。我们可以设置对应的奖励, 比如智能体到达状态 s_1 时, 可以获得 5 的奖励; 到达 s_7 的时候, 可以得到 10 的奖励; 到达其他状态没有任何奖励。因为这里的状态是有限的, 所以我们可以用向量 $R = [5, 0, 0, 0, 0, 0, 10]$ 来表示奖励函数, R 表示每个状态的奖励大小。

我们通过一个形象的例子来理解马尔可夫奖励过程。我们把一艘纸船放到河流之中, 它就会随着水流而流动, 它自身是没有动力的。所以我们可以把马尔可夫奖励过程看成一个随波逐流的例子, 当我们从某

一个点开始的时候，纸船就会随着事先定义好的状态转移进行流动，它到达每个状态后，我们都有可能获得一些奖励。

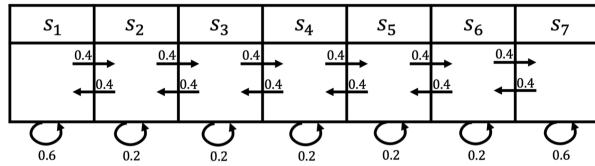


图 2.8 马尔可夫奖励过程的例子

2.3 马尔可夫决策过程

相对于马尔可夫奖励过程，马尔可夫决策过程多了决策（决策是指动作），其他的定义与马尔可夫奖励过程的是类似的。此外，状态转移也多了一个条件，变成了 $p(s_{t+1} = s' | s_t = s, a_t = a)$ 。未来的状态不仅依赖于当前的状态，也依赖于在当前状态智能体采取的动作。马尔可夫决策过程满足条件：

$$p(s_{t+1} | s_t, a_t) = p(s_{t+1} | h_t, a_t) \quad (2.16)$$

对于奖励函数，它也多了一个当前的动作，变成了 $R(s_t = s, a_t = a) = \mathbb{E}[r_t | s_t = s, a_t = a]$ 。当前的状态以及采取的动作会决定智能体在当前可能得到的奖励多少。

2.3.1 马尔可夫决策过程中的策略

策略定义了在某一个状态应该采取什么样的动作。知道当前状态后，我们可以把当前状态代入策略函数来得到一个概率，即

$$\pi(a | s) = p(a_t = a | s_t = s) \quad (2.17)$$

概率代表在所有可能的动作里面怎样采取行动，比如可能有 0.7 的概率往左走，有 0.3 的概率往右走，这是一个概率的表示。另外策略也可能是确定的，它有可能直接输出一个值，或者直接告诉我们当前应该采取什么样的动作，而不是一个动作的概率。假设概率函数是平稳的 (stationary)，不同时点，我们采取的动作其实都是在对策略函数进行采样。

已知马尔可夫决策过程和策略 π ，我们可以把马尔可夫决策过程转换成马尔可夫奖励过程。在马尔可夫决策过程里面，状态转移函数 $P(s'|s, a)$ 基于它当前的状态以及它当前的动作。因为我们现在已知策略函数，也就是已知在每一个状态下，可能采取的动作的概率，所以我们可以直接把动作进行加和，去掉 a ，这样我们就可以得到对于马尔可夫奖励过程的转移，这里就没有动作，即

$$P_\pi(s' | s) = \sum_{a \in A} \pi(a | s) p(s' | s, a) \quad (2.18)$$

对于奖励函数，我们也可以把动作去掉，这样就会得到类似于马尔可夫奖励过程的奖励函数，即

$$r_\pi(s) = \sum_{a \in A} \pi(a | s) R(s, a) \quad (2.19)$$

2.3.2 马尔可夫决策过程和马尔可夫过程/马尔可夫奖励过程的区别

马尔可夫决策过程里面的状态转移与马尔可夫奖励过程以及马尔可夫过程的状态转移的差异如图 2.9 所示。马尔可夫过程/马尔可夫奖励过程的状态转移是直接决定的。比如当前状态是 s ，那么直接通过转移概率决定下一个状态是什么。但对于马尔可夫决策过程，它的中间多了一层动作 a ，即智能体在当前状态的时候，首先要决定采取某一种动作，这样我们会到达某一个黑色的节点。到达这个黑色的节点后，因

为有一定的不确定性，所以当智能体当前状态以及智能体当前采取的动作决定过后，智能体进入未来的状态其实也是一个概率分布。在当前状态与未来状态转移过程中多了一层决策性，这是马尔可夫决策过程与之前的马尔可夫过程/马尔可夫奖励过程很不同的一点。在马尔可夫决策过程中，动作是由智能体决定的，智能体会采取动作来决定未来的状态转移。

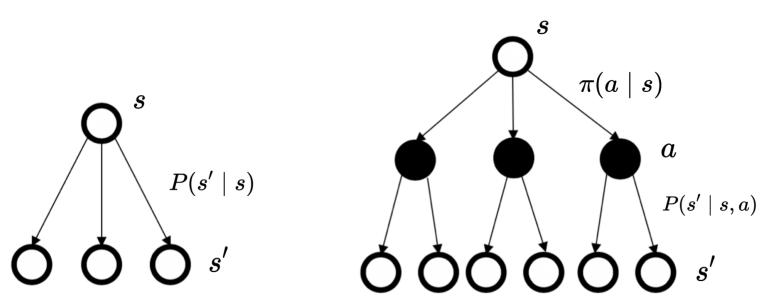


图 2.9 马尔可夫决策过程与马尔可夫过程/马尔可夫奖励过程的状态转移的对比

2.3.3 马尔可夫决策过程中的价值函数

马尔可夫决策过程中的价值函数可定义为

$$V_\pi(s) = \mathbb{E}_\pi [G_t \mid s_t = s] \quad (2.20)$$

其中，期望基于我们采取的策略。当策略决定后，我们通过对策略进行采样来得到一个期望，计算出它的价值函数。

这里我们另外引入了一个 **Q 函数 (Q-function)**。Q 函数也被称为**动作价值函数 (action-value function)**。Q 函数定义的是在某一个状态采取某一个动作，它有可能得到的回报的一个期望，即

$$Q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid s_t = s, a_t = a] \quad (2.21)$$

