第3章

强化学习初步

强化学习作为人工智能和机器学习的一个重要分支，近年来在各个领域都取得了令人瞩目的成就。从战胜世界顶级围棋选手的 AlphaGo，到自动驾驶汽车的决策系统，再到高效的推荐算法，强化学习正在深刻地改变着世界。

本章将为读者提供强化学习的基础知识和核心概念，帮助读者理解强化学习的工作原理，以及如何将其应用到实际问题中。本章的目标是让读者掌握强化学习的基本理论，并能够实现和应用一些经典的强化学习算法。

本章主要内容：

- 强化学习的基本概念和原理。
- 多臂赌博机问题及其解决方法。
- 马尔可夫决策过程(MDP)的基础理论。
- 动态规划方法在强化学习中的应用。

3.1 强化学习的基本概念和原理

3.1.1 监督学习、无监督学习与强化学习

创造能做出人类水平或优于人类水平决策的智能机器是许多科学家和工程师的梦想，这个梦想正逐渐接近现实。自图灵测试以来的几十年间，人工智能的研究和开发一直处于起伏不定的状态。最初的期望值非常高，例如，在 20 世纪 60 年代，赫伯特·西蒙(后来获得诺贝尔经济学奖)预测，机器将在 20 年内有能力完成人类能做的任何工作。这种令人兴奋的期望吸引了大量的政府和企业资金流入人工智能研究，但随后出现了巨大的失望，进入了一个被称为“人工智能冬天”的时期。几十年后，由于计算、数据和算法方面令人难以置信的发展，人类再次非常兴奋，比以往任何时候都更加追求人工智能的梦想。

在人们寻求开发达到或超过人类水平的人工智能系统的过程中，开发一个可以从自身经验中学习的模型显得尤为重要，而不一定需要一个监督者。强化学习(Reinforcement Learning，RL)是一个计算框架，使人们能够创建这样的智能代理。为了更好地理解 RL 的价值，将其与其他机器学习(ML)范式进行比较是很重要的。

如图 3.1 所示，RL 与监督学习(Supervised Learning，SL)和无监督学习(Unsupervised Learning，UL)一样，是 ML 的一个独立范式。它超越了其他两个范式所涉及的感知、分类、回归和聚类，并做出决策。更重要的是，RL 利用监督和无监督的 ML 方法来做这件事。因

此，RL 是一个与 SL 和 UL 不同但又密切相关的领域，掌握它们是很重要的。

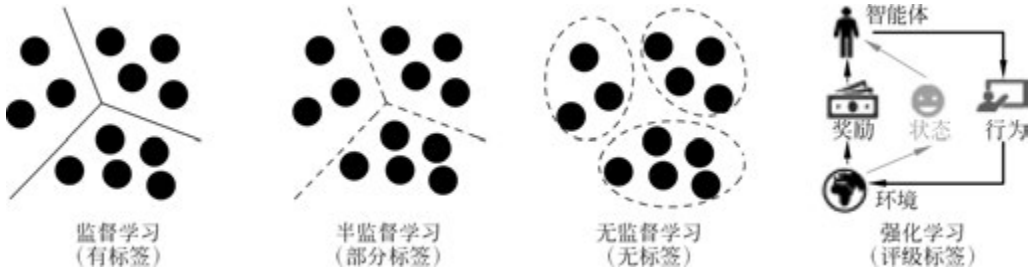

图 3.1　监督学习、无监督学习与强化学习

SL 是关于学习一个数学函数，尽可能准确地将一组输入映射到相应的输出/标签。这个想法是，我们不知道产生输出的过程的动态，但试图利用从它出来的数据来弄清楚它。例如，一种图像识别模型可将自动驾驶汽车摄像头上的物体分类为行人、停车标志、卡车等。一个预测模型，利用过去的销售数据预测客户对某一特定假日季节的需求。要想出精确的规则来对物体进行视觉上的区分，或者什么因素会导致客户对产品的需求，这是非常困难的。因此，SL 模型从标记的数据中推断出它们。SL 模型在训练期间从监督者（可以是人类专家或程序）提供的地面真实标签/输出中学习。在推理过程中，模型对给定输入的输出可能是什么做出预测，使用函数近似器来表示产生输出的过程的动态。

UL 算法可以识别数据中以前未知的模式。在使用这些模型时，我们可能对预期的结果有一个想法，但并没有给模型提供标签。例如，识别由摄像机提供的图像上的同质段，即自动驾驶汽车。该模型有可能根据图像上的纹理将天空、道路、建筑物等分开。根据销售量将每周的销售数据归类为三组。输出的可能是销售量低、中、高的周。UL 模型不知道地面真相是什么，也没有标签来映射输入。它们只是识别数据中的不同模式。在推理过程中，模型会将输入的内容归入它所确定的一个组，不知道这个组代表什么。函数近似器，如神经网络，在一些 UL 算法中使用，但并不总是如此。

RL 是一个框架，学习如何在不确定的情况下做出决定，通过实验和错误使长期的利益最大化。这些决策是按顺序做出的，早期的决策会影响以后遇到的情况和利益。这将 RL 与 SL 和 UL 区分开来，后者不涉及任何决策。例如，对于一辆自动驾驶汽车，鉴于摄像头上所有物体的类型和位置，以及道路上车道的边缘，模型可能会学习如何转向方向盘，以及汽车的速度应该是多少，以便安全地、尽可能快地超过前面的汽车。再如，考虑到一种产品的历史销售数字以及将库存从供应商运到商店所需的时间，该模型可能会学习何时以及向供应商订购多少单位，以使季节性的客户需求很可能得到满足，同时使库存和运输成本最小化。

RL 模型的输出是一个给定的决策，而不是预测或聚类。没有监督者提供的基础真理决定，告诉模型在不同情况下的理想决定是什么。相反，模型从自己的经验和过去的决定的反馈中学习最佳决定。例如，通过实验和错误，RL 模型会了解到，在超车时超速可能会导致事故，而在假期前订购过多的库存会导致以后的库存过剩。RL 模型经常使用 SL 模型的输出作为输入来做决策。例如，自动驾驶汽车中的图像识别模型的输出可以用来做出驾驶决策。同样地，预测模型的输出常常被用作做出库存补充决策的 RL 模型的输入。即使没有来自辅助模型的这种输入，RL 模型也会隐含地或明确地预测其决策在未来会导致什

么情况。RL 利用了许多为 SL 和 UL 开发的方法，如各种类型的神经网络作为函数近似器。

因此，RL 与其他 ML 方法的区别在于，它是一个决策框架。不过，令人兴奋的是它与我们作为人类如何从经验中学习决策的相似之处。想象一下，一个蹒跚学步的孩子正在学习如何用玩具积木搭建一座塔。通常情况下，塔越高，幼儿就越高兴。每一个高度的增加都是一种成功。每一次倒塌都是一次失败。他们很快发现，下一个积木越靠近下面的积木中心，塔就越稳定。当一个离边缘太近的积木更容易倒塌时，这一点得到了加强。通过练习，他们设法将几个积木叠在一起。他们意识到，他们如何堆叠先前的积木，形成了一个基础，决定了他们能建多高的塔。因此，他们学会了。

就像蹒跚学步的幼儿并非依靠蓝图，而是从搭高积木的成功和塔倒塌的失败中获得反馈、学习策略一样，强化学习（RL）的核心正是这种“从经验而非指令中学习决策”的范式，这使其区别于主要关注模式识别的监督学习和无监督学习。如同幼儿通过反复试错来追求更高的塔（即奖励），RL 代理也是在探索和试验中，利用反馈信号来优化能最大化长期回报的序贯决策。因此，强化学习之所以被视为一种深刻且强大的人工智能方法，关键在于其学习机制——这种通过试错、反馈和经验积累来优化决策的过程，与我们人类探索世界、掌握技能的基础学习方式惊人地相似，深刻触及了智能体与环境交互学习的本质。

3.1.2 强化学习的定义及其基本原理

强化学习（RL）是机器学习的一个重要分支，它关注如何使智能体（Agent）在与环境的交互中学习做出最优决策。与其他机器学习方法不同，强化学习更接近人类和动物的学习方式，通过尝试和错误来学习如何最大化长期收益。

强化学习可以定义为：一种计算方法，旨在学习如何将情境映射到动作，以最大化数值化的奖励信号。

如图 3.2 所示，强化学习的基本工作原理是：智能体观察环境的状态（state s_t），根据当前策略选择一个动作（action a_t），执行该动作，环境转换到新的状态（state s_{t+1}）。环境返回一个奖励信号（reward r_t），智能体根据新状态和奖励更新其知识和策略，重复上述步骤，直到达到终止条件。这个过程形成了一个反馈循环，使得智能体能够不断地改进其行为策略。

