

绪论

在正式介绍具体的强化学习（reinforcement learning, RL）算法之前，本章将从宏观角度讨论一下强化学习的相关概念以及应用等，帮助读者更好地“观其大略”。尤其是对于想用利强化学做一些交叉研究的读者来说，更应该先通过本章了解强化学习是什么、大概能做什么、能够实现什么样的效果等，而不是直接从一个个具体的算法开始学习。

强化学习发展至今，虽然算法已经有成百上千种样式，但实际上从大类来看要掌握的核心算法并不多，大多数算法都只是在核心算法的基础上做了一些较小的改进。举个例子，如图 1 所示，我们知道水加上咖啡豆通过一定方法就能调制成咖啡，水加上糖块就能变成糖水，它们虽然看起来形式不同，但本质上都是水，只是有不同的口味而已。

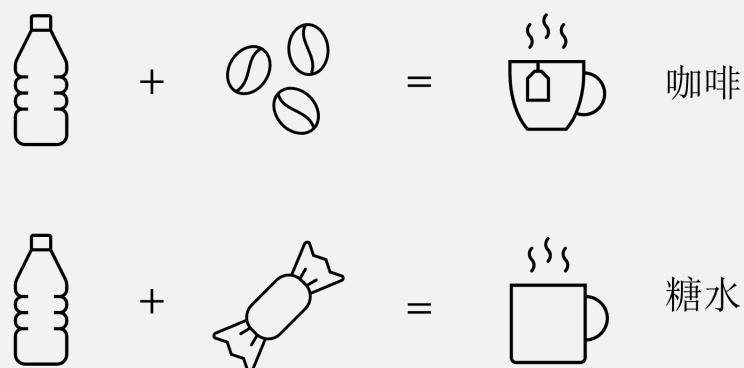


图 1: 咖啡与糖水的示例

为什么要学习强化学习？

我们先讨论一下为什么要学习强化学习，以及强化学习对于我们的意义。可能大部分读者都是通过了解人工智能才了解到强化学习的，但实际上早在我们认识人工智能之前可能就已经不知不觉地接触到了强化学习。

笔者想起了初中生物课本中关于蚯蚓的一个实验，其内容大致是这样的：如图 2 所示，将蚯蚓放在一个盒子中，盒子中间有一个分叉路口，路的尽头分别放有食物和电极，让蚯蚓自己爬行到其中一条路的尽头，在放有食物的路的尽头蚯蚓会品尝到食物的美味，而在放有电极的路的尽头则会受到轻微的电击。

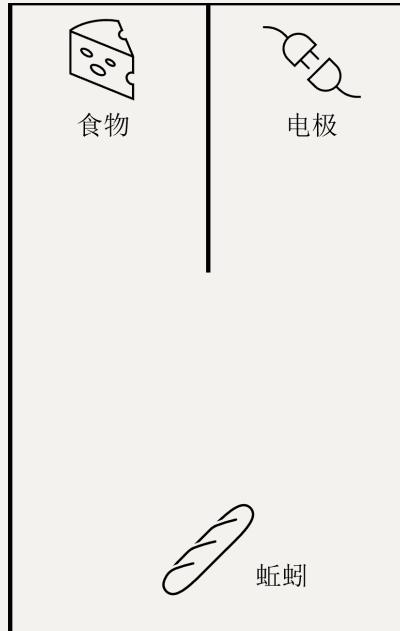


图 2: 蚯蚓实验

该实验的目的是让蚯蚓能一直朝着有食物的路爬行，但由于蚯蚓没有真正的眼睛，因此一开始蚯蚓可能会一直朝着有电极的路爬行并且遭到电击。每次蚯蚓遭到电击或者吃到食物之后实验人员会将其放回原处，经过多次实验，蚯蚓会逐渐学会朝着有食物的路爬行，而不是朝着有电极的路爬行。

在这个过程中，蚯蚓在不断地尝试和试错中学习到了正确的策略，虽然初中生物课本中这个实验的目的是为了说明蚯蚓的运动是由外界刺激所驱动的，而不是蚯蚓自身的意志所驱动的，但在今天，从人工智能的角度来看，这其实带有较为鲜明的强化学习的“味道”，即试错学习（trial and error learning）。

试错学习一开始是和行为心理学等工作联系在一起的，主要包括以下几个关键部分：

- 尝试：采取一系列动作或行为来尝试解决问题或实现目标。
- 错误：在尝试的过程中可能会出现错误，这些错误可能是环境的不确定性导致的，也可能是自身的不当行为导致的。
- 结果：每次尝试的后果，无论是积极的还是消极的，都会对下一次尝试产生影响。
- 学习：通过不断地尝试并出现错误，自身会逐渐积累经验，了解哪些动作或行为会产生有利的结果，从而在下一次尝试中做出更加明智的选择。

试错学习在我们的日常生活中屡见不鲜，并且通常与其他形式的学习形成对比，例如经典条件反射（巴甫洛夫条件反射）和观察学习（通过观察他人来学习）。注意，试错学习虽然是强化学习中最鲜明的要素之一，但并不是强化学习的全部，强化学习还包含其它的学习形式例如观察学习（对应模仿学习、离线强化学习等技术）。

此外，在学习过程中个人做出的每一次尝试都是是一次决策（**decision**），每一次决策都会带来相应的后果，这个后果可能是好的，也可能是坏的，也可能是即时的，比如我们吃到棉花糖就能立刻感受到它的甜度，也可能是延时的，比如寒窗苦读十年之后，方得一日踏阅长安花。

我们把好的结果称为奖励（**reward**），坏的结果称为惩罚（**punishment**）或者负的奖励。最终通过一次次的决策来实现目标，这个目标通常是以最大化累积的奖励来呈现的，这个过程就是序列决策（**sequential decision making**）过程，而强化学习就是解决序列决策问题的有效方法之一，即本书的主题。换句话说，对于任意问题，只要能够建模成序列决策问题或者带有鲜明的试错学习特征，就可以使用强化学习来解决，并且这是截至目前最为高效的方法之一，这就是要学习强化学习的原因。

强化学习的应用

从 1.1 中我们了解了强化学习大概是用来做什么的，那么它能实现什么样的效果呢？本节我们就来看看强化学习的一些例子和实际应用。强化学习的应用场景非常广泛，其中最为典型的场景之一就是游戏，以 AlphaGo 为代表的围棋 AI 就是强化学习的代表作之一，也是其为人们广泛熟知的得意之作。除了部分棋类游戏，以 AlphaStar 为代表的《星际争霸》AI、以 AlphaZero 为代表的通用游戏 AI，以及近年以 OpenAI Five 为代表的 Dota 2 AI，这些都是强化学习在游戏领域的典型应用。

除了游戏领域之外，强化学习在机器人领域中也有所应用。举个例子，图 3 演示了 NICO 机器人学习抓取任务的过程。该任务的目标是将桌面上的物体抓取到指定的位置，机器人通过每次输出相应关节的参数来活动手臂，然后通过摄像头观测当前的状态，最后通过人为设置的奖励（例如接近目标就给一个奖励）来学习到正确的抓取策略。