这里的期望其实也是基于策略函数的。所以我们需要对策略函数进行一个加和，然后得到它的价值。对 Q 函数中的动作进行加和，就可以得到价值函数：

$$V_\pi(s) = \sum_{a \in A} \pi(a \mid s) Q_\pi(s, a) \quad (2.22)$$

此处我们对 Q 函数的贝尔曼方程进行推导：

$$\begin{aligned}
 Q(s, a) &= \mathbb{E}[G_t \mid s_t = s, a_t = a] \\
 &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \mid s_t = s, a_t = a] \\
 &= \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a] + \gamma \mathbb{E}[r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \dots \mid s_t = s, a_t = a] \\
 &= R(s, a) + \gamma \mathbb{E}[G_{t+1} \mid s_t = s, a_t = a] \\
 &= R(s, a) + \gamma \mathbb{E}[V(s_{t+1}) \mid s_t = s, a_t = a] \\
 &= R(s, a) + \gamma \sum_{s' \in S} p(s' \mid s, a) V(s')
 \end{aligned} \quad (2.23)$$

2.3.4 贝尔曼期望方程

我们可以把状态价值函数和 Q 函数拆解成两个部分：即时奖励和后续状态的折扣价值 (discounted value of successor state)。通过对状态价值函数进行分解，我们就可以得到一个类似于之前马尔可夫奖励过程的贝尔曼方程——**贝尔曼期望方程 (Bellman expectation equation)**：

$$V_\pi(s) = \mathbb{E}_\pi [r_{t+1} + \gamma V_\pi(s_{t+1}) \mid s_t = s] \quad (2.24)$$

对于 Q 函数，我们也可以做类似的分解，得到 Q 函数的贝尔曼期望方程：

$$Q_\pi(s, a) = \mathbb{E}_\pi [r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a] \quad (2.25)$$

贝尔曼期望方程定义了当前状态与未来状态之间的关联。

我们进一步进行简单的分解，先给出式 (2.26)：

$$V_\pi(s) = \sum_{a \in A} \pi(a \mid s) Q_\pi(s, a) \quad (2.26)$$

接着，我们再给出式 (2.27)：

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' \mid s, a) V_\pi(s') \quad (2.27)$$

式 (2.26) 和式 (2.27) 代表状态价值函数与 Q 函数之间的关联。

我们把式 (2.27) 代入式 (2.26) 可得

$$V_\pi(s) = \sum_{a \in A} \pi(a \mid s) \left(R(s, a) + \gamma \sum_{s' \in S} p(s' \mid s, a) V_\pi(s') \right) \quad (2.28)$$

式 (2.28) 代表当前状态的价值与未来状态价值之间的关联。

我们把式 (2.26) 代入式 (2.27) 可得

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' \mid s, a) \sum_{a' \in A} \pi(a' \mid s') Q_\pi(s', a') \quad (2.29)$$

式 (2.29) 代表当前时刻的 Q 函数与未来时刻的 Q 函数之间的关联。

式 (2.28) 和式 (2.29) 是贝尔曼期望方程的另一种形式。

2.3.5 备份图

接下来我们介绍**备份 (backup)** 的概念。备份类似于自举之间的迭代关系，对于某一个状态，它的当前价值是与它的未来价值线性相关的。我们将与图 2.10 类似的图称为**备份图 (backup diagram)**，因为它们所示的关系构成了更新或备份操作的基础，而这些操作是强化学习方法的核心。这些操作将价值信息从一个状态（或状态-动作对）的后继状态（或状态-动作对）转移回它。每一个空心圆圈代表一个状态，每一个实心圆圈代表一个状态-动作对。

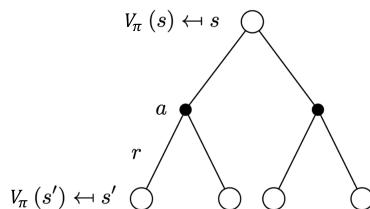


图 2.10 V_π 的备份图

如式 (2.30) 所示，这里有两层加和。第一层加和是对叶子节点进行加和，往上备份一层，我们就可以把未来的价值 (s' 的价值) 备份到黑色的节点。第二层加和是对动作进行加和，得到黑色节点的价值后，再往上备份一层，就会得到根节点的价值，即当前状态的价值。

$$V_\pi(s) = \sum_{a \in A} \pi(a \mid s) \left(R(s, a) + \gamma \sum_{s' \in S} p(s' \mid s, a) V_\pi(s') \right) \quad (2.30)$$

图 2.11 所示为状态价值函数的计算分解，图 2.11 B 的计算公式为

$$V_\pi(s) = \sum_{a \in A} \pi(a | s) Q_\pi(s, a) \quad (2.31)$$

图 2.11 B 给出了状态价值函数与 Q 函数之间的关系。图 2.11 C 计算 Q 函数为

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V_\pi(s') \quad (2.32)$$

我们将式 (2.32) 代入式 (2.31) 可得

$$V_\pi(s) = \sum_{a \in A} \pi(a | s) \left(R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V_\pi(s') \right) \quad (2.33)$$

所以备份图定义了未来下一时刻的状态价值函数与上一时刻的状态价值函数之间的关联。

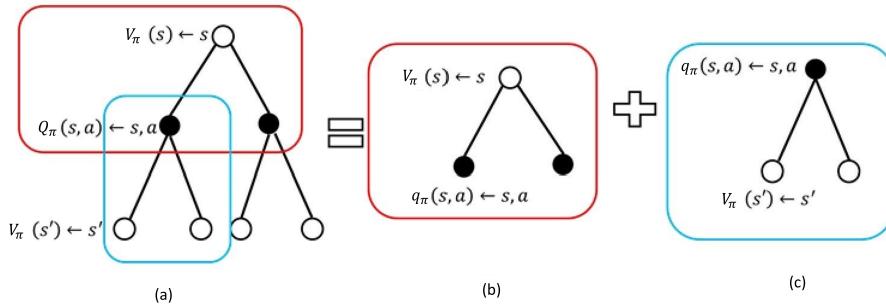


图 2.11 状态价值函数的计算分解

对于 Q 函数，我们也可以进行这样的一个推导。如图 2.12 所示，现在的根节点是 Q 函数的一个节点。Q 函数对应于黑色的节点。下一时刻的 Q 函数对应于叶子节点，有 4 个黑色的叶子节点。