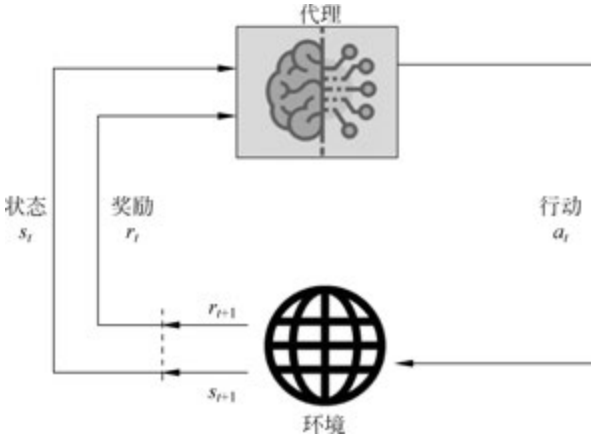

图 3.2 强化学习基本原理图

强化学习在人工智能领域占有独特的地位，因为它不仅是实现通用人工智能的潜在途径之一，还能够处理复杂的决策问题，如围棋和机器人控制等。这些问题通常具有高度的不确定性和动态变化的环境，使得传统的监督学习和无监督学习方法难以有效应对。强化学习通过智能体与环境的持续交互，逐步改进其决策策略，从而在长期内最大化收益。此外，强化学习提供了一种自主学习的范式，无须依赖人工标注的数据，使得智能体能够通过实验和错误，自我调整和优化其行为策略。这种自主学习能力使得强化学习在诸多领域中展现出巨大的潜力，如自动驾驶、资源管理、游戏 AI 等，从而推动了人工智能技术的不断进步和发展。

强化学习的基本元素如下。

(1) 智能体(Agent)。

智能体是强化学习系统中的学习和决策主体，它通过与环境交互来学习最优策略。智能体的特点包括能够感知环境状态、能够做出决策(选择动作)，并且具有根据经验调整策略的学习能力。

(2) 环境(Environment)。

环境是智能体所处的外部世界，它定义了问题的背景和规则。环境的特点包括可以被智能体观察和影响、根据智能体的动作改变状态，并且提供奖励信号。

(3) 状态(State)。

状态是对环境在特定时刻的描述。它可以是完全可观察的，即智能体可以直接获取环境的完整信息；也可以是部分可观察的，即智能体只能获取环境的部分信息。

(4) 动作(Action)。

动作是智能体可以执行的操作，用来影响环境。动作可以是离散的，例如，在棋类游戏中选择落子位置；也可以是连续的，例如，控制机器人关节的角度。

(5) 奖励(Reward)。

奖励是环境对智能体动作的即时反馈，是一个标量值。奖励函数定义了任务的目标，是强化学习的核心概念。奖励信号指导智能体调整其策略，以最大化长期收益。

强化学习具有几个独特之处，使其在解决复杂决策问题时表现出色。首先，强化学习能够处理延迟奖励的问题，这在现实世界的决策中非常常见。智能体在执行一系列动作后，可能需要等待一段时间才能收到最终的奖励反馈。其次，强化学习需要在探索与利用之间取得平衡。探索意味着尝试新的动作，以发现潜在的更优解；而利用则是选择已知的最佳动作，以获得稳定的收益。有效地平衡探索与利用是强化学习中的一个核心挑战。最后，强化学习特别适合解决需要做出一系列决策的问题。在许多应用场景中，单一决策往往不足以实现目标，而是需要智能体在多个时刻做出连贯的决策，以达到长期的最优效果。序列决策的能力使强化学习在复杂任务中具有显著优势。

3.1.3 强化学习的典型应用场景

随着计算能力的提升和算法的不断改进，强化学习在游戏 AI、机器人控制、推荐系统、自动驾驶等诸多领域展现出了广泛的应用前景，为实现通用人工智能(AGI)奠定了坚实的基础。

1. 游戏 AI

游戏 AI 是强化学习的经典应用场景之一。知名案例包括 AlphaGo 和 OpenAI Five。强化学习在游戏 AI 中的应用特点是规则明确、状态空间大，并且需要制定长期策略。通过强化学习，智能体可以在复杂的游戏环境中不断学习和优化其策略，从而在游戏中取得优势。这些应用不仅在技术上取得了突破，也为强化学习在其他领域的应用提供了宝贵的经验和启示。

2. 机器人控制

机器人控制是另一个重要的应用领域，涵盖了工业机器人和家庭服务机器人。该领域的特点是连续动作空间，需要智能体能够进行实时反应，并且安全性要求极高。通过强化学习，机器人可以在多变的环境中自主调整其动作，提高操作的精度和效率。例如，在工业环境中，机器人可以学习如何最有效地组装产品，而在家庭环境中，服务机器人可以学习如何更好地完成清洁任务和其他家务。

3. 推荐系统

在电商和内容推荐系统中，强化学习被广泛应用。这些系统需要处理大规模的用户和物品数据，并在平衡短期和长期目标之间做出决策。通过强化学习，推荐系统可以不断优化推荐效果，提高用户满意度和平台收益。例如，电商平台可以通过强化学习优化推荐算法，提升商品的曝光率和销售量；内容平台则可以通过优化推荐策略，提高用户的黏性和活跃度。

4. 自动驾驶

自动驾驶是强化学习的一个前沿应用，涉及路径规划和障碍物避让等任务。该领域的特点是高度动态的环境，安全性至关重要，并且需要多传感器的融合。通过强化学习，自动驾驶系统可以在复杂的道路环境中自主学习驾驶策略，提升行车安全性和效率。例如，自动驾驶汽车可以学习如何在复杂的交通环境中安全行驶，如何应对突发情况，以及如何优化行车路线以提高燃油效率和行驶速度。

5. 金融交易

在金融领域，强化学习被用于优化交易策略和资产管理。通过分析市场数据和历史交易记录，强化学习算法可以学习和预测市场趋势，制定出最优的交易策略。该应用的特点是高频数据、非线性关系和高风险，因此对算法的稳定性和准确性要求非常高。例如，强化学习可以用于股票交易中的买卖决策、风险管理和组合优化，从而帮助投资者在动态的市场环境中获取最大收益。

6. 医疗健康

在医疗健康领域，强化学习被用于个性化治疗方案的制定、药物发现和医疗资源的优化配置。通过分析病人的病历数据和治疗效果，强化学习算法可以帮助医生制定最优的治疗方案，优化治疗效果。例如，强化学习可以用于优化癌症治疗中的放疗和化疗方案，帮助医生根据病人的具体情况调整治疗策略，从而提高治疗效果和病人存活率。

7. 资源管理

在资源管理领域，强化学习被用于优化资源分配和调度，提高资源利用效率和降低成本。通过分析资源使用数据，强化学习算法可以优化资源分配策略，提高资源利用率。例如，强化学习可以用于优化云计算中的资源分配，帮助云服务提供商根据用户需求调整资

源分配策略，从而提高资源利用率，降低运营成本。

3.2 强化学习案例分析：多臂老虎机

当你登录你最喜欢的社交媒体应用程序时，可能会看到正在测试的多个版本之一；当你访问一个网站时，显示给你的广告是根据你的个人资料定制的；在许多在线购物平台上，价格是动态确定的。这些应用有一个共同点：它们通常被建模为多臂老虎机（Multi-Armed Bandit，MAB）问题，以确定最佳决策。

MAB 问题是强化学习（RL）的一种形式，其中，代理在单步问题中做出决策。目标是最大化即时奖励，而不考虑后续步骤的后果。虽然这是对多步 RL 的简化，但代理仍然必须处理 RL 的一个基本权衡：探索可能带来更高奖励的新动作与利用已知的好动作。

3.2.1 探索与利用的权衡

探索对于发现潜在的更优选项至关重要。在不完全信息的环境中，仅依赖于已知信息可能会错过更好的机会。通过探索，智能体能够扩大知识面，发现未知的高回报选项。例如，在一个广告投放场景中，探索不同的广告创意可以帮助找到点击率更高的版本。此外，探索还能减少对初始估计的依赖，提高决策的鲁棒性。在动态环境中，探索能够帮助智能体适应变化，及时调整策略，以应对环境的变化。

尽管探索非常重要，利用已知的最佳选项同样不可或缺。利用可以确保获得稳定的回报，特别是在资源有限的情况下，可以最大化短期收益。例如，在广告投放中，利用当前已知的高点击率广告可以带来即时的商业价值。此外，利用能够帮助智能体在某些情况下快速收敛到一个好的解决方案，从而节省时间和资源。因此，在实际应用中，利用已知信息是保证效益的重要手段。

找到探索与利用之间的最佳平衡是一个复杂的问题。过度探索可能会浪费资源在次优选项上，而过度利用则可能错过更好的机会。理想的策略应该能够在初期进行足够的探索以获得准确的估计，随着时间推移逐渐倾向于利用，以最大化长期收益。同时，策略还应根据环境的不确定性和动态性适当调整探索的程度。比如在广告投放中，初期可以尝试多种广告创意，随着数据积累逐渐集中资源于效果最好的创意。这样既保证了前期的充分探索，又能在后期通过利用获得最大化的收益。

探索与利用的平衡不仅是一个理论问题，更是在实际应用中需要不断调整和优化的实践过程。智能体需要动态调整策略，根据实时反馈优化决策过程，从而在复杂的环境中实现最优表现。

下面以多臂老虎机问题为例，来进一步说明探索与利用的机制。

如图 3.3 所示，多臂老虎机问题源于这样一个场景：一个赌徒站在一排老虎机前，每个老虎机有不同的概率分布，决定是否会给出奖励。赌徒的目标是最大化获得的总奖励，但他在初始时对各个老虎机的奖励分布一无所知。为了实现这一目标，赌徒需要在探索和利用之间找到一个平衡。