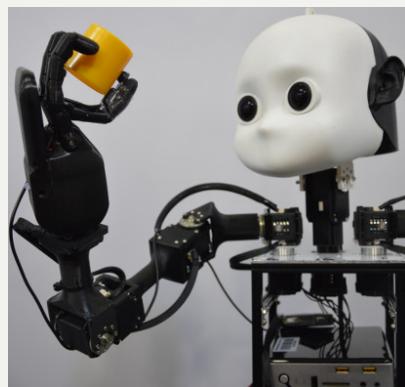


图 3: NICO 机器人学习抓取任务

不同于游戏领域，在机器人中实现强化学习的成本往往较高，一方面观测环境的状态需要大量的传感器，另一方面则是试错学习带来的实验成本，在训练过程中如果机器人决策稍有失误就有可能导致设备损坏，因此在实际应用中往往需要结合其他的方法来辅助强化学习进行决策。其中最典型的方法之一就是建立一个仿真环境，通过仿真环境来模拟真实环境，这样就可以大大降低实验成本。

如图 4 所示，该仿真环境模拟了机器人抓取任务的真实环境，通过仿真环境免去大量视觉传感器的搭建过程从而可以大大降低实验成本，同时由于仿真环境中机器人关节响应速度更快，进而算法的迭代速度更快，可以更快地得到较好的策略。

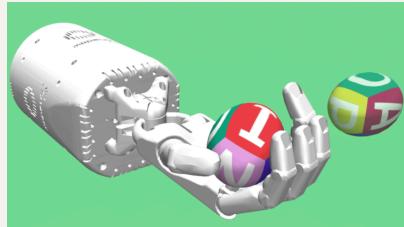


图 4: 机器人抓取任务的仿真环境

当然，仿真环境也并不是万能的，因为仿真环境和真实环境之间往往存在一定的差异，这就需要我们在设计仿真环境的时候尽可能全面地考虑到真实环境的各种因素，这也是一个非常重要的研究方向。除了简单的抓取任务之外，研究者们还在探索将强化学习应用于更加复杂的机器人任务，例如仓储搬运、机器人足球以及自动驾驶等等。

除了游戏和机器人领域之外，强化学习在金融领域也有所应用，例如股票交易、期货交易、外汇交易等。在股票交易中，我们的目标是通过买卖股票来最大化我们的资产。在这个过程中，我们需要不断地观测当前的股票价格，然后根据当前的价格来决定买入或卖出股票的数量，最后通过股票价格的变化来更新我们的资产。在这个过程中，我们的资产会随着股票价格的变化而变化，这就是奖励或惩罚，每次的买卖就是决策。当然，强化学习的应用还远不止如此，例如自动驾驶、推荐系统、交通派单、广告投放以及近来大火的 ChatGPT 等，这些都是。

强化学习方向概述

强化学习不仅应用十分广泛，而且从技术角度来讲其方向也非常多。在学习基础的强化学习知识之后，读者可根据自身的兴趣选择相应方向进行深入学习。本节将对强化学习的一些典型方向进行简要介绍，以便读者能够对强化学习有更加全面的认识，同时也为后续的学习做好铺垫，强化学习的典型应用主要如下。

多智能体强化学习

顾名思义，多智能体强化学习（multi-agent reinforcement learning, MARL）就是在多个智能体的环境下进行强化学习。与单智能体环境不同，在多智能体环境中通常存在非静态问题，即环境的状态不仅由智能体的动作决定，还受到其他智能体的动作的影响。例如在 AlphaStar 中，每个智能体都是一个《星际争霸》中的玩家，每个玩家都有自己的目标，例如攻击对方的基地或者防守自己的基地，这就导致环境的状态不仅由自己的动作决定，还受到其他玩家的动作的影响。

其次存在信号问题，即智能体之间可能需要进行通信以合作或竞争，如何高效地通信并从信号中学习是一个难题。还存在信誉分配问题，在多智能体的合作任务中，确定每个智能体对于整体目标的贡献（或责任）是一个挑战。多智能体环境通常也存在复杂的博弈场景，对于此类研究往往引入博弈论来找到环境中的纳什均衡或其他均衡策略，但这是一个复杂的挑战。

从数据中学习

从数据中学习，或者从演示中学习（learn from demonstration）包含丰富的门类，例如以模仿学习为代表的从专家数据中学习策略、以逆强化学（*inverse reinforcement learning*, IRL 为为代表的从人类数据中学习奖励函数和以及从人类反馈中学习（reinforcement learning from human feedback, RLHF）为代表的从人类标注的数据中学习奖励模型来进行微调（fine-tune）。此外，还包括离线强化学（offline reinforce learning）、世界模型（world model）等等，这些方法都利用了数据来辅助强化学，因此本书将它们同归为一类从数据中学习的方法。注意，这些方法的思路实际上是大相径庭的，完全可以作为一个单独的子方向来研究。

模仿学习是指在奖励函数难以明确定义或者策略本身就很难学出来的情况下，我们可以通过模仿人类的行为来学习到一个较好的策略。最典型的模仿策略之一就是行为克隆（behavioral cloning, BC），即将每一个状态-动作对视为一个训练样本，并使用监督学习的方法（如神经网络）来学习一个策略。但这种方法容易受到分布漂移（distribution shift）的影响。智能体可能会遇到从未见过的状态，导致策略出错。

逆强化学是指通过观察人类的行为来学习到一个奖励函数，然后通过强化学来学习一个策略。由于需要专家数据，逆强化学会受到噪声的影响，因此如何从噪声数据中学习到一个较好的奖励函数也是一个难题。

探索策略

探索策略（exploration strategy）。在强化学中，探索策略是一个非常重要的问题，即如何在探索和利用之间做出权衡。在探索的过程中，智能体会尝试一些未知的动作，从而可能会获得更多的奖励，但同时也可能会遭受到惩罚。而在利用的过程中，智能体会选择已知的动作，从而可能会获得较少的奖励，但同时也可能会遭受较少的惩罚。因此，如何在探索和利用之间做出权衡是一个非常重要的问题。目前比较常用的方法有 ϵ - greedy 和 置信上界（upper confidence bound, UCB）等等。此外，提高探索的本质也是为了避免局部最优问题，从而提高智能体的鲁棒性，近年来也有研究结合进化算法来提高探索的效率，例如 NEAT（neuro evolution of augmenting topologies）和 PBT（population based training）等算法，当然这些算法在提高探索的同时也会带来一定的计算成本。