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) \sum_{a' \in A} \pi(a' | s') Q_\pi(s', a') \quad (2.34)$$

如式 (2.34) 所示，这里也有两层加和。第一层加和先把叶子节点从黑色节点推到空心圆圈节点，进入到空心圆圈结点的状态。当我们到达某一个状态后，再对空心圆圈节点进行加和，这样就把空心圆圈节点重新推回到当前时刻的 Q 函数。

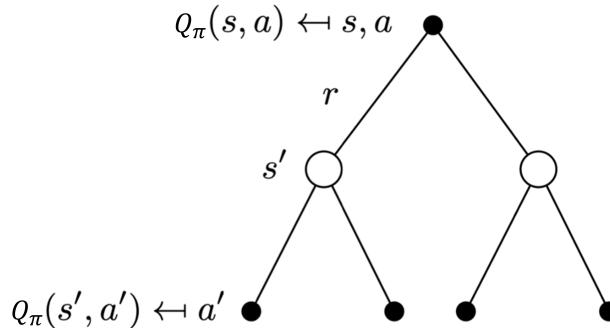


图 2.12 Q^π 的备份图

图 2.13 C 中，

$$V_\pi(s') = \sum_{a' \in A} \pi(a' | s') Q_\pi(s', a') \quad (2.35)$$

我们将式 (2.35) 代入式 (2.32) 可得未来 Q 函数与当前 Q 函数之间的关联，即

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) \sum_{a' \in A} \pi(a' | s') Q_\pi(s', a') \quad (2.36)$$

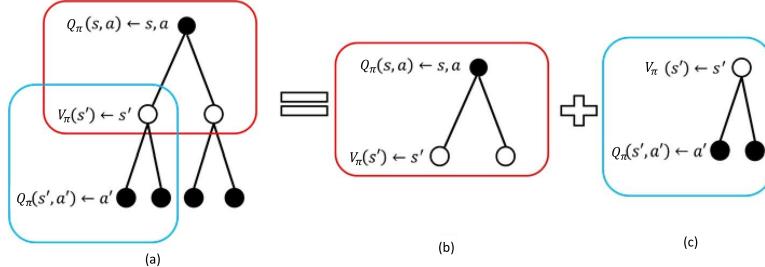


图 2.13 Q 函数的计算分解

2.3.6 策略评估

已知马尔可夫决策过程以及要采取的策略 π ，计算价值函数 $V_\pi(s)$ 的过程就是**策略评估**。策略评估在有些地方也被称为**(价值)预测** [**(value) prediction**]，也就是预测我们当前采取的策略最终会产生多少价值。如图 2.14a 所示，对于马尔可夫决策过程，我们其实可以把它想象成一个摆渡的人在船上，她可以控制船的移动，避免船随波逐流。因为在每一个时刻，摆渡的人采取的动作会决定船的方向。如图 2.14b 所示，对于马尔可夫奖励过程与马尔可夫过程，纸的小船会随波逐流，然后产生轨迹。马尔可夫决策过程的不同之处在于有一个智能体控制船，这样我们就可以尽可能多地获得奖励。



(a) 马尔可夫决策过程：人为控制船

(b) 马尔可夫过程/马尔可夫奖励过程：随波逐流

图 2.14 马尔可夫决策过程与马尔可夫过程/马尔可夫奖励过程的区别

我们再看一下策略评估的例子，探究怎么在决策过程中计算每一个状态的价值。如图 2.15 所示，假设环境里面有两种动作：往左走和往右走。现在的奖励函数应该是关于动作和状态两个变量的函数。但这里规定，不管智能体采取什么动作，只要到达状态 s_1 ，就有 5 的奖励；只要到达状态 s_7 ，就有 10 的奖励，到达其他状态没有奖励。我们可以将奖励函数表示为 $\mathbf{R} = [5, 0, 0, 0, 0, 0, 10]$ 。假设智能体现在采取一个策略：不管在任何状态，智能体采取的动作都是往左走，即采取的是确定性策略 $\pi(s) = \text{左}$ 。假设价值折扣因子 $\gamma = 0$ ，那么对于确定性策略，最后估算出的价值函数是一致的，即 $\mathbf{V}_\pi = [5, 0, 0, 0, 0, 0, 10]$ 。

s_1	s_2	s_3	s_4	s_5	s_6	s_7

图 2.15 策略评估示例

我们可以直接通过贝尔曼方程来得到价值函数：

$$V_\pi^k(s) = r(s, \pi(s)) + \gamma \sum_{s' \in S} p(s' | s, \pi(s)) V_\pi^{k-1}(s') \quad (2.37)$$

其中， k 是迭代次数。我们可以不停地迭代，最后价值函数会收敛。收敛之后，价值函数的值就是每一个状态的价值。

再来看一个例子，如果折扣因子 $\gamma = 0.5$ ，我们可以通过式 (2.38) 进行迭代：

$$V_\pi^t(s) = \sum_a p(\pi(s) = a) \left(r(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V_\pi^{t-1}(s') \right) \quad (2.38)$$

其中， t 是迭代次数。然后就可以得到它的状态价值。

最后，例如，我们现在采取随机策略，在每个状态下，有 0.5 的概率往左走，有 0.5 的概率往右走，即 $p(\pi(s) = \text{左}) = 0.5$, $p(\pi(s) = \text{右}) = 0.5$ ，如何求出这个策略下的状态价值呢？我们可以这样做：一开始的时候，我们对 $V(s')$ 进行初始化，不同的 $V(s')$ 都会有一个值；接着，我们将 $V(s')$ 代入贝尔曼期望方程里面进行迭代，就可以算出它的状态价值。

2.3.7 预测与控制

预测 (prediction) 和控制 (control) 是马尔可夫决策过程里面的核心问题。

预测（评估一个给定的策略）的输入是马尔可夫决策过程 $< S, A, P, R, \gamma >$ 和策略 π ，输出是价值函数 V_π 。预测是指给定一个马尔可夫决策过程以及一个策略 π ，计算它的价值函数，也就是计算每个状态的价值。