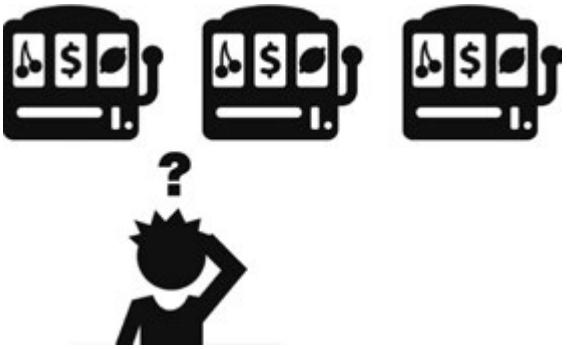

图 3.3　MAB 问题

在 MAB 问题中，探索是指尝试不同的老虎机，即便这些老虎机在过去的尝试中并没有给出高奖励。探索对于发现潜在的高回报老虎机至关重要。假设赌徒只尝试过几个老虎机，其中一个给了相对较高的奖励，他可能会倾向于一直使用这个老虎机。然而，其他未尝试的老虎机可能会有更高的奖励概率。如果不进行探索，赌徒可能会错过这些更好的机会。

通过探索，赌徒能够扩大知识面，发现那些可能带来更高回报的老虎机。此外，探索还减少了对初始估计的依赖，提高了决策的鲁棒性。在一个动态环境中，例如，老虎机的奖励概率可能随时间变化，探索可以帮助赌徒及时调整策略，适应新的环境。

尽管探索很重要，利用已知的最佳选项同样关键。利用指的是选择当前已知的回报最高的老虎机，以确保获得稳定的回报。在资源有限的情况下，例如赌徒只有有限的赌注，利用能够帮助他最大化短期收益。例如，赌徒已经发现某个特定的老虎机在过去的尝试中给出了较高的奖励，他可以继续使用这个老虎机来确保获得稳定的回报。

利用能够帮助赌徒在某些情况下快速收敛到一个好的解决方案，从而节省时间和资源。在多臂老虎机问题中，找到一个已经表现出高回报的老虎机并持续利用，可以帮助赌徒在短期内获得较高的累计奖励。

找到探索与利用之间的最佳平衡是一个复杂的问题。过度探索可能会浪费资源在次优选项上，例如，尝试过多表现不佳的老虎机；而过度利用则可能错过更好的机会，例如，一直使用一个稍好但并非最优的老虎机。

理想的策略应该能够在初期进行足够的探索，以获得准确的奖励分布估计。随着时间推移，策略应逐渐倾向于利用，以最大化长期收益。例如，赌徒可以在初期尝试每个老虎机几次，收集初步的奖励信息。然后，逐渐增加在那些表现较好的老虎机上的尝试次数，从而平衡探索和利用。

同时，策略还应根据环境的不确定性和动态性适当调整探索的程度。例如，如果发现某个老虎机的奖励概率发生了变化，赌徒需要重新进行探索，以适应新的环境。通过不断调整和优化，赌徒能够在复杂的环境中实现最优表现。

总结来说，探索与利用的平衡不仅是一个理论问题，更是在实际应用中需要不断调整和优化的实践过程。在多臂老虎机问题中，通过合理地平衡探索和利用，赌徒能够在长期内实现最大的累计奖励。

3.2.2 ε-贪婪算法

ε-贪婪算法是一种简单而有效的策略，用于在多臂老虎机问题中平衡探索与利用。算法的核心思想是通过引入一个参数，在探索和利用之间进行权衡。在每一轮决策中，算法以概率 ε 选择一个随机动作（探索），以概率 1−ε 选择当前估计最优的动作（利用）。这种方法确保了即使在利用阶段，智能体仍然有机会进行探索，从而避免陷入次优解。

形式化表示如下。

$$a_t = \begin{cases} \text{随机动作}, & \text{概率} = \varepsilon \\ \arg\max\limits_a Q_t(a), & \text{概率} = 1-\varepsilon \end{cases}$$

其中，a_t 是 t 时刻选择的动作，$Q_t(a)$是对动作 a 的价值估计。

参数 ε 控制了探索与利用的比例。较大的 ε 值意味着更多的探索，较小的 ε 值则意味着更多的利用。典型的选择是 ε 在 0.1 到 0.01 之间。随着时间的推移，ε 可以逐渐减小，从而在初期进行更多探索，而在后期逐渐集中于利用。例如，可以使用一种衰减策略，将 ε 随着时间的推移逐渐减小：

$$\varepsilon_t = \frac{\varepsilon_0}{1 + \text{decay rate} \times t}$$

ε-算法的优点是：简单易实现，计算复杂度低；保证了持续的探索，能够适应动态环境；适用于许多实际应用场景，如在线广告推荐。ε-算法的缺点是：探索是盲目的，不考虑各个动作的不确定性；固定的值可能不适合所有阶段，需根据实际情况调整；在过度探索或利用情况下，可能导致次优解。

3.2.3 上置信界算法

上置信界（Upper Confidence Bound，UCB）算法是一种更智能的探索方法，考虑了每个动作的不确定性。UCB 算法通过计算每个动作的上置信界，将当前估计的期望回报与一个反映不确定性的置信项相结合，从而选择具有最高上置信界的动作。这样，算法既能够利用当前已知的最佳动作，又能优先探索那些不确定性较高的动作。

UCB1 算法是 UCB 算法的一种经典实现，选择规则如下。

$$a_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

其中，$Q_t(a)$是动作 a 的当前估计价值；$N_t(a)$是到时刻 t 为止动作 a 被选择的次数；c 是一个控制探索程度的参数，通常是一个正数。

在初期，所有动作都会被选择至少一次以初始化 $N_t(a)$。随着时间的推移，算法会根据公式计算每个动作的上置信界，并选择上置信界最高的动作。置信项 $c\sqrt{\frac{\ln t}{N_t(a)}}$ 随着选择次数的增加而减小，鼓励对不常选择的动作进行更多的探索。

UCB 算法的优势是：智能探索，优先探索不确定性高的动作；理论上保证在某些条件下能达到最优的对数级别的遗憾上界；自适应性强，能够根据环境动态调整策略。UCB 算法的缺点是：需要调整参数 c，选择不当可能影响算法性能；在非常动态的环境中，可能需

要更复杂的变种算法；实现较为复杂，相比 ε-贪婪算法计算开销更大。

通过 ε-贪婪算法和 UCB 算法的介绍，可以看出它们在平衡探索与利用方面各有特点和适用场景。在实际应用中，根据具体需求选择合适的算法能够显著提升决策效果。

3.2.4 案例研究：简单的在线广告优化问题

假设一个在线广告平台有 5 个不同的广告版本（A，B，C，D，E）可以展示给用户。每个广告版本有其自身的点击率（Click-Through Rate，CTR），但这个 CTR 对广告平台是未知的。平台的目标是通过选择最佳的广告版本来最大化总点击数。

在多臂老虎机（MAB）问题中，有多个选项可供选择，如 5 个不同的广告版本。每个广告版本的实际点击率（CTR）是未知的，这引入了不确定性。平台需要反复决定展示哪个广告版本，这涉及序列决策的问题。为了做出最佳决策，平台必须在尝试新广告（探索）和选择已知效果好的广告（利用）之间取得平衡。最终目标是最大化累积点击数。

在线广告优化问题符合上述问题的特征，因此，被归类为 MAB 问题。首先，每个广告版本可以被视为一个“臂”（选项）。展示一个广告并观察是否被点击相当于“拉动”一个臂并获得奖励。CTR 的不确定性对应于每个臂的奖励分布的不确定性。平台需要通过反复选择和学习来找到最佳策略，这符合 MAB 的序列决策特性。平台面临探索（尝试不同广告以学习其 CTR）和利用（选择已知 CTR 高的广告）之间的权衡，这正是 MAB 问题的核心。

首先，创建一个简单的环境来模拟广告展示。

【编程示例 3.1】 在线广告优化。

```
import numpy as np
import matplotlib.pyplot as plt
class Ad:
    def __init__(self, ctr):
        self.ctr = ctr
    def show(self):
        return np.random.random() < self.ctr
#创建 5 个广告，每个都有不同的 CTR
ads = [Ad(0.1), Ad(0.2), Ad(0.3), Ad(0.25), Ad(0.15)]
```

上述代码创建了 5 个广告对象，每个都有一个预定义的 CTR。show()方法模拟广告展示，根据 CTR 返回是否被点击。接下来，实现 ε-贪婪算法。

```
def epsilon_greedy(ads, num_trials, epsilon):
    num_ads = len(ads)
    clicks = np.zeros(num_ads)
    impressions = np.zeros(num_ads)
    for _ in range(num_trials):
        if np.random.random() < epsilon:
            ad = np.random.randint(num_ads)                          #探索
        else:
            ad = np.argmax(clicks / np.maximum(impressions, 1))      #利用
        reward = ads[ad].show()
        clicks[ad] += reward
        impressions[ad] += 1
    return clicks, impressions
```

```
#运行ε-贪婪算法
epsilon = 0.1
num_trials = 10000
clicks, impressions = epsilon_greedy(ads, num_trials, epsilon)
```