实时环境

实时环境（**real-time environment**）。在实际应用中，智能体往往需要在实时或者在线环境中进行决策，例如自动驾驶、机器人等等。在这种情况下训练不仅会降低效率（实时环境响应动作更慢），而且还会带来安全隐患（训练过程中可能会出现意外）。解决这一问题的思路之一就是离线强化学习（**offline reinforcement learning**），即在离线环境中进行训练，然后将训练好的模型部署到在线环境中进行决策。但这种方法也存在着一定的问题，例如离线环境和在线环境之间可能存在着分布漂移，即两个环境的状态分布不同，这就导致了训练好的模型在在线环境中可能会出现意外。另外有一种是近两年比较流行的思路，即世界模型（**world model**），即在离线环境中训练一个世界模型，然后将世界模型部署到在线环境中进行决策。世界模型的思路是将环境分为两个部分，一个是世界模型，另一个是控制器。世界模型的作用是预测下一个状态，而控制器的作用是根据当前的状态来决策动作。这样就可以在离线环境中训练世界模型，然后将世界模型部署到在线环境中进行决策，从而避免了在线环境中的训练过程，提高了效率，同时也避免了在线环境中的安全隐患。但世界模型也存在着一定的问题，例如世界模型的预测误差会导致控制器的决策出错，因此如何提高世界模型的预测精度也是一个难题。

多任务强化学习

多任务强化学习（**multi-task reinforcement learning**）。这个问题在深度学习中也较为常见，在实际应用中，智能体往往需要同时解决多个任务，例如机器人需要同时完成抓取、搬运、放置等任务，而不是单一的抓取任务。在这种情况下，如何在多个任务之间做出权衡是一个难题。目前比较常用的方法有联合训练（**joint training**）和分层强化学习（**hierarchical reinforcement learning**）等等。联合训练的思路是将多个任务的奖励进行加权求和，然后通过强化学习来学习一个策略。分层强化学习的思路是将多个任务分为两个层次，一个是高层策略，另一个是低层策略。高层策略的作用是决策当前的任务，而低层策略的作用是决策当前任务的动作。这样就可以通过强化学习来学习高层策略和低层策略，从而解决多任务强化学习的问题。但分层强化学习也存在着一定的问题，例如高层策略的决策可能会导致低层策略的决策出错，因此如何提高高层策略的决策精度也是一个难题。

学习本书之前的一些准备

我们先介绍一下关于本书的初衷。其实目前强化学习相关的书籍在市面上已经琳琅满目了，但是这些普遍偏向理论，缺少一些实际的经验性总结，比如大佬们可能会通过数学推导来告诉你某某算法是可行的，可是一些实验细节和不同算法的对比很难在这些书籍中体现出来，理论与实践之间、公式与代码之间其实存在着一定的鸿沟。另一方面，由于信息时代知识的高速迭代，面对如海洋一般的信息，我们需要从中梳理出重点并快速学习，以便于尽快看到实际应用的效果，而这中间就不得不需要一个经验丰富的老师傅来带路了，这也是本书的初衷之一。笔者会基于大量的强化学习实践经验，对于理论部分删繁就简，并与实践紧密结合，以更通俗易懂的方式帮助读者们快速实践。

其次，在具体的学习之前，先给读者做一些基础的知识铺垫。第一，强化学习是机器学习的一个分支，因此读者需要具备一定的机器学习基础，例如基本的线性代数、概率论、数理统计等等。当然只需要读者们修过相关的大学课程即可，不必再去刻意回顾一些知识，原理部分跟随本书的推导即可。第二，在学习强化学习初期是不涉及深度神经网络相关的东西的，这一部分通常称为传统强化学习部分。尽管这部分的算法在今天已经不常用，但是其中蕴含的一些思想和技巧是非常重要的，因此读者们需要对这部分内容有所了解。在过渡到结合神经网络的深度强化学习部分之前，本书会花一章的时间帮助读者整理需要的深度学习知识。

深度学习在强化学习中扮演的角色主要是提供了一个强大的函数拟合能力，使得智能体能够处理复杂、高维度和非线性的环境。深度学习与强化学习之间的关系相当于眼睛和大脑的关系，眼睛是帮助大脑决策更好地观测世界的工具，对于一些没有眼睛的动物例如蚯蚓也可以通过其他的感官来观测并解析状态。再比如，同样大脑水平下，即相同的强化学习算法条件下，正常人要比双目失明的人日常的决策要更方便。但是，即使深度学习部分是相同的，例如正常大人和小孩都能通过眼睛观测世界，然由于大脑决策水平的差异也会让两者表现有所差异。总而言之，深度与强化在更复杂的环境下缺一不可。最后，尽管强化学习算法很多，但基本上就分为两类，即基于价值的和基于策略梯度的算法，这两种算法各有优势，读者们在学习之后根据实际需要选择即可。

马尔可夫决策过程

本章开始介绍马尔可夫决策过程的基本概念，包括马尔可夫性质、回报、状态转移矩阵等内容。马尔可夫决策过程是强化学习的核心问题模型，即想用强化学习来解决问题，首先需要将问题建模为马尔可夫决策过程，并明确状态空间、动作空间、状态转移概率和奖励函数等要素。此外，还介绍了策略、状态价值和动作价值等重要概念，这些概念在后续的强化学习算法中会频繁用到，务必牢记。

马尔可夫决策过程

在强化学习中，马尔可夫决策过程（Markov Decision Process, MDP）是用来描述智能体与环境交互的数学模型。如图 1 所示，智能体（Agent）与环境（Environment）在一系列离散的时步（time step）中交互，在每个时步 t ，智能体接收环境的状态 s_t ，并根据该状态选择一个动作 a_t 。执行该动作后，智能体会收到一个奖励 r_t ，同时环境会转移到下一个状态 s_{t+1} 。

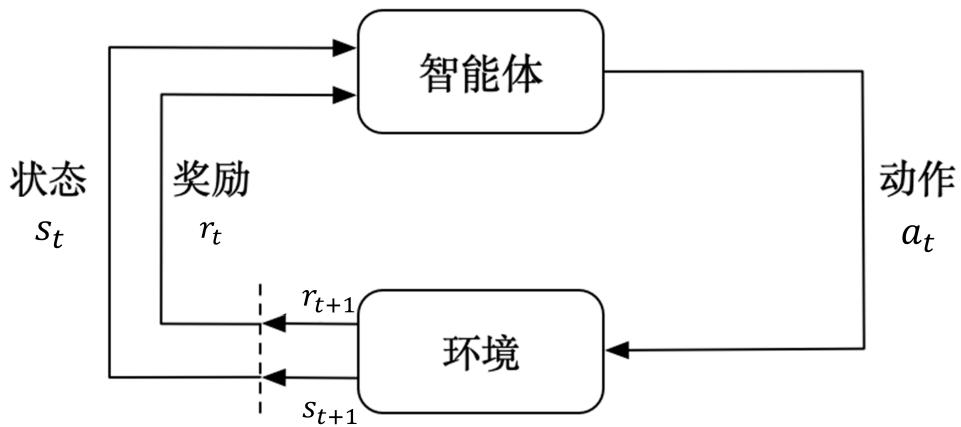


图 1: 智能体与环境的交互过程

这个过程不断重复，形成一条轨迹，如式 (1) 所示。

$$s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_t, a_t, r_t, \dots \quad (1)$$