控制（搜索最佳策略）的输入是马尔可夫决策过程 $< S, A, P, R, \gamma >$ ，输出是最佳价值函数 (optimal value function) V^* 和最佳策略 (optimal policy) π^* 。控制就是我们去寻找一个最佳的策略，然后同时输出它的最佳价值函数以及最佳策略。

在马尔可夫决策过程里面，预测和控制都可以通过动态规划解决。要强调的是，这两者的区别就在于，预测问题是给定一个策略，我们要确定它的价值函数是多少。而控制问题是在没有策略的前提下，我们要确定最佳的价值函数以及对应的决策方案。实际上，这两者是递进的关系，在强化学习中，我们通过解决预测问题，进而解决控制问题。

举一个例子来说明预测与控制的区别。首先是预测问题。在图 2.16a 的方格中，我们规定从 $A \rightarrow A'$ 可以得到 +10 的奖励，从 $B \rightarrow B'$ 可以得到 +5 的奖励，其他步骤的奖励为 -1。如图 2.16b 所示，现在，我们给定一个策略：在任何状态下，智能体的动作模式都是随机的，也就是上、下、左、右的概率均为 0.25。预测问题要做的就是，求出在这种决策模式下，价值函数是什么。图 2.16c 是对应的价值函数。

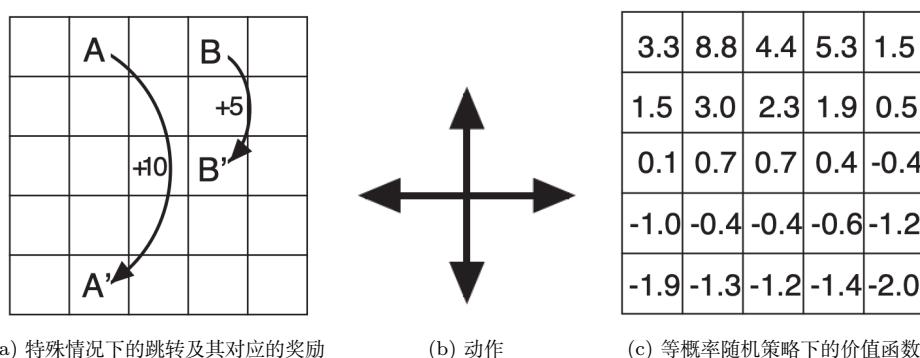


图 2.16 网格世界例子：预测^[3]

接着是控制问题。在控制问题中，问题背景与预测问题的相同，唯一的区别就是：不再限制策略。也就是动作模式是未知的，我们需要自己确定。所以我们通过解决控制问题，求得每一个状态的最优的价值

函数, 如图 2.17b 所示; 也得到了最优的策略, 如图 2.17c 所示。控制问题要做的就是, 给定同样的条件, 求出在所有可能的策略下最优的价值函数是什么, 最优策略是什么。

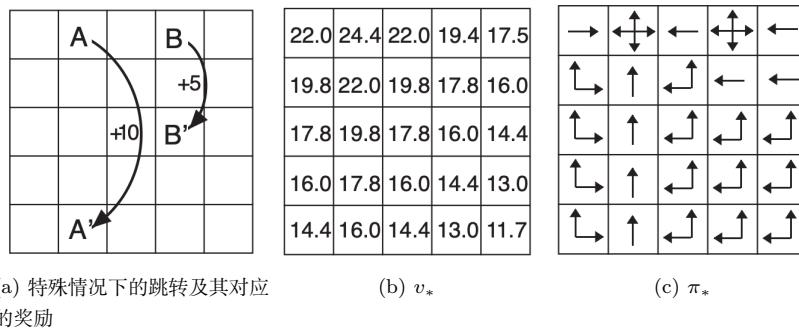


图 2.17 网格世界例子: 控制

2.3.8 动态规划

动态规划 (dynamic programming, DP) 适合解决满足**最优子结构** (optimal substructure) 和**重叠子问题** (overlapping subproblem) 两个性质的问题。最优子结构意味着, 问题可以拆分成一个个的小问题, 通过解决这些小问题, 我们能够组合小问题的答案, 得到原问题的答案, 即最优的解。重叠子问题意味着, 子问题出现多次, 并且子问题的解决方案能够被重复使用, 我们可以保存子问题的首次计算结果, 在再次需要时直接使用。

马尔可夫决策过程是满足动态规划的要求的, 在贝尔曼方程里面, 我们可以把它分解成递归的结构。当我们把它分解成递归的结构的时候, 如果子问题的子状态能得到一个值, 那么它的未来状态因为与子状态是直接相关的, 我们也可以将之推算出来。价值函数可以存储并重用子问题的最佳的解。动态规划应用于马尔可夫决策过程的规划问题而不是学习问题, 我们必须对环境是完全已知的, 才能做动态规划, 也就是要知道状态转移概率和对应的奖励。使用动态规划完成预测问题和控制问题的求解, 是解决马尔可夫决策过程预测问题和控制问题的非常有效的方式。

2.3.9 马尔可夫决策过程中的策略评估

策略评估就是给定马尔可夫决策过程和策略, 评估我们可以获得多少价值, 即对于当前策略, 我们可以得到多大的价值。我们可以直接把**贝尔曼期望备份** (Bellman expectation backup), 变成迭代的过程, 反复迭代直到收敛。这个迭代过程可以看作**同步备份** (synchronous backup) 的过程。

同步备份是指每一次的迭代都会完全更新所有的状态, 这对于程序资源的需求特别大。异步备份 (asynchronous backup) 的思想就是通过某种方式, 使得每一次迭代不需要更新所有的状态, 因为事实上, 很多状态也不需要被更新。

式 (2.39) 是指我们可以把贝尔曼期望备份转换成动态规划的迭代。当我们得到上一时刻的 V_t 的时候, 就可以通过递推的关系推出下一时刻的值。反复迭代, 最后 V 的值就是从 V_1 、 V_2 到最后收敛之后的值 V_π 。 V_π 就是当前给定的策略 π 对应的价值函数。

$$V^{t+1}(s) = \sum_{a \in A} \pi(a | s) \left(R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V^t(s') \right) \quad (2.39)$$