上边这段代码实现了一个ε-贪婪算法，用于解决多臂老虎机(MAB)问题，以最大化点击率(CTR)。程序首先初始化广告列表(ads)、总运行次数(num_trials)和探索概率(epsilon)。通过设置点击次数(clicks)和展示次数(impressions)的初始值为零，程序在每次循环中模拟一次广告展示。在每次迭代中，程序生成一个随机数以决定是否进行探索(以概率ε随机选择一个广告)或利用(以概率1—ε选择当前点击率最高的广告)。选定广告的show()方法返回点击奖励，随后更新该广告的点击次数和展示次数。最终，算法返回每个广告的点击次数和展示次数，从而帮助找到最优广告版本。这个算法通过随机探索和选择当前最优广告(利用)来不断改进策略，以期在多次展示后找到最优的广告版本，从而最大化累积点击率。

下面使用另一个算法，即上置信界(UCB)算法来最大化点击率(CTR)。

```
def ucb(ads, num_trials):
    num_ads = len(ads)
    clicks = np.zeros(num_ads)
    impressions = np.zeros(num_ads)
    for t in range(1, num_trials + 1):
        ucb_values = np.zeros(num_ads)
        for i in range(num_ads):
            if impressions[i] > 0:
                mean = clicks[i] / impressions[i]
                confidence = np.sqrt(2 * np.log(t) / impressions[i])
                ucb_values[i] = mean + confidence
            else:
                ucb_values[i] = float('inf')
        ad = np.argmax(ucb_values)
        reward = ads[ad].show()
        clicks[ad] += reward
        impressions[ad] += 1
    return clicks, impressions
#运行UCB算法
clicks_ucb, impressions_ucb = ucb(ads, num_trials)
```

程序首先初始化广告列表(ads)、总运行次数(num_trials)，并设置点击次数(clicks)和展示次数(impressions)的初始值为零。然后在每次迭代中，程序计算每个广告的UCB值。对于已经展示过的广告，计算其平均点击率(mean)和置信区间(confidence)，并将两者相加得到UCB值；对于尚未展示过的广告，将其UCB值设为无穷大以确保其被选择。在选择UCB值最高的广告后，通过调用广告的show()方法获取点击奖励，并更新该广告的点击次数和展示次数。该算法在不断平衡探索与利用的过程中，通过选择具有最高UCB值的广告逐步优化策略，以期在多次展示后找到最优的广告版本，从而实现点击率的最大化。最终，算法返回每个广告的点击次数和展示次数。

最后，通过比较两种算法的累积点击率来评估它们的性能。代码如下。

```
def plot_results(clicks_eg, impressions_eg, clicks_ucb, impressions_ucb):
    plt.figure(figsize = (10, 6))
    #绘制 ε - Greedy 算法的累积点击率，使用实线
    plt.plot(np.cumsum(clicks_eg) / np.cumsum(impressions_eg), label = 'ε - Greedy', linestyle = '-')
    #绘制 UCB 算法的累积点击率，使用虚线
    plt.plot(np.cumsum(clicks_ucb) / np.cumsum(impressions_ucb), label = 'UCB', linestyle = '--')
    plt.xlabel('展示次数')
    plt.ylabel('累积点击率')
    plt.legend()
    plt.show()
#调用函数绘制结果
plot_results(clicks, impressions, clicks_ucb, impressions_ucb)
```

比较的结果如图 3.4 所示。

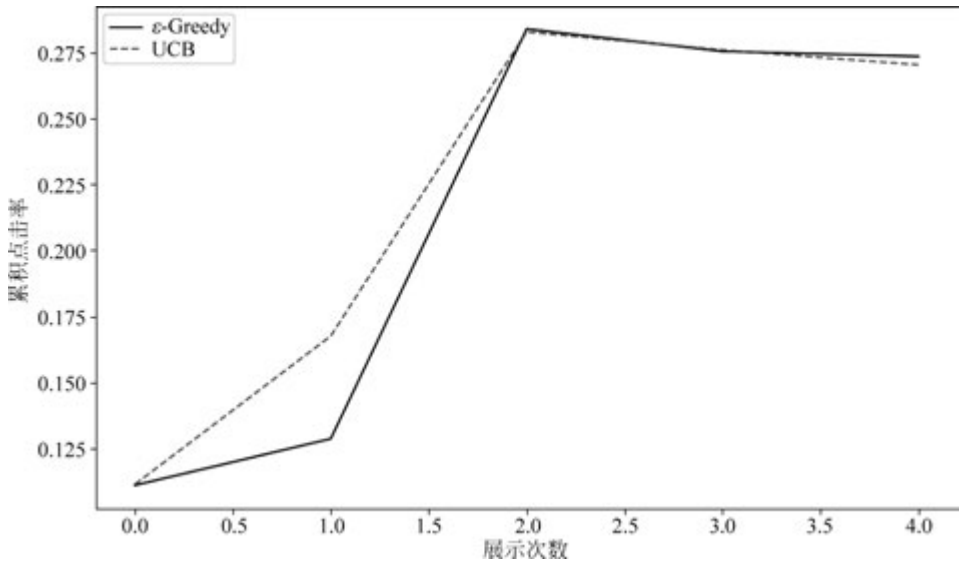

图 3.4　ε-Greedy 与 UCB 算法在在线广告优化中的表现

根据图 3.4 可以得出一些结论：在初期表现方面，UCB 算法在早期阶段（0～1.5 展示次数）表现明显优于 ε-贪婪算法。这体现了 UCB 更智能的探索策略，它能够更快地识别和利用高回报的广告。在收敛速度方面，UCB 算法比 ε-贪婪算法更快地达到稳定的高点击率。这说明 UCB 在平衡探索和利用方面更为有效，能够更快地找到最优策略。长期表现，在 1.5～2.0 展示次数后，两种算法的表现趋于一致，都达到了较高的点击率水平。这表明在足够长的时间内，两种算法都能找到接近最优的策略。从稳定性来看，在 2.0 展示次数之后，两种算法的表现都略有下降，但保持相对稳定。UCB 算法的曲线略微平滑一些，可能表明它在长期运行中的稳定性稍好。从实现复杂度与效果权衡来看，ε-贪婪算法虽然实现简单，但在初期表现不如 UCB。这验证了其“探索完全随机”的缺点。UCB 算法虽然实现稍复杂，但其智能的探索策略带来了明显的早期优势。

3.3 马尔可夫决策过程基础

在上述内容中，讨论了强化学习(RL)的许多应用，从机器人到金融。在实现这些应用的任何RL算法之前，首先需要对它们进行数学建模。马尔可夫决策过程(Markov Decision Process，MDP)是用来建模这些序列决策问题的框架。MDP具有一些独特的特性，能够更容易地对这些问题进行理论分析。基于这种理论，动态规划(Dynamic Programming，DP)领域提出了MDP的解决方法。从某种意义上说，RL可以被视为一系列近似的DP方法，能够为那些无法用精确DP方法解决的非常复杂的问题找到良好的(但不一定是最优的)解决方案。

在本节中，将逐步构建一个MDP，解释其特性，并为未来学习RL算法奠定数学基础。在MDP中，智能体采取的行动具有长期影响，这也是它与之前介绍的多臂老虎机问题的主要区别。

3.3.1 马尔可夫链

马尔可夫链是一种数学模型，用于描述一个系统在不同状态之间的转移。在强化学习中，马尔可夫链被用来建模智能体与环境的交互过程。在这种模型中，系统的下一个状态仅依赖于当前状态，而与之前的状态序列无关。该性质极大地简化了对系统行为的分析和预测。

马尔可夫性质定义为：给定当前状态，未来状态的转移概率只依赖于当前状态，而与之前的状态序列无关。形式化地，若系统在时间 t 的状态为 s_t，则马尔可夫性质表示为

$$P(s_{t+1} \mid s_t, s_{t-1}, \cdots, s_0) = P(s_{t+1} \mid s_t)$$

这一性质的重要性在于它简化了系统的建模，使得许多强化学习算法能够高效地应用和求解。

状态转移概率矩阵 $\boldsymbol{P}$ 描述了系统从一个状态转移到另一个状态的概率。对于任意两个状态 s 和 s'，状态转移概率 $\boldsymbol{P}(s' \mid s)$ 表示系统从状态 s 转移到状态 s' 的概率。状态转移概率矩阵可以表示为

$$\boldsymbol{P} = \begin{pmatrix} P(s_1 \mid s_1) & P(s_2 \mid s_1) & \cdots & P(s_n \mid s_1) \\ P(s_1 \mid s_2) & P(s_2 \mid s_2) & \cdots & P(s_n \mid s_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(s_1 \mid s_n) & P(s_2 \mid s_n) & \cdots & P(s_n \mid s_n) \end{pmatrix}$$