完成一条完整的轨迹，即从初始状态到终止状态（例如玩游戏玩到最终的胜负结算阶段），也称为一个回合（episode），通常在有限的时步 T 后结束，即 $t = 0, 1, 2, \dots, T$ ， T 是回合的最大步数。

如果要用强化学习来解决问题，首先需要将问题建模为马尔可夫决策过程，即明确状态空间、动作空间、状态转移概率和奖励函数等要素。通常我们用一个五元组来定义马尔可夫决策过程，如式 (2) 所示。

$$MDP = (S, A, P, R, \gamma) \quad (2)$$

其中 S 是状态空间，表示所有可能的环境状态的集合， A 是动作空间，表示智能体可以选择的所有可能动作的集合， P 是状态转移概率矩阵，描述了在给定当前状态和动作的情况下，环境转移到下一个状态的概率分布， R 是奖励函数，定义了在特定状态下执行某个动作所获得的即时奖励， γ 是折扣因子，用于权衡当前奖励和未来奖励的重要性，其取值范围在 0 到 1 之间。其中状态转移矩阵和折扣因子将在下文详细展开说明。

马尔可夫性质

马尔可夫决策过程的核心假设是**马尔可夫性质 (Markov Property)**，即系统未来状态的概率分布只依赖于当前的状态和动作，而与过去的状态和动作无关，如式 (3) 所示。

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) = P(s_{t+1}|s_t, a_t) \quad (3)$$

然而，在真实世界中严格满足马尔可夫性质的情况并不多见，但大多情况下，我们依然可以通过适当的状态表示来近似满足马尔可夫性质。

例如在自动驾驶中，将当前车辆的位置、速度和周围环境信息作为状态表示，这些信息足以预测下一时刻的状态，而不需要考虑更早之前的历史数据，从而近似满足马尔可夫性质，这样的过程也叫做**部分可观测马尔可夫决策过程 (Partially Observable Markov Decision Process, POMDP)**。

状态转移矩阵

通常，马尔可夫决策过程通常指有限马尔可夫决策过程 (finite MDP)，即状态空间和动作空间都是有限的。如果状态空间或动作空间是无限的，通常需要采用其他方法进行建模，例如连续时间马尔可夫过程等。

既然状态数有限，就可以用一种状态流向图的形式表示智能体与环境交互过程中的走向。如图 2 所示，图中每个曲线箭头表示指向自己，对于状态 s_1 来说，有 0.2 的概率继续保持在 s_1 状态，同时也有 0.4 和 0.4 的概率转移到状态 s_2 和 s_3 。同理，其他状态之间也有类似的转移概率。

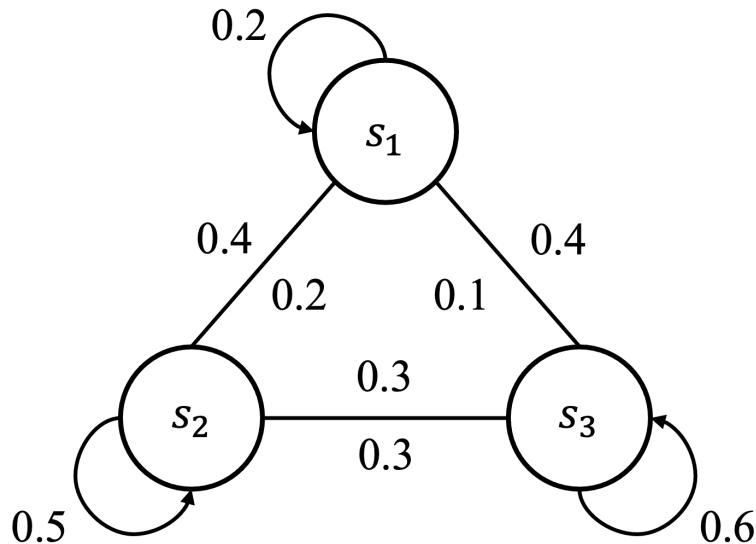


图 2: 马尔可夫链

注意，图 2 中并没有包含动作和奖励等元素，因此严格来说它表示的是**马尔可夫链 (Markov Chain)**，又叫做离散时间的马尔可夫过程 (Markov Process)，但它与马尔可夫决策过程有着密切的联系，都是基于马尔可夫性质构建的。

我们用一个概率来表示状态之间的切换，如式 (4) 所示。

$$P_{ss'} = P(S_{t+1} = s' | S_t = s) \quad (4)$$

即当前状态是 s 时，下一个状态是 s' 的概率，其中大写的 S 表示所有状态的集合，即 $S = \{s_1, s_2, s_3\}$ 。例如， $P_{12} = P(S_{t+1} = s_2 | S_t = s_1) = 0.4$ 表示当前时步的状态是 s_1 ，下一个时步切换到 s_2 的概率为 0.4。

拓展到所有状态，可以把这些概率绘制成一个状态转移表，如表 1 所示。

表 1: 马尔可夫状态转移表

$S_t = s_1$	0.2	0.4	0.4
$S_t = s_2$	0.2	0.5	0.6

	$S_{t+1} = s_1$	$S_{t+1} = s_1$	$S_{t+1} = s_3$
$S_t = s_3$	0.1	0.3	0.6

在数学上也可以用矩阵来表示，如式 (5) 所示。

$$P_{ss'} = \begin{bmatrix} 0.2 & 0.4 & 0.4 \\ 0.2 & 0.5 & 0.3 \\ 0.1 & 0.3 & 0.6 \end{bmatrix} \quad (5)$$

这个矩阵就叫做**状态转移矩阵 (State Transition Matrix)**，拓展到所有状态可表示为式 (6) 所示。

$$P_{ss'} = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix} \quad (6)$$

其中 n 表示状态数，注意从同一个状态出发转移到其他状态的概率之和是等于 1 的，即 $\sum_{j=1}^n p_{ij} = 1$ ， $i = 1, 2, \dots, n$ 。**状态转移矩阵是环境的一部分**，描述了环境状态之间的转移关系。

目标与回报

在强化学习中，智能体的目标是通过与环境的交互，学习一个最优策略，使得在每个状态下选择的动作能够最大化累积的奖励。这个累积的奖励通常被称为**回报 (Return)**，如式 (7) 所示。

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (7)$$

表示从时间步 t 开始，未来所有奖励的加权和，其中 γ 是折扣因子，在 0 到 1 之间。折扣因子的作用是用来控制未来奖励在当前决策中的重要性。当 γ 接近 0 时，智能体更关注当前的奖励，而忽略未来的奖励；当 γ 接近 1 时，智能体会更加重视未来的奖励。

折扣因子一方面在数学上确保回报 G_t 的收敛性，另一方面也代表了时间价值，就像经济学中的货币折现，同样面值的货币现在拿到手中比未来拿到更有价值。此外，折扣因子还可以用来衡量智能体对长期回报的关注度，或者说“能看到多远”，称之为有效视界 (effective horizon)，如式 (8) 所示。

$$H_{eff} = \frac{1}{1 - \gamma} \quad (8)$$

当 $\gamma = 0.9$ 时， $H_{eff} = 10$ ，表示智能体主要关注未来 10 个时步内的奖励；当 $\gamma = 0.99$ 时， $H_{eff} = 100$ ，表示智能体关注未来 100 个时步内的奖励。