策略评估的核心思想就是把如式 (2.39) 所示的贝尔曼期望备份反复迭代, 然后得到一个收敛的价值函数的值。因为已经给定了策略函数, 所以我们可以直接把它简化成一个马尔可夫奖励过程的表达形式,

相当于把 a 去掉，即

$$V_{t+1}(s) = r_\pi(s) + \gamma P_\pi(s' | s) V_t(s') \quad (2.40)$$

这样迭代的式子中就只有价值函数与状态转移函数了。通过迭代式 (2.40)，我们也可以得到每个状态的价值。因为不管是在马尔可夫奖励过程，还是在马尔可夫决策过程中，价值函数 V 包含的变量都是只与状态有关，其表示智能体进入某一个状态，未来可能得到多大的价值。

比如现在的环境是一个小网格世界 (small gridworld)，智能体的目的是从某一个状态开始行走，然后到达终止状态，它的终止状态就是左上角与右下角 (如图 2.18 (右) 所示的阴影方块)。小网格世界总共有 14 个非终止状态：1, …, 14。我们把它的每个位置用一个状态来表示。

如图 2.18 (左) 所示，在小网格世界中，智能体的策略函数直接给定了，它在每一个状态都是随机行走，即在每一个状态都是上、下、左、右行走，采取均匀的随机策略 (uniform random policy)， $\pi(a | s) = 0.25$ 。它在边界状态的时候，比如在第 4 号状态的时候往左走，依然留在第 4 号状态。我们对其加了限制，这个限制就是出边界的动作不会改变状态，相应概率设置为 1，如 $p(7 | 7, r) = 1$ 。我们给出的奖励函数就是智能体每走一步，就会得到 -1 的奖励，也就是到达终止状态之前每走一步获得的奖励都是 -1 ，所以智能体需要尽快地到达终止状态。

给定动作之后状态之间的转移 (transition) 是确定的，例如 $p(2 | 6, u) = 2$ ，即从第 6 号状态往上走，它就会直接到达第 2 号状态。很多时候有些环境是概率性的 (probabilistic)，比如智能体在第 6 号状态，它选择往上走的时候，地板可能是滑的，然后它可能滑到第 3 号状态或者第 1 号状态，这就是有概率的转移。但我们把环境进行了简化，从 6 号状态往上走，它就到了第 2 号状态。因为我们已经知道环境中的每一个概率以及概率转移，所以就可以直接使用式 (2.40) 进行迭代，这样就会算出每一个状态的价值。

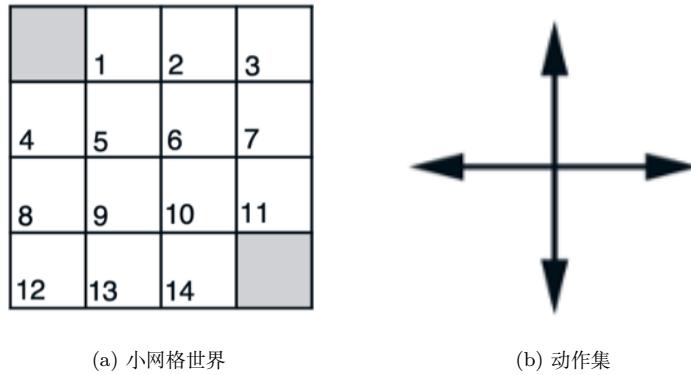


图 2.18 小网格世界环境

我们再来看一个动态的例子，推荐斯坦福大学的一个网页：https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html，这个网页模拟了式 (2.39) 所示的单步更新的过程中，所有格子的状态价值的变化过程。

如图 2.19a 所示，网格世界里面有很多格子，每个格子都代表一个状态。每个格子里都有一个初始值 0。每个格子里还有一个箭头，这个箭头是指智能体在当前状态应该采取什么策略。我们这里采取随机的策略，不管智能体在哪一个状态，它往上、下、左、右的概率都是相同的。比如在某个状态，智能体都有上、下、左、右各 0.25 的概率采取某一个动作，所以它的动作是完全随机的。在这样的环境里面，我们想计算每一个状态的价值。我们也定义了奖励函数，我们可以看到有些格子里面有一个 R 的值，比如有些值是负的。我们可以看到有几个格子里面是 -1 的奖励，只有一个 $+1$ 奖励的格子。在网格世界的中间位置，我们可以看到有一个 R 的值是 1。所以每个状态对应一个值，有一些状态没有任何值，它的奖励就为 0。

如图 2.19b 所示，我们开始策略评估，策略评估是一个不停迭代的过程。当我们初始化的时候，所有的 $V(s)$ 都是 0。我们现在迭代一次，迭代一次之后，有些状态的值已经产生了变化。比如有些状态的 R

值为 -1 , 迭代一次之后, 它就会得到 -1 的奖励。对于中间绿色的格子, 因为它的奖励为正, 所以它是值为 $+1$ 的状态。当迭代第 1 次的时候, 某些状态已经有些值的变化。

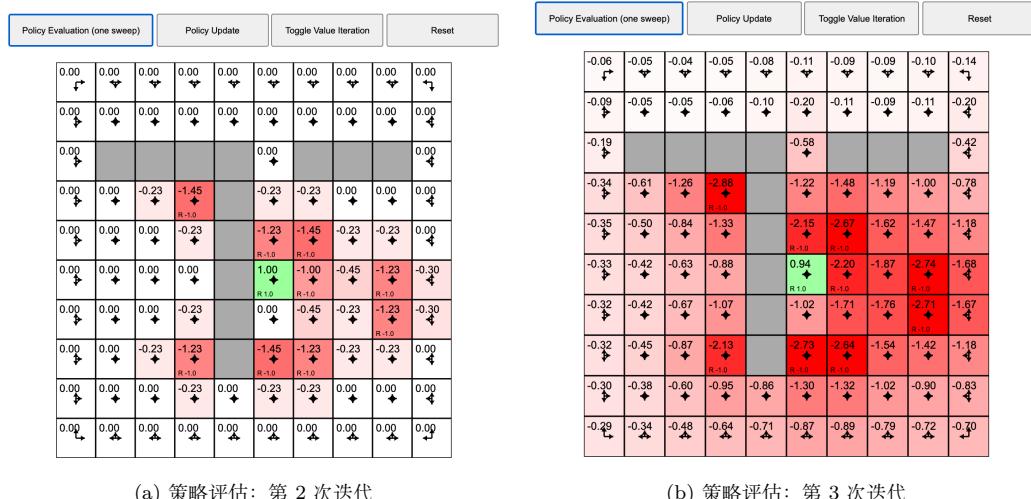


(a) 网格世界: 初始化界面

(b) 策略评估: 第 1 次迭代

图 2.19 网格世界: 动态规划示例

如图 2.20a 所示, 我们再迭代一次, 之前有值的状态的周围状态也开始有值。因为周围状态与之前有值的状态是临近的, 所以这就相当于把周围的状态转移过来。如图 2.20b 所示, 我们逐步迭代, 值是一直在变换的。