如图3.5所示，我们使用网格世界中的机器人来解释马尔可夫链。机器人在一个4×4的网格中移动。图中机器人位于坐标为(1,2)的格子内，并且它可以向4个方向移动：上(p_u)、下(p_d)、左(p_l)、右(p_r)。各方向的移动概率分别为 $p_u=0.2$、$p_d=0.3$、$p_l=0.25$、$p_r=0.25$。马尔可夫链的解释状态在这个网格世界中，每个格子代表一个状态。对于4×4的网格，总共有16个状态。可以用(i,j)来表示状态，其中，i 和 j 分别表示机器人的行和列的位置。动作机器人在每个状态下都有4个可能的动作：向上移动、向下移动、向左移动和向右移动。每个动作都有一个对应的概率。状态转移概率马尔可夫链的一个重要特性是状态转移概率，它描述了系统从一个状态转移到另一个状态的概率。在这个例子

中，状态转移概率由各方向的移动概率决定。例如，机器人在状态(1,2)时，向上移动到状态(2,2)的概率为0.2，向下移动到状态(0,2)的概率为0.3，向左移动到状态(1,1)的概率为0.25，向右移动到状态(1,3)的概率为0.25。

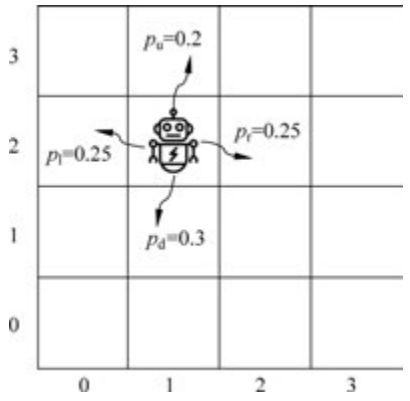

图3.5 马尔可夫链示意

这些概率可以表示为状态转移概率矩阵 $\boldsymbol{P}$ 的一部分，如下。

$$\boldsymbol{P}=\begin{pmatrix} \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & P((2,2)\mid(1,2)) & \cdots \\ \cdots & P((0,2)\mid(1,2)) & \cdots & P((1,3)\mid(1,2)) \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

其中，$P((i',j')\mid(i,j))$表示从状态(i,j)转移到状态(i',j')的概率。

机器人从当前状态转移到下一个状态的概率仅依赖于当前状态和所选动作，而与之前的状态序列无关。例如，机器人在(1,2)状态选择向上移动，转移到(2,2)的概率仅由当前状态(1,2)和动作向上决定，而与机器人之前是否从其他状态到达(1,2)无关。

在强化学习中，可以使用这个马尔可夫链模型来训练智能体。在每个状态下，智能体根据策略选择一个动作，然后根据状态转移概率和即时奖励更新其策略和价值函数。通过这种方法，智能体能够在复杂的环境中学习最优策略，以实现最大化累积奖励的目标。

下面介绍马尔可夫链中状态的分类。

(1) 可达状态和通信状态。

在一个马尔可夫链中，如果环境可以在一定数量的步骤后以正概率从状态 i 转移到状态 j，我们说 j 从 i 可达。如果 i 也可以从 j 到达，那么这些状态被称为通信状态。如果马尔可夫链中的所有状态都相互通信，我们称这个马尔可夫链是不可约的。这意味着从任何状态都可以到达其他任何状态。

(2) 吸收状态。

如果一个状态 s 的唯一可能的转移是到自身，即 $P(S_{t+1}=s \mid S_t=s)=1$，那么它就是一个吸收状态。一个吸收状态可以被视为终止状态，系统一旦进入该状态便无法离开。例如，如果机器人在前面的示例中撞到墙壁后无法再移动，这将是一个吸收状态的例子。图3.6展示了一个具有吸收状态的马尔可夫链，其中，Crashed状态为吸收状态。

(3) 瞬态状态和常返状态。

如果存在另一个状态 j 可以从 i 到达，但反之则不成立，那么状态 i 被称为瞬态状态。

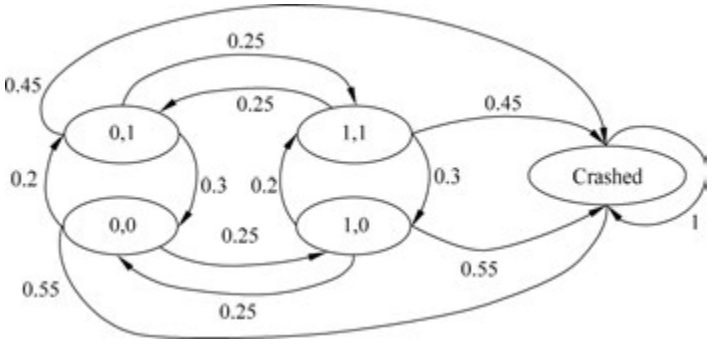

图 3.6　马尔可夫吸收状态演示

给定足够的时间，系统最终会离开瞬态状态并且永远不会返回。相反，一个不是瞬态的状态被称为常返状态。例如，如图 3.7 所示，考虑一个网格世界，有光明面和黑暗面。光明面的所有状态都是瞬态状态，因为机器人一旦进入黑暗面就无法返回。黑暗面的状态是常返状态。

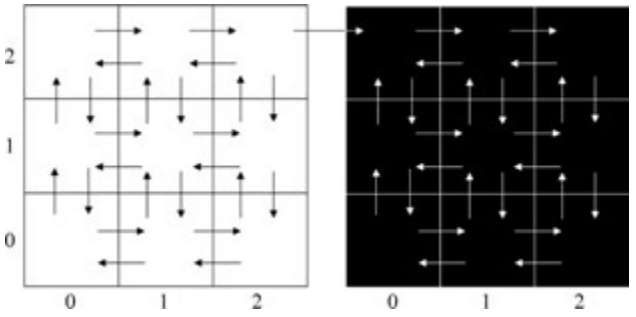

图 3.7　瞬态状态和常返状态示意图

(4) 周期性状态和非周期性状态。

如果从状态 s 离开的所有路径都在 $k>1$ 步的某个倍数后返回，称状态 s 是周期性的。若 $k=1$，则常返状态被称为非周期性的。图 3.8 展示了一个具有周期性状态的马尔可夫链，其中所有状态的周期为 $k=4$。

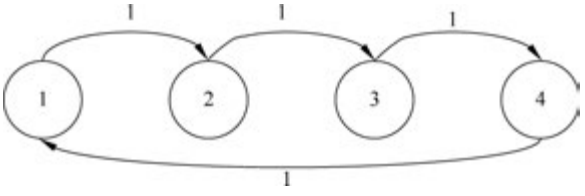

图 3.8　具有周期性状态的马尔可夫链

一个马尔可夫链被称为遍历的，如果所有状态都具有属性：相互通信(不可约)，是常返的，是非周期性的。对于遍历马尔可夫链，可以计算出一个稳态概率分布，告诉我们系统在初始化很长时间后会以什么概率处于哪个状态。

(5) 瞬态行为和稳态行为。

马尔可夫链的行为可以分为瞬态行为和稳态行为。瞬态行为描述了系统随时间变化的短期行为，而稳态行为描述了系统达到平衡状态后的长期行为。我们可以数学地计算马

尔可夫链随时间的行为。首先，需要知道系统的初始概率分布。然后，定义转移概率矩阵，其条目给出了从一个时间步到下一个时间步所有状态对之间的转移概率。更正式地，该矩阵第 i 行和第 j 列的条目给出 $P_{ij}=P(S_{t+1}=j \mid S_t=i)$。要计算系统在 n 步后处于状态 j 的概率，使用以下公式。

$$\pi^{(n)}=\pi^{(0)}P^n$$

其中，$\pi^{(0)}$ 是初始概率分布，P^n 是转移概率矩阵的 n 次幂。注意，P_{ij}^n 给出从状态 i 开始后 n 步处于状态 j 的概率。

3.3.2 马尔可夫奖励过程

在我们的机器人示例中，到目前为止，还没有真正确定任何“好”或“坏”的情况或状态。然而，在任何系统中，都有一些状态是期望处于的，而另一些则是不太理想的。在本节中，将对状态和转移附加奖励，这使我们能够构建一个马尔可夫奖励过程（MRP）。随后，将评估每个状态的“价值”。

马尔可夫奖励过程（Markov Reward Process，MRP）是马尔可夫决策过程（MDP）的简化版本，用于建模和分析具有奖励结构的随机过程。在 MRP 中，不仅关心状态的转移，还关心每次状态转移所带来的奖励。MR 的引入提供了评估和优化系统性能的工具，是理解更复杂的 MDP 和强化学习（RL）的基础。

一个马尔可夫奖励过程可以由一个四元组〈S，$\boldsymbol{P}$，R，γ〉表示，其中：

（1）S 是状态的集合。

（2）$\boldsymbol{P}$ 是状态转移概率矩阵，其中，$P(s' \mid s)$ 表示从状态 s 转移到状态 s 的概率。

（3）R 是奖励函数，其中，$R(s)$ 表示在状态 s 时获得的即时奖励。

（4）γ 是折扣因子，取值范围为 $[0,1]$，用于折扣未来的奖励，使得远期奖励的影响逐渐减小。

在 MRP 中，引入价值函数 $V(s)$ 来表示从状态 s 开始的预期累计折扣奖励。形式上，价值函数定义为

$$V(s)=E[G_t \mid S_t=s]=E\left[\sum_{k=0}^{\infty}\gamma^k R(S_{t+k}) \mid S_t=s\right]$$