此外，当前时步的回报 G_t 跟下一个时步 G_{t+1} 的回报是有所关联的，即递归地定义，如式 (9) 所示。

$$\begin{aligned} G_t &\doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \\ &= r_{t+1} + \gamma (r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \cdots) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned} \quad (9)$$

策略与价值

策略

策略 (Policy) 表示智能体在每个状态下选择动作的规则或方法，用 π 表示。如式 (10) 所示，策略是一个从状态到动作的映射或函数。

$$\pi(a|s) = P(A_t = a|S_t = s) \quad (10)$$

表示在状态 s 下选择动作 a 的概率分布。策略可以是确定性的 (deterministic)，即在每个状态下总是选择同一个动作，或者是随机性的 (stochastic)，即在每个状态下根据一定的概率分布选择动作。

状态价值

状态价值函数 (State-Value Function) 表示在给定状态下，按照某种策略 π 进行决策所能获得的回报期望值，用 $V_\pi(s)$ 表示，如式 (11) 所示。

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots | S_t = s] \\ &= \mathbb{E}_\pi[G_t | S_t = s] \end{aligned} \quad (11)$$

举例来说，假设智能体处于一个 3×3 的网格世界中，目标是走到右下角，对应获得 10 分的奖励，走到其他格子的奖励为 0，策略是随机选择一个方向（上、下、左、右）移动一步。

如果初始状态 s_0 即起点在左上角，按照当前策略，平均可能需要 10 才能到达终点，每步奖励为 0，到达终点时获得 10 分奖励，设折扣因子 $\gamma = 0.9$ ，则从起点状态出发的状态价值计算如式 (12) 所示。

$$V_\pi(s_0) = 0 + 0.9^1 \times 0 + 0.9^2 \times 0 + \dots + 0.9^{10} \times 10 \approx 3.49 \quad (12)$$

动作价值

动作价值函数 (Action-Value Function) 表示在给定状态 s 和动作 a 下，按照某种策略 π 进行决策所能获得的回报期望值，用 $Q_\pi(s, a)$ 表示，如式 (13) 所示。

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] \quad (13)$$

状态价值与动作价值的关系

状态价值函数和动作价值函数之间存在密切的关系，如式 (14) 所示。

$$V_\pi(s) = \sum_{a \in A} \pi(a | s) Q_\pi(s, a) \quad (14)$$

换句话说，状态价值是对所有可能的动作价值的加权平均，而动作价值则是对特定动作的评价。

状态价值反映了策略本身的好坏，它不关心智能体在状态 s 下选择了哪个具体动作，而是关注在该状态下按照策略 π 进行决策所能获得的整体回报期望值。动作价值则更具体地反映了在特定状态下选择某个动作所能获得的回报期望值，即不仅考虑智能体所处的状态，还考虑了智能体在该状态下选择的具体动作。

有模型与无模型

在谈到有模型 (Model-Based) 与无模型 (Model-Free) 方法时，这里的模型通常指与智能体交互的环境模型，即对环境的状态转移概率和奖励函数进行建模。根据是否使用环境模型，强化学习方法可以分为有模型方法和无模型方法两大类。

有模型的方法利用环境模型来进行规划和决策，通常包括动态规划（Dynamic Programming）等方法。这些方法通过对环境的状态转移概率和奖励函数进行建模，能够在不与环境直接交互的情况下，预测未来的状态和奖励，从而制定最优策略。

无模型的方法则不依赖于环境模型，包括 Q-Learning、SARSA 等算法，这些方法通过与环境的直接交互，学习状态价值函数或动作价值函数，从而逐步改进策略。无模型方法通常更适用于复杂或未知的环境，因为它们不需要对环境进行显式建模，在强化学习中应用更为广泛。

预测与控制

预测（Prediction）问题是指在给定策略 π 的情况下，评估该策略的好坏，即计算状态价值函数 $V_\pi(s)$ 或动作价值函数 $Q_\pi(s, a)$ 。预测问题的目标是了解在当前策略下，智能体在不同状态下能够获得的回报期望值。用于预测任务的算法主要包括蒙特卡洛方法（Monte Carlo）、时序差分学习（Temporal Difference, TD）等，后续会详细介绍。

控制（Control）问题是指在给定环境模型的情况下，寻找最优策略 π^* ，使得在每个状态下选择的动作能够最大化累积的回报。控制问题的目标是通过与环境的交互，学习一个最优策略，使得智能体能够在不同状态下获得最大的回报。用于控制任务的算法主要包括动态规划（Dynamic Programming）、Q 学习（Q-Learning）、策略梯度方法（Policy Gradient）等，后续也会详细介绍。

复杂问题中通常需要同时解决预测和控制问题，即在学习最优策略的过程中，同时评估当前策略的好坏，从而不断改进策略，最终收敛到最优策略。

思考

强化学习所解决的问题一定要严格满足马尔可夫性质吗？请举例说明。

不一定。例如在围棋游戏场景中，不仅需要考虑当前棋子的位置，还需要考虑棋子的历史位置，因此不满足马尔可夫性质。但依然可以使用强化学习的方法进行求解，例如在 AlphaGO 论文中使用了蒙特卡洛树搜索算法来解决这个问题。在一些时序性场景中，也可以通过引入记忆单元来解决这个问题，例如在 DQN 算法中，使用了记忆单元来存储历史状态，从而解决了这个问题，尽管它也不满足马尔可夫性质。

马尔可夫决策过程主要包含哪些要素？

马尔可夫决策 $< S, A, R, P, \gamma >$ 主要包含状态空间 S 、动作空间 A 、奖励函数 R 、状态转移矩阵 P 、折扣因子 γ 等要素，其中状态转移矩阵 P 是环境的一部分，而其他要素是智能体的一部分。在实际应用中，通常还考虑值函数 V 和策略函数 π 等要素，值函数用于某个状态下的长期累积奖励，策略函数用于某个状态下的动作选择。

马尔可夫决策过程引入折扣因子 γ 的作用是什么？如何理解折扣因子的意义？

折扣因子 γ 的作用是用来控制未来奖励在当前决策中的重要性。它的取值范围在 0 到 1 之间。当 γ 接近 0 时，智能体更关注当前的奖励，而忽略未来的奖励；当 γ 接近 1 时，智能体会更加重视未来的奖励。一方面在数学上能够确保回报 G_t 的收敛性，另一方面也代表了时间价值，就像经济学中的货币折现，同样面值的货币现在拿到手中比未来拿到更有价值。