(a) 策略评估: 第 2 次迭代

(b) 策略评估: 第 3 次迭代

图 2.20 网格世界: 策略评估过程示例

当我们迭代了很多次之后, 有些很远的状态的价值函数已经有值了, 而且整个过程是一个呈逐渐扩散的过程, 这其实也是策略评估的可视化。当我们每一步进行迭代的时候, 远的状态就会得到一些值, 值从已经有奖励的状态逐渐扩散。当我们执行很多次迭代之后, 各个状态的值会逐渐稳定下来, 最后值就会确定不变。收敛之后, 每个状态的值就是它的状态价值。

2.3.10 马尔可夫决策过程控制

策略评估是指给定马尔可夫决策过程和策略, 我们可以估算出价值函数的值。如果我们只有马尔可夫决策过程, 那么应该如何寻找最佳的策略, 从而得到最佳价值函数 (optimal value function) 呢?

最佳价值函数的定义为

$$V^*(s) = \max_{\pi} V_{\pi}(s) \quad (2.41)$$

最佳价值函数是指，我们搜索一种策略 π 让每个状态的价值最大。 V^* 就是到达每一个状态，它的值的最大化情况。在这种最大化情况中，我们得到的策略就是最佳策略，即

$$\pi^*(s) = \arg \max_{\pi} V_{\pi}(s) \quad (2.42)$$

最佳策略使得每个状态的价值函数都取得最大值。所以如果我们可以得到一个最佳价值函数，就可以认为某个马尔可夫决策过程的环境可解。在这种情况下，最佳价值函数是一致的，环境中可达到的上限的值是一致的，但这里可能有多个最佳策略，多个最佳策略可以取得相同的最佳价值。

当取得最佳价值函数后，我们可以通过对 Q 函数进行最大化来得到最佳策略：

$$\pi^*(a | s) = \begin{cases} 1, & a = \arg \max_{a \in A} Q^*(s, a) \\ 0, & \text{其他} \end{cases} \quad (2.43)$$

当 Q 函数收敛后，因为 Q 函数是关于状态与动作的函数，所以如果在某个状态采取某个动作，可以使得 Q 函数最大化，那么这个动作就是最佳的动作。如果我们能优化出一个 Q 函数 $Q^*(s, a)$ ，就可以直接在 Q 函数中取一个让 Q 函数值最大化的动作的值，就可以提取出最佳策略。

Q: 怎样进行策略搜索？

A: 最简单的策略搜索方法就是穷举。假设状态和动作都是有限的，那么每个状态我们可以采取 A 种动作的策略，总共就是 $|A|^{|S|}$ 个可能的策略。我们可以把策略穷举一遍，算出每种策略的价值函数，对比一下就可以得到最佳策略。

但是穷举非常没有效率，所以我们要采取其他方法。搜索最佳策略有两种常用的方法：策略迭代和价值迭代。

寻找最佳策略的过程就是马尔可夫决策过程的控制过程。马尔可夫决策过程控制就是去寻找一个最佳策略使我们得到一个最大的价值函数值，即

$$\pi^*(s) = \arg \max_{\pi} V_{\pi}(s) \quad (2.44)$$

对于一个事先定好的马尔可夫决策过程，当智能体采取最佳策略的时候，最佳策略一般都是确定的，而且是稳定的（它不会随着时间的变化而变化）。但最佳策略不一定是唯一的，多种动作可能会取得相同的价值。

我们可以通过策略迭代和价值迭代来解决马尔可夫决策过程的控制问题。

2.3.11 策略迭代

策略迭代由两个步骤组成：策略评估和策略改进 (policy improvement)。如图 2.21a 所示，第一个步骤是策略评估，当前我们在优化策略 π ，在优化过程中得到一个最新的策略。我们先保证这个策略不变，然后估计它的价值，即给定当前的策略函数来估计状态价值函数。第二个步骤是策略改进，得到状态价值函数后，我们可以进一步推算出它的 Q 函数。得到 Q 函数后，我们直接对 Q 函数进行最大化，通过在 Q 函数做一个贪心的搜索来进一步改进策略。这两个步骤一直在迭代进行。所以如图 2.21b 所示，在策略迭代里面，在初始化的时候，我们有一个初始化的状态价值函数 V 和策略 π ，然后在这两个步骤之间迭代。图 2.21b 上面的线就是我们当前状态价值函数的值，下面的线是策略的值。策略迭代的过程与踢皮球一样。我们先给定当前已有的策略函数，计算它的状态价值函数。算出状态价值函数后，我们会得到一个 Q 函数。我们对 Q 函数采取贪心的策略，这样就像踢皮球，“踢”回策略。然后进一步改进策略，得到一个改进的策略后，它还不是最佳的策略，我们再进行策略评估，又会得到一个新的价值函数。基于这个新的价值函数再进行 Q 函数的最大化，这样逐渐迭代，状态价值函数和策略就会收敛。