其中，G_t 是从时间步 t 开始的折扣汇报。

价值函数 $V(S)$ 满足贝尔曼方程，该方程描述了当前状态的价值与其后续状态的价值之间的递归关系。贝尔曼方程表示为

$$V(S)=R(s)+\gamma\sum_{s'\in S}P(s' \mid s)V(s')$$

这一递归关系提供了计算 MRP 中状态值的基础。

【编程示例 3.2】 MRP 示例。

假设一个机器人在一个 3×3 的网格世界中移动。每个格子代表一个状态，机器人可以向 4 个方向移动：上、下、左、右。机器人从任意状态开始移动，每次移动会获得 +1 的奖励，但如果机器人撞到墙壁，它会进入一个崩溃状态，并终止回合。在崩溃状态，机器人无法移动，且奖励为 0。

(1) 定义转移概率矩阵 $\boldsymbol{P}$。

首先,定义状态转移概率矩阵 $\boldsymbol{P}$,它描述了从一个状态转移到其他状态的概率。

```
import numpy as np
#定义状态数量和转移概率矩阵
m2 = 9                                                    #3×3 网格,有 9 个状态
P = np.zeros((m2 + 1, m2 + 1))
def get_P(grid_size, pu, pd, pl, pr):
    P = np.zeros((grid_size * grid_size, grid_size * grid_size))
    for i in range(grid_size):
        for j in range(grid_size):
            current_state = i * grid_size + j
            if i > 0:
                P[current_state, current_state - grid_size] = pu  #向上
            if i < grid_size - 1:
                P[current_state, current_state + grid_size] = pd  #向下
            if j > 0:
                P[current_state, current_state - 1] = pl          #向左
            if j < grid_size - 1:
                P[current_state, current_state + 1] = pr          #向右
    return P
#获取 3×3 网格的转移概率矩阵
P[:m2, :m2] = get_P(3, 0.2, 0.3, 0.25, 0.25)
#为崩溃状态分配自转移概率
for i in range(m2):
    P[i, m2] = P[i, i]
    P[i, i] = 0
P[m2, m2] = 1                                             #崩溃状态的自转移概率
```

(2) 定义奖励函数。

奖励函数 RRR 用于定义每个状态的即时奖励。在示例中,所有非崩溃状态的奖励为 1,而崩溃状态的奖励为 0。

```
#定义奖励向量
R = np.ones(m2 + 1)
R[-1] = 0                                                 #崩溃状态的奖励为 0
```

(3) 计算状态值。

使用贝尔曼方程计算每个状态的值。贝尔曼方程描述了当前状态的价值与其后续状态的价值之间的递归关系。设置折扣因子 γ 接近 1,以确保未来的奖励被充分考虑。

```
gamma = 0.9999
#计算状态值
inv = np.linalg.inv(np.eye(m2 + 1) - gamma * P)
v = np.matmul(inv, R)
#输出结果
print(np.round(v, 2))
```

输出结果为

```
[2.99 3.82 2.99 3.44 4.42 3.44 2.47 3.12 2.47 0. ]
```

这些值表示每个状态的预期折扣回报。

分析结果：初始状态为(1,1)的状态值最高，说明从这个位置开始，机器人在崩溃之前可以获得更多的奖励。状态值低的状态，如(2,0)，表明从这些位置开始，机器人更容易撞到墙壁并进入崩溃状态，因此获得的总奖励较少。

(4) 迭代估计状态值。

还可以使用迭代方法估计状态值。通过反复更新状态值，直到收敛到一个稳定值，可以得到一个近似的状态值。

```
def estimate_state_values(P, m2, threshold):
    v = np.zeros(m2 + 1)
    max_change = threshold
    terminal_state = m2
    while max_change >= threshold:
        max_change = 0
        for s in range(m2 + 1):
            v_new = 0
            for s_next in range(m2 + 1):
                r = 1 * (s_next != terminal_state)
                v_new += P[s, s_next] * (r + gamma * v[s_next])
            max_change = max(max_change, np.abs(v[s] - v_new))
            v[s] = v_new
    return np.round(v, 2)
#运行迭代估计算法
estimated_values = estimate_state_values(P, m2, 0.01)
print(estimated_values)
```

马尔可夫奖励过程提供了一个框架，用于评估和优化系统在不同状态下的性能。通过引入奖励和折扣因子，可以更准确地建模现实世界中的决策问题。在 MRP 的基础上，可以进一步扩展到马尔可夫决策过程(MDP)，并引入策略优化的概念，这为复杂的强化学习算法奠定了基础。通过这些方法，可以有效地估计状态值，并为更复杂的强化学习算法奠定基础。接下来，将在这个图景中加入最后一个主要部分：行动。

3.3.3 马尔可夫决策过程

马尔可夫决策过程(MDP)是马尔可夫奖励过程(MRP)的扩展，加入了决策元素。MDP 可以通过一个五元组$\langle S, A, \boldsymbol{P}, R, \gamma \rangle$来表征，其中：

(1) S 是状态集合。

(2) A 是动作集合。

(3) $\boldsymbol{P}$ 是状态转移概率矩阵，$P(s' \mid s, a)$表示在状态 s 执行动作 a 后转移到状态 s'的概率。

(4) R 是奖励函数，$R(s, a)$表示在状态 s 执行动作 a 所获得的奖励。

(5) γ 是折扣因子，取值范围为 $[0,1]$，用于折扣未来的奖励。

在 MDP 中，智能体的目标是找到一个策略，使得预期累积奖励最大化。策略 π 定义为在给定状态下选择动作的概率分布，即 $\pi(a \mid s) = P[A_t = a \mid S_t = s]$。下面，在 MDP 中，可以定义状态值函数 $V_\pi(s)$ 和动作值函数 $Q_\pi(s, a)$。

(1) 状态值函数。

状态值函数 $V_\pi(s)$定义为从状态 s 开始,遵循策略 π 的预期累积折扣奖励:

$$V_\pi(s)=E_\pi\left[\sum_{k=0}^{\infty}y^kR_{t+k+1}\mid S_t=s\right]$$

贝尔曼方程描述了状态值函数的递归关系:

$$V_\pi(s)=\sum_a\pi(a\mid s)\sum_{s',r}P(s',r\mid s,a)\left[r+\gamma V_\pi(s')\right]$$

(2) 动作值函数。

动作值函数 $Q_\pi(s,a)$定义为在状态 s 执行动作 a,然后遵循策略 π 的预期累积折扣奖励:

$$Q_\pi(s,a)=E_\pi\left[\sum_{k=0}^{\infty}\gamma^kR_{t+k+1}\mid S_t=s,A_t=a\right]$$

它也可以通过贝尔曼方程描述:

$$Q_\pi(s,a)=\sum_{s',r}P(s',r\mid s,a)\left[r+\gamma V_\pi(s')\right]$$

接下来,编写一个示例程序。程序旨在通过马尔可夫决策过程(MDP)来模拟和优化一个机器人在 3×3 网格世界中的行动。机器人可以在每个时间步选择向上、向下、向左或向右移动。移动的成功率为 70%,其余 30%的概率下,机器人会朝其他三个方向之一移动。在边界状态下,如果机器人尝试向网格外移动,它会进入一个崩溃状态,并终止回合。在崩溃状态中,机器人无法移动,且奖励为 0。在其他状态下,每次移动奖励为 1。定义了一种策略,使机器人在每个状态下总是选择向右移动。程序的目标是计算在这种策略下,各状态的状态值函数 $V(s)$和动作值函数 $Q(s,a)$,从而评估该策略的有效性。

【编程示例 3.3】 MDP 示例。

```
import numpy as np
#定义网格大小和状态数量
grid_size = 3
m2 = grid_size * grid_size                          #3×3 网格,有 9 个状态
P = np.zeros((m2 + 1, m2 + 1, 4))                   #包含 4 个动作的转移概率矩阵
#定义动作集合
UP, DOWN, LEFT, RIGHT = 0, 1, 2, 3
#定义函数来生成转移概率矩阵
def get_P_with_actions(grid_size, pu, pd, pl, pr):
    P = np.zeros((grid_size * grid_size + 1, grid_size * grid_size + 1, 4))
    for i in range(grid_size):
        for j in range(grid_size):
            current_state = i * grid_size + j
            if i > 0:
                P[current_state, current_state - grid_size, UP] = pu     #向上
            if i < grid_size - 1:
                P[current_state, current_state + grid_size, DOWN] = pd   #向下
            if j > 0:
                P[current_state, current_state - 1, LEFT] = pl           #向左
            if j < grid_size - 1:
                P[current_state, current_state + 1, RIGHT] = pr          #向右
            for action in [UP, DOWN, LEFT, RIGHT]:
```

```
                if i == 0 and action == UP:
                    P[current_state, grid_size * grid_size, action] = pu
                if i == grid_size - 1 and action == DOWN:
                    P[current_state, grid_size * grid_size, action] = pd
                if j == 0 and action == LEFT:
                    P[current_state, grid_size * grid_size, action] = pl
                if j == grid_size - 1 and action == RIGHT:
                    P[current_state, grid_size * grid_size, action] = pr
    return P
#获取 3×3 网格的转移概率矩阵
P = get_P_with_actions(3, 0.7, 0.1, 0.1, 0.1)
#定义奖励向量
R = np.ones((m2 + 1, 4))
R[m2, :] = 0                          #崩溃状态的奖励为 0
#定义策略
pi = np.zeros((m2, 4))
pi[:, RIGHT] = 1.0                    #在每个状态下总是选择向右
#设置折扣因子
gamma = 0.9999
#计算状态值函数
def compute_state_value_function(P, R, pi, gamma, m2):
    V = np.zeros(m2 + 1)
    threshold = 1e-6
    max_change = threshold
    while max_change >= threshold:
        max_change = 0
        V_new = np.zeros(m2 + 1)
        for s in range(m2):
            V_new[s] = sum(pi[s, a] * sum(P[s, s_next, a] * (R[s, a] + gamma * V[s_next])
for s_next in range(m2 + 1)) for a in range(4))
        max_change = max(max_change, np.max(np.abs(V_new - V)))
        V = V_new
    return V
#计算动作值函数
def compute_action_value_function(P, R, V, gamma, m2):
    Q = np.zeros((m2, 4))
    for s in range(m2):
        for a in range(4):
            Q[s, a] = sum(P[s, s_next, a] * (R[s, a] + gamma * V[s_next]) for s_next in
range(m2 + 1))
    return Q
#计算状态值和动作值函数
V = compute_state_value_function(P, R, pi, gamma, m2)
Q = compute_action_value_function(P, R, V, gamma, m2)
#输出结果
print("状态值函数:")
print(np.round(V, 2))
print("动作值函数:")
print(np.round(Q, 2))
```