马尔可夫决策过程与金融科学中的马尔可夫链有什么区别与联系？

马尔可夫链是一个随机过程，其下一个状态只依赖于当前状态而不受历史状态的影响，即满足马尔可夫性质。马尔可夫链由状态空间、初始状态分布和状态转移概率矩阵组成。马尔可夫决策过程是一种基于马尔可夫链的决策模型，它包含了状态、行动、转移概率、奖励、值函数和策略等要素。马尔可夫决策过程中的状态和状态转移概率满足马尔可夫性质，但区别在于它还包括了行动、奖励、值函数和策略等要素，用于描述在给定状态下代理如何选择行动以获得最大的长期奖励。

有模型与免模型算法的区别？举一些相关的算法？

有模型算法在学习过程中使用环境模型，即环境的转移函数和奖励函数，来推断出最优策略。这种算法会先学习环境模型，然后使用模型来生成策略。因此，有模型算法需要对环境进行建模，需要先了解环境的转移函数和奖励函数，例如动态规划等算法。无模型算法不需要环境模型，而是直接通过试错来学习最优策略。这种算法会通过与环境的交互来学习策略，不需要先了解环境的转移函数和奖励函数。无模型算法可以直接从经验中学习，因此更加灵活，例如 Q-learning、Sarsa 等算法。

举例说明预测与控制的区别与联系。

区别：预测任务主要是关注如何预测当前状态或动作的价值或概率分布等信息，而不涉及选择动作的问题；控制任务则是在预测的基础上，通过选择合适的动作来最大化累计奖励，即学习一个最优的策略。**联系：**预测任务是控制任务的基础，因为在控制任务中需要对当前状态或动作进行预测才能选择最优的动作；控制任务中的策略通常是根据预测任务中获得的状态或动作价值函数来得到的，因此预测任务对于学习最优策略是至关重要的。以赌博机问题为例，预测任务是估计每个赌博机的期望奖励（即价值函数），控制任务是选择最优的赌博机来最大化累计奖励。在预测任务中，我们可以使用多种算法来估计每个赌博机的期望奖励，如蒙特卡罗方法、时间差分方法等。在控制任务中，我们可以使用贪心策略或 ϵ -贪心策略来选择赌博机，这些策略通常是根据预测任务中得到的每个赌博机的价值函数来确定的。因此，预测任务对于控制任务的实现至关重要。

动态规划

动态规划是一种用于解决复杂问题的数学方法，广泛应用于计算机科学、经济学等领域，其核心思想是将问题分解为更小的子问题，并存储这些子问题的解决方案，以避免重复计算，从而提高效率。

在强化学习中，动态规划被用于求解值函数和最优策略，核心概念是贝尔曼方程，它描述了状态价值函数和动作价值函数的递归关系。

基于动态规划的强化学习算法主要包括策略迭代和价值迭代等，这些算法依赖于对环境的完全了解，即需要知道状态转移概率和奖励函数，即属于有模型的方法。这两种算法虽然现在很少直接应用于实际问题，但它们为理解更复杂的强化学习算法奠定了基础。

动态规划

动态规划 (dynamic programming, DP) 是一种用于解决复杂问题的数学优化方法，在计算机科学、管理科学和经济学等领域都有着广泛的应用。它通过将问题分解为更小的子问题，并存储这些子问题的解决方案，以避免重复计算，从而提高效率。

动态规划能够解决的问题通常具备三个性质：**重叠子问题** (overlapping subproblems)、**最优化原理** (optimal substructure) 和**无后效性** (no aftereffect)。重叠子问题指的是子问题在求解过程中会被多次计算，而最优子结构则意味着一个问题的最优解可以通过其子问题的最优解来构建。无后效性则表示未来只依赖于当前状态，不依赖于过去，即未来的独立性，这与马尔可夫性质相契合。

动态规划通常采用自底向上的方法，通过迭代地解决子问题，最终得到原始问题的解决方案。常见的动态规划算法包括斐波那契数列、背包问题和路径规划问题等。

为帮助理解动态规划的思想，这里以一个路径规划问题为例进行说明。如图 1 所示，在一个 $m \times n$ 的网格世界中，机器人以左上角为起点，右下角为终点，每次只能向右或向下移动一步，这里的问题是需要计算出从起点到终点的不同路径数量。

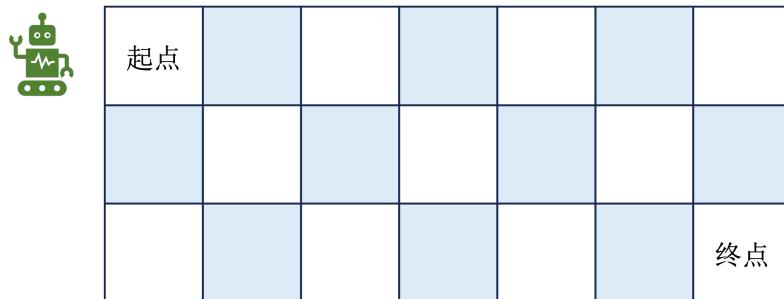


图 1: 路径之和

使用动态规划的方法来解决这个问题的话，主要包含几个步骤：**确定状态，写出状态转移方程和寻找边界条件**。

首先可以将状态定义为 $f(i, j)$ ，表示从左上角即坐标为 $(0, 0)$ 到坐标 (i, j) 的路径数量，其中 $i \in [0, m)$ 以及 $j \in [0, n)$ 。由于机器人只能向右或者向下走，所以当机器人处于 (i, j) 的位置时，它的前一个坐标只能是上边一格 $(i, j - 1)$ 或者左边一格 $(i - 1, j)$ ，这样一来就能建立出一个状态之间的关系，如式 (1) 所示。

$$f(i, j) = f(i - 1, j) + f(i, j - 1) \quad (1)$$

即走到当前位置 (i, j) 的路径数量等于走到前一个位置 $(i, j - 1)$ 和 $(i - 1, j)$ 的所有路径之和，这个就是状态转移方程。

然后再考虑一些边界条件，首先 i 和 j 是不能等于 0 的，因为这种情况下会出现 $f(0, 0) + f(-1, 0) + f(0, -1)$ ， $f(-1, 0)$ 或者 $f(0, -1)$ 在本题中是没有意义的。换句话说， $f(0, 0)$ 就是该题的一个边界，在这种情况下机器人在起始点，从起始点到起始点 $(0, 0)$ 对应的路径数量 $f(0, 0)$ 必然是 1，对于 $i \neq 0, j = 0$ ，此时机器人会一直沿着网格左边沿往下走，这条路径上的所有 $f(i, j)$ 也都会是 1， $i = 0, j \neq 0$ 的情况同理。

代入边界条件后，状态转移方程可以完善如式 (2) 所示。

$$f(i, j) = \begin{cases} 0, & i = 0, j = 0 \\ 1, & i = 0, j \neq 0 \\ 1, & i \neq 0, j = 0 \\ f(i - 1, j) + f(i, j - 1) & \end{cases} \quad (2)$$