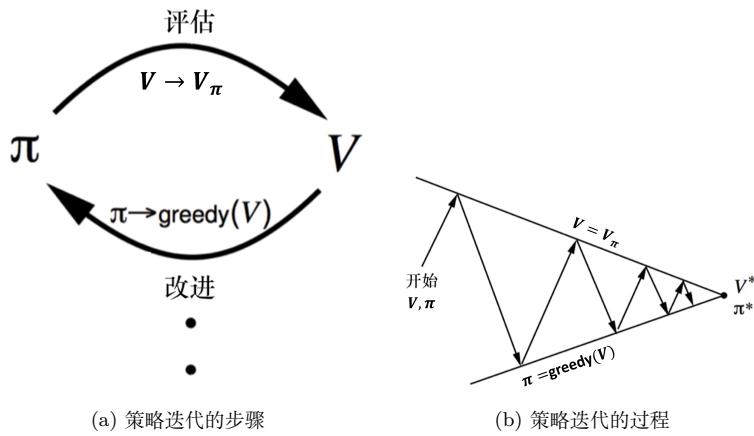


图 2.21 策略迭代

这里再来看一下第二个步骤——策略改进，看我们是如何改进策略的。得到状态价值函数后，我们就可以通过奖励函数以及状态转移函数来计算 Q 函数：

$$Q_{\pi_i}(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V_{\pi_i}(s') \quad (2.45)$$

对于每个状态，策略改进会得到它的新一轮的策略，对于每个状态，我们选取它得到最大值的动作，即

$$\pi_{i+1}(s) = \arg \max_a Q_{\pi_i}(s, a) \quad (2.46)$$

如图 2.22 所示，我们可以把 Q 函数看成一个 **Q 表格 (Q-table)**：横轴是它的所有状态，纵轴是它的可能的动作。如果我们得到了 Q 函数，Q 表格也就得到了。对于某个状态，每一列里面我们会取最大的值，最大值对应的动作就是它现在应该采取的动作。所以 $\arg \max$ 操作是指在每个状态里面采取一个动作，这个动作是能使这一列的 Q 函数值最大化的动作。

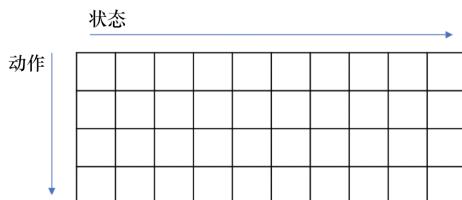


图 2.22 Q 表格

贝尔曼最优方程

当我们一直采取 $\arg \max$ 操作的时候，我们会得到一个单调的递增。通过采取这种贪心操作 ($\arg \max$ 操作)，我们就会得到更好的或者不变的策略，而不会使价值函数变差。所以当改进停止后，我们就会得到一个最佳策略。当改进停止后，我们取让 Q 函数值最大化的动作， Q 函数就会直接变成价值函数，即

$$Q_\pi(s, \pi'(s)) = \max_{a \in A} Q_\pi(s, a) = Q_\pi(s, \pi(s)) = V_\pi(s) \quad (2.47)$$

我们也就得到了贝尔曼最优方程 (Bellman optimality equation)

$$V_\pi(s) = \max_{a \in A} Q_\pi(s, a) \quad (2.48)$$

贝尔曼最优方程表明：最佳策略下的一个状态的价值必须等于在这个状态下采取最好动作得到的回报的期望。

当马尔可夫决策过程满足贝尔曼最优方程的时候，整个马尔可夫决策过程已经达到最佳的状态。只有当整个状态已经收敛后，我们得到最佳价值函数后，贝尔曼最优方程才会满足。满足贝尔曼最优方程后，我们可以采用最大化操作，即

$$V^*(s) = \max_a Q^*(s, a) \quad (2.49)$$

当我们取让 Q 函数值最大化的动作对应的值就是当前状态的最佳的价值函数的值。

另外，我们给出 Q 函数的贝尔曼方程

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V^*(s') \quad (2.50)$$

我们把式 (2.49) 代入式 (2.50) 可得

$$\begin{aligned} Q^*(s, a) &= R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V^*(s') \\ &= R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) \max_a Q^*(s', a') \end{aligned} \quad (2.51)$$

我们就可以得到 Q 函数之间的转移。

Q 学习是基于贝尔曼最优方程来进行的，当取 Q 函数值最大的状态 ($\max_{a'} Q^*(s', a')$) 的时候可得

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) \max_{a'} Q^*(s', a') \quad (2.52)$$

我们会在第三章介绍 Q 学习的具体内容。

我们还可以把式 (2.50) 代入式 (2.49) 可得

$$\begin{aligned} V^*(s) &= \max_a Q^*(s, a) \\ &= \max_a \mathbb{E}[G_t | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[r_{t+1}] + \max_a \mathbb{E}[\gamma V^*(s_{t+1}) | s_t = s, a_t = a] \\ &= \max_a R(s, a) + \max_a \gamma \sum_{s' \in S} p(s' | s, a) V^*(s') \\ &= \max_a \left(R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V^*(s') \right) \end{aligned} \quad (2.53)$$

我们就可以得到状态价值函数的转移。

2.3.12 价值迭代

1. 最优性原理

我们从另一个角度思考问题，动态规划的方法将优化问题分成两个部分。第一步执行的是最优的动作。之后后继的状态的每一步都按照最优的策略去做，最后的结果就是最优的。

最优性原理定理 (principle of optimality theorem)：一个策略 $\pi(s|a)$ 在状态 s 达到了最优价值，也就是 $V_\pi(s) = V^*(s)$ 成立，当且仅当对于任何能够从 s 到达的 s' ，都已经达到了最优价值。也就是对于所有的 s' ， $V_\pi(s') = V^*(s')$ 恒成立。

2. 确认性价值迭代

如果我们知道子问题 $V^*(s')$ 的最优解，就可以通过价值迭代来得到最优的 $V^*(s)$ 的解。价值迭代就是把贝尔曼最优方程当成一个更新规则来进行，即

$$V(s) \leftarrow \max_{a \in A} \left(R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V(s') \right) \quad (2.54)$$

只有当整个马尔可夫决策过程已经达到最佳的状态时，式 (2.54) 才满足。但我们可以把它转换成一个备份的等式。备份的等式就是一个迭代的等式。我们不停地迭代贝尔曼最优方程，价值函数就能逐渐趋向于最佳的价值函数，这是价值迭代算法的精髓。

为了得到最佳的 V^* ，对于每个状态的 V ，我们直接通过贝尔曼最优方程进行迭代，迭代多次之后，价值函数就会收敛。这种价值迭代算法也被称为确认性价值迭代 (deterministic value iteration)。

3. 价值迭代算法

价值迭代算法的过程如下。

(1) 初始化：令 $k = 1$ ，对于所有状态 s , $V_0(s) = 0$ 。

(2) 对于 $k = 1 : H$ (H 是迭代次数)