代码解析：代码首先初始化了网格大小 grid_size 和状态数量 m2(3×3 网格有 9 个状

态),并初始化了转移概率矩阵 $\boldsymbol{P}$ 和奖励函数 R。动作集合包括 UP、DOWN、LEFT 和 RIGHT,分别表示向上、向下、向左和向右的移动方向。这些初始化步骤为后续的转移概率计算和奖励分配提供了基础。

转移概率矩阵由函数 get_P_with_actions 生成。该函数根据给定的动作概率(pu,pd,pl,pr)填充矩阵 $\boldsymbol{P}$,表示从当前状态执行不同动作后转移到其他状态的概率。函数遍历每个网格状态,计算在执行不同动作后的转移概率,同时处理边界条件,即当机器人尝试向网格外移动时会进入崩溃状态。崩溃状态被表示为网格状态外的一个特定状态,其转移概率为 1。

奖励函数 R 被定义为一个矩阵,其中,崩溃状态的奖励为 0,其他状态的即时奖励为 1。策略 pi 定义为一个矩阵,表示在每个状态下选择各个动作的概率。在本例中,机器人在每个状态下总是选择向右移动,即 pi[:,RIGHT]=1.0。这种策略定义方式简单明了,为后续的状态值和动作值计算提供了明确的决策依据。

状态值函数的计算由函数 compute_state_value_function 实现。该函数使用迭代方法,根据贝尔曼方程更新状态值 $V(s)$。每次迭代中,函数根据当前策略和转移概率矩阵计算状态值,直到所有状态值的变化量小于设定的阈值 threshold。这种迭代方法确保了状态值函数的收敛,使得每个状态的预期累积折扣奖励得以准确计算。

动作值函数的计算由函数 compute_action_value_function 完成。该函数利用状态值函数 $V(s)$计算在每个状态下执行特定动作后的预期累积折扣奖励 $Q(s,a)$。通过遍历所有状态和动作,函数计算在当前策略下执行每个动作的预期回报,并输出动作值函数矩阵。最后,程序输出计算得到的状态值函数 $V(s)$和动作值函数 $Q(s,a)$,为进一步评估和优化策略提供了数据支持。

通过这种方法,可以有效地评估和优化 MDP 中的策略,从而找到最优策略,最大化预期累积奖励。在 3.4 节中,将详细讨论如何改进和找到最优策略,并解决更复杂的例子。

3.4 动态规划方法

动态规划(Dynamic Programming,DP)是解决马尔可夫决策过程(MDP)的一类重要方法。它基于将复杂问题分解为子问题的原理,并利用子问题的解来构建更大问题的解。在强化学习中,DP 方法假设完全知道环境模型,即状态转移概率和奖励函数。虽然这在很多实际问题中是不现实的,但 DP 方法为理解和开发更复杂的强化学习算法提供了基础。

3.4.1 策略评估

策略评估,也称为预测问题,是指给定一个策略 π,计算其状态值函数 V_π 的过程。

迭代策略评估算法基于贝尔曼期望方程,通过反复应用该方程来逼近真实的状态值函数。算法步骤如下。

(1) 初始化 $V(s)$为任意值,通常为 0。

(2) 重复直到收敛。

对于每个状态 s:

$$V(s) \leftarrow \sum_{a} \pi(a \mid s) \sum_{s',r} P(s',r \mid s,a)\,[r + \gamma V(s')]$$

这个更新规则直接来自贝尔曼期望方程。下面给出相应的代码片段。

```
def policy_evaluation(policy, env, discount_factor = 1.0, theta = 1e - 9):
    V = np.zeros(env.nS)
    while True:
        delta = 0
        for s in range(env.nS):
            v = 0
            for a, action_prob in enumerate(policy[s]):
                for prob, next_state, reward, done in env.P[s][a]:
                    v += action_prob * prob * (reward + discount_factor * V[next_state])
            delta = max(delta, np.abs(v - V[s]))
            V[s] = v
        if delta < theta:
            break
    return V
```

迭代策略评估算法的收敛性可以通过收缩映射原理来证明。每次迭代都是一个收缩映射，因为折扣因子 $\gamma<1$。根据 Banach 不动点定理，这保证了算法最终会收敛到唯一的解，即真实的状态值函数 V_π。收敛速度受折扣因子 γ 的影响。较小的 γ 值会加快收敛，但可能会导致短视的评估。

3.4.2 策略改进

策略改进是在给定当前策略 π 的值函数 V_π 的基础上，找到一个更好的策略 π'的过程。策略改进定理指出，如果改变策略使得

$$\sum_{s',r} P(s',r \mid s,a)\,[r + \gamma V_\pi(s')] \geqslant V_\pi(s)$$

对于所有状态 s 成立，那么新策略 π'一定不会比 π 更差，即 $V_{\pi'}(s) \geqslant V_\pi(s)$对所有 s 成立。

基于策略改进定理，可以定义贪婪策略改进：

$$\pi'(s) = \arg\max_{a} \sum_{s',r} P(s',r \mid s,a)\,[r + \gamma V_\pi(s')]$$

这个新策略 π'在每个状态都选择能够最大化期望回报的动作。下面给出相应的算法实现。

```
def policy_improvement(env, V, discount_factor = 1.0):
    policy = np.zeros([env.nS, env.nA])
    for s in range(env.nS):
        q_values = np.zeros(env.nA)
        for a in range(env.nA):
            for prob, next_state, reward, done in env.P[s][a]:
                q_values[a] += prob * (reward + discount_factor * V[next_state])
        best_a = np.argmax(q_values)
        policy[s, best_a] = 1.0
    return policy
```

上述代码实现了策略改进算法，用于在给定的值函数 V 和环境模型 env 下优化策略。它遍历每个状态 s，计算在当前状态下执行每个可能动作 a 的期望回报 q_values，然后选择

期望回报最大的动作作为最优动作，将其设置为新的策略。最终返回优化后的策略。

3.4.3 值迭代

值迭代结合了策略评估和策略改进的思想，直接操作值函数来找到最优策略。

值迭代算法基于贝尔曼最优方程：

(1) 初始化$V(s)$为任意值。

(2) 重复直到收敛。

对于每个状态s：

$$V(s) \leftarrow \max_a \sum_{s',r} P(s',r \mid s,a)\left[r+\gamma V(s')\right]$$

下面给出相应的代码。

```
def value_iteration(env, discount_factor = 1.0, theta = 1e - 9):
    V = np.zeros(env.nS)
    while True:
        delta = 0
        for s in range(env.nS):
            v = V[s]
            V[s] = max(sum(prob * (reward + discount_factor * V[next_state])
                           for prob, next_state, reward, done in env.P[s][a])
                       for a in range(env.nA))
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    policy = np.zeros([env.nS, env.nA])
    for s in range(env.nS):
        q_values = np.zeros(env.nA)
        for a in range(env.nA):
            q_values[a] = sum(prob * (reward + discount_factor * V[next_state])
                              for prob, next_state, reward, done in env.P[s][a])
        best_a = np.argmax(q_values)
        policy[s, best_a] = 1.0
    return policy, V
```

上述代码实现了值迭代算法，用于在给定环境中找到最优策略。首先，初始化状态值函数V为零，然后通过反复应用贝尔曼最优方程更新状态值，直到值函数的变化量δ小于设定的阈值θ，表示收敛。接着，根据最终的值函数V计算每个状态下的动作值Q，选择能够最大化Q的动作作为策略。最终返回最优策略和对应的状态值函数。