实现如代码 1 所示。

代码 1: 路径问题求解

```
def solve(m, n):
    # 初始化边界条件
    f = [[1] * n] + [[1] + [0] * (n - 1) for _ in range(m - 1)]
    # 状态转移
    for i in range(1, m):
        for j in range(1, n):
            f[i][j] = f[i - 1][j] + f[i][j - 1]
    return f[m - 1][n - 1]
```

输入 $m = 7, n = 3$ 时，最终输出结果为 28，表示从起点到终点一共有 28 种不同的路径，对应的解析流程如图 2 所示。

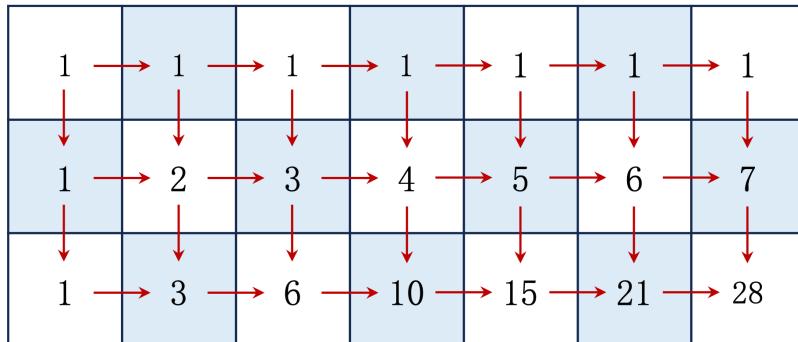


图 2: 路径之和解析

在强化学习中，基于动态规划的算法主要有两种，一是策略迭代 (policy iteration, PI)，二是价值迭代 (value iteration, VI)。其中策略迭代由两部分组成，分别是策略评估 (policy evaluation) 和策略改进 (policy improvement)。

价值迭代则是将策略评估和策略改进合并在一起进行。这两种算法都依赖于对环境的完全了解，即需要知道状态转移概率和奖励函数，因此在实际应用中受到了一定的限制。然而，它们为理解更复杂的强化学习算法奠定了基础。

贝尔曼方程

状态价值形式

类比于回报公式 $G_t = R_{t+1} + \gamma G_{t+1}$, 状态价值也有类似的递归公式, 如式 (3) 所示。

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi [G_t | S_t = s] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma G_{t+1} | S_t = s] \text{ 回报递归公式} \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s] \text{ 状态价值定义} \\ &= \sum_{a \in A} \pi(a | s) \sum_{s' \in S} p(S_{t+1} = s' | S_t = s, A_t = a) [R(s, a) + \gamma V_\pi(s')] \text{ 全期望公式展开} \\ &= \sum_a \pi(a | s) \sum_{s'} p(s' | s, a) [r + \gamma V_\pi(s')] \text{ 简写} \end{aligned} \quad (3)$$

其中 $p(s' | s, a)$ 是状态转移概率, 即在状态 s 下采取动作 a 后转移到状态 s' 并获得奖励 r 的概率, 这就是**贝尔曼方程**的状态价值函数形式。

贝尔曼方程相当于动态规划中的状态转移方程, 将整体期望拆分“当前+未来”两部分递归地计算。它也进一步表示了当智能体在环境中处于某个状态 s 时, 它未来的“好坏”(即回报 Return) 不仅取决于当前获得的奖励 r , 还取决于之后的状态中能获得的所有未来奖励。

动作价值形式

类似地, 贝尔曼方程的动作价值形式如式 (4) 所示。

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) \sum_{a' \in A} \pi(a' | s') Q_\pi(s', a') \quad (4)$$

贝尔曼最优方程

在一个马尔可夫决策过程中, 我们可以找到一个最优策略 π^* , 使得从任意状态 s 出发, 获得的期望回报最大, 如式 (5) 所示。

$$\begin{aligned} V^*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma V^*(s')] \end{aligned} \quad (5)$$

这个公式就是**贝尔曼最优方程 (Bellman optimality equation)**, 它对于后面要讲的策略迭代算法具有一定的指导意义。同理, 对应的动作价值函数形式如式 (6) 所示。

$$\begin{aligned} Q^*(s, a) &= \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} Q^*(s', a') \right] \end{aligned} \quad (6)$$

策略迭代

在最优策略 π^* 下, 对应的状态和动作价值函数也都是最优的, 即 $V^*(s)$ 和 $Q^*(s)$ 。但是实际求解中在优化策略的过程中, 同时我们还需要优化状态和动作价值函数, 这其实是一个多目标优化的问题。

策略迭代算法的思路是分成两个步骤，首先固定策略 π 不变，然后估计对应的状态价值 V ，这一叫做**策略估计 (policy evaluation)**。然后根据估计好的状态价值函数 V 结合策略推算出动作价值函数 Q ，并对 Q 函数优化然后进一步改进策略，这一步叫**策略改进 (policy improvement)**。在策略改进的过程中一般是通过贪心策略来优化的，即选择使得 $Q(s, a)$ 最大的动作 a ，更新策略 π ，如式 (7) 所示。

$$\pi'(a|s) = \begin{cases} 1, & a = \arg \max_a Q(s, a) \\ 0, & \text{else} \end{cases} \quad (7)$$

然后在策略改进时选择最大的 $Q(s, a)$ 值来更新。在一轮策略估计和改进之后，又会进入新一轮策略估计和改进，直到收敛为止。

如图 3 所示，(a) 描述了上面所说的策略估计和改进持续迭代的过程，(b) 则描述了在迭代过程中策略 π 和状态价值函数 V 最后会同时收敛到最优。

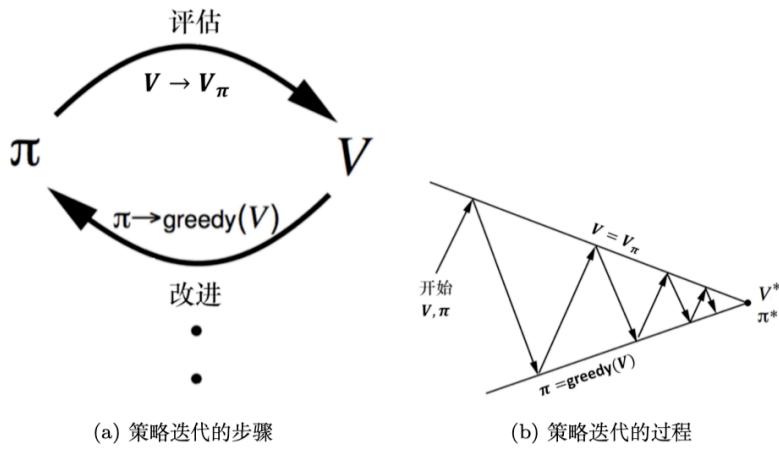


图 3: 策略迭代的收敛过程

算法流程的伪代码如图 4 所示。

策略迭代算法

```
1: 初始化状态价值函数  $V(s)$  和策略  $\pi(s)$ 
2: 策略估计:
3: repeat
4:    $\Delta \leftarrow 0$ 
5:   repeat
6:      $v \leftarrow V(s)$ 
7:      $V(s) \leftarrow \sum_{s',r} p(s',r | s, \pi(s)) [r + \gamma V(s')]$ 
8:      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
9:   until 遍历所有的状态  $s \in S$ 
10:  until  $\Delta < \theta$ 
11: 策略改进:
12:  $stable\_flag \leftarrow true$ 
13: repeat
14:   根据策略  $\pi(a|s)$  生成动作  $a_{temp}$ 
15:   更新策略:  $\pi(a|s) \leftarrow \arg \max_a \sum_{s',r} p(s',r | s, a) [r + \gamma V(s')]$ 
16:   if  $a_{temp} \neq \pi(a|s)$  then
17:     说明策略还未收敛,  $stable\_flag \leftarrow false$ 
18:   end if
19:   until 遍历所有的状态  $s \in S$ 
20:   if  $stable\_flag \leftarrow true$  then
21:     结束迭代并返回最优策略  $\pi \approx \pi_*$  和状态价值函数  $V \approx V_*$ 
22:   else
23:     继续执行策略估计 .
24:   end if
```