(a) 对于所有状态 s

$$Q_{k+1}(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V_k(s') \quad (2.55)$$

$$V_{k+1}(s) = \max_a Q_{k+1}(s, a) \quad (2.56)$$

(b) $k \leftarrow k + 1$ 。

(3) 在迭代后提取最优策略：

$$\pi(s) = \arg \max_a R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V_{k+1}(s') \quad (2.57)$$

我们使用价值迭代算法是为了得到最佳的策略 π 。我们可以使用式 (2.54) 进行迭代，迭代多次且收敛后得到的值就是最佳的价值。

价值迭代算法开始的时候，把所有值初始化，接着对每个状态进行迭代。我们把式 (2.55) 代入式 (2.56)，就可以得到式 (2.54)。因此，我们有了式 (2.55) 和式 (2.56) 后，就不停地迭代，迭代多次后价值函数就会收敛，收敛后就会得到 V^* 。我们有了 V^* 后，一个问题是如何进一步推算出它的最佳策略。我们可以直接用 $\arg \max$ 操作来提取最佳策略。我们先重构 Q 函数，重构后，每一列对应的 Q 值最大的动作就是最佳策略。这样我们就可以从最佳价值函数里面提取出最佳策略。我们只是在解决一个规划的问题，而不是强化学习的问题，因为我们知道环境如何变化。

价值迭代做的工作类似于价值的反向传播，每次迭代做一步传播，所以中间过程的策略和价值函数是没有意义的。而策略迭代的每一次迭代的结果都是有意义的，都是一个完整的策略。图 2.23 所示为一个可视化的求最短路径的过程，在一个网格世界中，我们设定了一个终点，也就是左上角的点。不管我们在哪一个位置开始，我们都希望能够到达终点（实际上这个终点在迭代过程中是不必要的，只是为了更好地演示）。价值迭代的迭代过程像是一个从某一个状态（这里是我们的终点）反向传播到其他各个状态的过程，因为每次迭代只能影响到与之直接相关的关系。让我们回忆一下最优性原理定理：如果我们某次迭代求解的某个状态 s 的价值函数 $V_{k+1}(s)$ 是最优解，它的前提是能够从该状态到达的所有状态 s' 都已经得到了最优解；如果不是，它所做的只是一个类似传递价值函数的过程。

如图 2.23 所示，实际上，对于每一个状态，我们都可以将其看成一个终点。迭代由每一个终点开始，我们每次都根据贝尔曼最优方程重新计算价值。如果它的相邻节点价值发生了变化，变得更好了，那么它

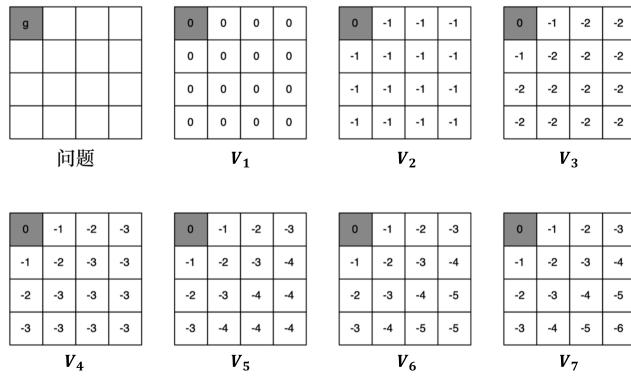


图 2.23 例子：最短路径

的价值也会变得更好，一直到相邻节点都不变。因此，在我们迭代到 V_7 之前，也就是还没将每个终点的最优的价值传递给其他的所有状态之前，中间的几个价值只是一种暂存的不完整的数据，它不能代表每一个状态的价值，所以生成的策略是没有意义的策略。价值迭代是一个迭代过程，图 2.23 可视化了从 V_1 到 V_7 每一个状态的价值的变化。而且因为智能体每走一步就会得到一个负的价值，所以它需要尽快地到达终点，可以发现离它越远的状态，价值就越小。 V_7 收敛过后，右下角的价值是 -6 ，相当于它要走 6 步，才能到达终点。智能体离终点越近，价值越大。当我们得到最优价值后，我们就可以通过策略提取来得到最佳策略。

2.3.13 策略迭代与价值迭代的区别

我们来看一个马尔可夫决策过程控制的动态演示，图 2.24 所示为网格世界的初始化界面。

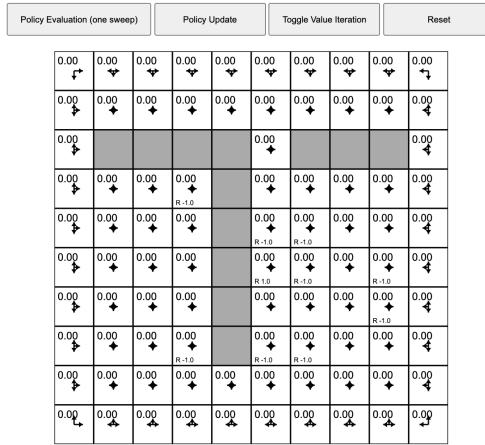


图 2.24 网格世界：初始化界面

首先我们来看看策略迭代，之前的例子在每个状态都采取固定的随机策略，每个状态都以 0.25 的概率往上、下、左、右，没有策略的改变。但是我们现在想进行策略迭代，每个状态的策略都进行改变。如图 2.25a 所示，我们先执行一次策略评估，得到价值函数，每个状态都有一个价值函数。如图 2.25b 所示，我们接着进行策略改进，单击“策略更新 (policy update)”，这时有些格子里面的策略已经产生变化。比如对于中间 -1 的这个状态，它的最佳策略是往下走。当我们到达 -1 状态后，我们应该往下走，这样就会得到最佳的价值。绿色右边的格子的策略也改变了，它现在选取的最佳策略是往左走，也就是在这个状态的时候，最佳策略应该是往左走。

如图 2.26a 所示，我们再执行下一轮的策略评估，格子里面的值又被改变了。多次之后，格子里面的值会收敛。如图 2.26b 所示，我们再次执行策略更新，每个状态里面的值基本都改变了，它们不再上、下、