值迭代的收敛性同样可以通过收缩映射原理证明。此外，值迭代还保证收敛到最优值函数V^*。一旦有了V^*，可以直接得到最优策略π^*。

3.4.4 案例分析

下面使用动态规划解决简单的网格世界问题。为了具体说明这些方法，考虑一个4×4的网格世界：起点在左上角(0,0)，目标在右下角(3,3)，在(1,1)位置有一个障碍物，每次移动的奖励为-1(包括撞墙)，到达目标的奖励为$+10$，折扣因子$\gamma=0.9$。

将使用 OpenAI Gym 风格的环境来实现这个问题，然后应用策略迭代和值迭代算法。

OpenAI Gym 是一个用于开发和比较强化学习算法的工具包。它提供了一组标准化的环境，这些环境涵盖了从经典控制和 Atari 游戏到机器人控制和自动驾驶等各种任务。Gym 的主要目标是为研究人员和开发人员提供一个统一的接口，以便于创建、测试和比较不同的强化学习算法。Gym 环境包括一系列的组件和标准接口，有兴趣的读者可以自行查找相关资料。

【编程示例 3.4】 动态规划示例。

```
class GridWorldEnv:
    def __init__(self):
        self.nS = 16
        self.nA = 4
        self.P = self._build_transitions()
    def _build_transitions(self):
        #构建转移概率和奖励
        P = {}
        for s in range(self.nS):
            P[s] = {a: [] for a in range(self.nA)}
        #定义网格世界的转移概率和奖励
        for row in range(4):
            for col in range(4):
                s = row * 4 + col
                if s == 3 * 4 + 3:
                    P[s][0] = [(1.0, s, 0, True)]
                    P[s][1] = [(1.0, s, 0, True)]
                    P[s][2] = [(1.0, s, 0, True)]
                    P[s][3] = [(1.0, s, 0, True)]
                    continue
                if row > 0:
                    P[s][0].append((1.0, s - 4, -1, False))
                else:
                    P[s][0].append((1.0, s, -1, False))
                if row < 3:
                    P[s][1].append((1.0, s + 4, -1, False))
                else:
                    P[s][1].append((1.0, s, -1, False))
                if col > 0:
                    P[s][2].append((1.0, s - 1, -1, False))
                else:
                    P[s][2].append((1.0, s, -1, False))
                if col < 3:
                    P[s][3].append((1.0, s + 1, -1, False))
                else:
                    P[s][3].append((1.0, s, -1, False))
        P[1 * 4 + 1][0] = [(1.0, 1 * 4 + 1, -1, False)]
        P[1 * 4 + 1][1] = [(1.0, 1 * 4 + 1, -1, False)]
        P[1 * 4 + 1][2] = [(1.0, 1 * 4 + 1, -1, False)]
        P[1 * 4 + 1][3] = [(1.0, 1 * 4 + 1, -1, False)]
        return P
#策略评估函数
```

```
def policy_evaluation(policy, env, discount_factor = 1.0, theta = 1e - 9):
    V = np.zeros(env.nS)
    while True:
        delta = 0
        for s in range(env.nS):
            v = 0
            for a, action_prob in enumerate(policy[s]):
                for prob, next_state, reward, done in env.P[s][a]:
                    v += action_prob * prob * (reward + discount_factor * V[next_state])
            delta = max(delta, np.abs(v - V[s]))
            V[s] = v
        if delta < theta:
            break
    return V
#策略改进函数
def policy_improvement(env, V, discount_factor = 1.0):
    policy = np.zeros([env.nS, env.nA])
    for s in range(env.nS):
        q_values = np.zeros(env.nA)
        for a in range(env.nA):
            for prob, next_state, reward, done in env.P[s][a]:
                q_values[a] += prob * (reward + discount_factor * V[next_state])
        best_a = np.argmax(q_values)
        policy[s, best_a] = 1.0
    return policy
#实现策略迭代
def policy_iteration(env, discount_factor = 0.9):
    policy = np.ones([env.nS, env.nA]) / env.nA
    while True:
        V = policy_evaluation(policy, env, discount_factor)
        new_policy = policy_improvement(env, V, discount_factor)
        if (new_policy == policy).all():
            break
        policy = new_policy
    return policy, V
#实现值迭代
def value_iteration(env, discount_factor = 0.9):
    V = np.zeros(env.nS)
    while True:
        delta = 0
        for s in range(env.nS):
            v = V[s]
            V[s] = max(sum(prob * (reward + discount_factor * V[next_state])
                           for prob, next_state, reward, done in env.P[s][a])
                       for a in range(env.nA))
            delta = max(delta, abs(v - V[s]))
        if delta < 1e - 9:
            break
    policy = np.zeros([env.nS, env.nA])
    for s in range(env.nS):
        q_values = np.zeros(env.nA)
```

```
            for a in range(env.nA):
                q_values[a] = sum(prob * (reward + discount_factor * V[next_state])
                                  for prob, next_state, reward, done in env.P[s][a])
            best_a = np.argmax(q_values)
            policy[s, best_a] = 1.0
        return policy, V
    #创建环境
    env = GridWorldEnv()
    #运行策略迭代
    pi_policy, pi_V = policy_iteration(env)
    #运行值迭代
    vi_policy, vi_V = value_iteration(env)
```

这段代码定义了一个4×4网格世界（GridWorldEnv类），其中每个状态都有4个可能的动作（上、下、左、右），并为每个状态和动作定义了转移概率和奖励函数。代码中包含策略评估（policy_evaluation）、策略改进（policy_improvement）、策略迭代（policy_iteration）和值迭代（value_iteration）函数，用于计算和优化策略，使智能体在网格世界中行动时最大化其累积奖励。策略评估函数通过迭代方法计算每个状态的值函数，策略改进函数根据当前值函数更新策略，而策略迭代和值迭代函数则结合这些方法反复迭代，最终输出最优策略和对应的值函数。最后，代码创建环境并运行策略迭代和值迭代，计算出最优策略和状态值函数。

为了更加直观地比较策略迭代和值迭代的结果，在本书附带的代码中增加了可视化的结果。读者可以自行运行程序来观察结果的不同。通常，值迭代收敛更快，因为它在每次迭代中都执行部分策略改进，而两种方法应该产生相同的最优策略；在计算复杂度方面，策略迭代在每次迭代中都需要完整的策略评估，这在大型状态空间中可能很昂贵，而值迭代通过将策略评估限制为一次扫描来避免这个问题；此外，在中间结果的解释上，策略迭代在每次迭代后都产生一个完整的策略，而值迭代直到最后才产生策略。

编程示例3.4展示了如何将动态规划方法应用于实际问题。虽然这是一个简单的例子，但相同的原理可以扩展到更复杂的MDP问题。在实际应用中，动态规划方法的主要限制是它们需要完整的环境模型和相对较小的状态空间。对于大型或未知环境，需要转向其他强化学习方法，如蒙特卡洛方法或时序差分学习。然而，理解动态规划方法为这些更高级的技术提供了重要的理论基础。

小结

本章介绍了强化学习的基本概念和原理，从多臂赌博机问题开始，逐步深入马尔可夫决策过程（MDP）的理论基础。首先讨论了强化学习与监督学习、无监督学习的区别，强调了强化学习在序列决策问题中的独特优势。通过多臂赌博机问题，探讨了探索与利用的权衡，并介绍了ε-贪婪和上置信界（UCB）等算法来解决这一问题。

接着，深入研究了马尔可夫链、马尔可夫奖励过程和马尔可夫决策过程的概念。这些理论为理解复杂的强化学习问题奠定了基础。讨论了状态、动作、奖励、状态转移概率等关键概念，并引入了值函数和策略的概念，为后续的算法设计做好了铺垫。

最后，介绍了动态规划方法，包括策略评估、策略改进、策略迭代和值迭代等算法。这些方法虽然在实际应用中受限于对环境完全了解的假设，但它们为理解更复杂的强化学习算法提供了重要的理论基础。通过一个简单的网格世界问题，展示了如何将这些方法应用于实际问题，并比较了不同方法的优缺点。

通过学习这些内容，读者不仅掌握了强化学习的基本理论和方法，还为未来进一步深入学习更复杂的强化学习算法打下了坚实的基础。理解这些基础概念和算法，有助于更好地理解蒙特卡洛方法、时序差分学习以及深度强化学习等高级主题，从而在面对现实世界中的复杂问题时，能够设计出更有效的解决方案。

习题

(1) 解释强化学习中的“探索-利用”权衡，并讨论它在实际应用中的重要性。

(2) 比较马尔可夫链、马尔可夫奖励过程和马尔可夫决策过程的异同。

(3) 解释贝尔曼方程在强化学习中的作用，并给出状态值函数的贝尔曼方程。

(4) 讨论动态规划方法在解决强化学习问题时的优势和局限性。

(5) 解释策略迭代和值迭代算法的主要区别，并分析它们各自的适用场景。

(6) 实现一个多臂赌博机问题的仿真环境，并比较 ε-贪婪算法和 UCB 算法在不同参数设置下的表现。

(7) 设计并实现一个简单的网格世界环境(类似于 3.4.4 节的例子)，使用策略迭代算法求解最优策略。可视化最优策略和值函数。

(8) 基于题目(2)的环境，实现值迭代算法，并与策略迭代的结果进行比较。分析两种方法在收敛速度和计算效率上的差异。