图 4: 策略迭代算法流程

价值迭代

价值迭代算法相对于策略迭代更加直接, 如式 (8) 所示。

$$V(s) \leftarrow \max_{a \in A} \left(R(s, a) + \gamma \sum_{s' \in S} p(s' | s, a) V(s') \right) \quad (8)$$

价值迭代算法流程的伪代码如图 5 所示。

价值迭代算法

```

1: 初始化一个很小的参数阈值  $\theta > 0$ , 以及状态价值函数  $V(s)$ , 注意终止
   状态的  $V(s_T) = 0$ 
2: repeat
3:    $\Delta \leftarrow 0$ 
4:   repeat
5:      $v \leftarrow V(s)$ 
6:      $V(s) \leftarrow \max_a \sum_{s',r} p(s',r | s,a) [r + \gamma V(s')]$ 
7:      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
8:   until 遍历所有的状态  $s \in S$ 
9: until  $\Delta < \theta$ 
10: 输出一个确定性策略  $\pi \approx \pi_*$ ,
    且  $\pi(s) = \arg \max_a \sum_{s',r} p(s',r | s,a) [r + \gamma V(s')]$ 

```

图 5: 价值迭代算法伪代码

首先将所有的状态价值初始化，然后不停地对每个状态迭代，直到收敛到最优价值 V^* ，并且根据最优价值推算出最优策略 π^* 。这样其实更像是一个动态规划本身的思路，而不是强化学习的问题。这种情况下，其实比策略迭代算法要慢得多，尽管两种方法都需要多次遍历。但是在策略迭代算法中考虑了中间每个时步可能遇到的最优策略并及时加以改进，这意味着就算策略在早期并不完美（也许需要改进），策略迭代仍能够更快地接近最优解。

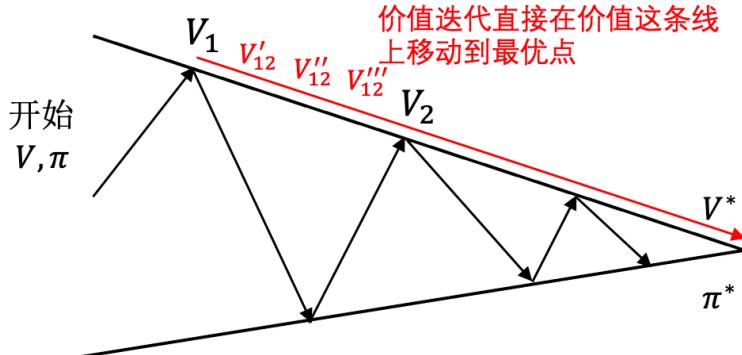


图 6: 策略迭代与价值迭代收敛过程的区别

举例来说，回顾一下策略迭代的收敛过程，如图 6 所示，我们知道策略迭代是不停地在 V 和 π 这两条线之间“跳变”直到收敛到 V^* 。这种“跳变”是几乎不需要花费时间的，只是一个 π 与 V 互相推算的过程，通过一个公式就能得到，也就是策略估计和策略改进之间的切换过程。而在各自的线上，比如价值函数这条线从 V_1 到 V_2 这个点是需要更多时间的，这其实就是一个策略估计的过程，需要遍历到所有状态，在策略这条线上同理。而实际上 V_1 到 V_2 中间也可能存在更多个点，比如 $V'_{12}, V''_{12}, V'''_{12}$ ，每次在这些点之间移动是需要遍历所有的状态的，只是在策略迭代算法中借助了策略这条线跳过了中间的 $V'_{12}, V''_{12}, V'''_{12}$ 这些点，而在价值迭代算法的时候会经过价值这条线上的所有点，直到最优，从这个角度来看策略迭代算法是要比价值迭代更快的。

思考

动态规划问题的主要性质有哪些？

动态规划主要性质包括最优化原理、无后效性和有重叠子问题，其中无后效性指的是某状态以后的过程不会影响以前的状态，这十分契合马尔可夫性质，因此动态规划问题可以看作是马尔可夫决策过程的一种特殊情况。

状态价值函数和动作价值函数之间的关系是什么？

状态价值函数是所有可能动作的动作价值函数的平均值，也就是说，对于一个状态 s ，其状态价值函数 $V(s)$ 等于所有可能动作 a 的动作价值函数 $Q(s, a)$ 的平均值，即 $V(s) = 1/|A(s)| * \sum Q(s, a)$ ，其中 $|A(s)|$ 表示在状态 s 下可用的动作数。

策略迭代和价值迭代是有模型还是无模型的方法？

策略迭代和价值迭代都是有模型的方法，因为它们都需要知道环境的状态转移概率和奖励函数，以便进行状态价值函数和动作价值函数的计算和更新。

策略迭代和价值迭代哪个算法速度会更快？

通常，价值迭代算法的收敛速度比策略迭代算法更快。因为价值迭代算法在每次迭代中更新所有状态的价值函数，而策略迭代算法需要在每次迭代中更新策略和状态的价值函数，因此策略迭代算法的计算量比价值迭代算法更大。此外，策略迭代算法的每次迭代都需要进行一次策略评估和一次策略改进，而价值迭代算法只需要进行一次价值迭代，因此策略迭代算法的迭代次数通常比价值迭代算法多。

蒙特卡洛方法

蒙特卡洛方法的核心思想是通过大量的随机采样来近似估计期望或积分。在强化学习中，一方面可以用来解决预测问题，即估计状态价值函数 $V(s)$ 或动作价值函数 $Q(s, a)$ 。另一方面，可以用来优化策略，即通过采样来评估和改进策略来解决控制问题。

蒙特卡洛预测包括首次访问法和每次访问法两种基本方法，前者只在每个状态的首次访问时更新价值估计，后者则在每次访问时都进行更新。

蒙特卡洛控制则结合了蒙特卡洛预测和策略改进，通过采样来估计动作价值函数，并基于该估计来改进策略，最终实现策略的优化。

状态价值计算示例

为帮助理解蒙特卡洛方法，我们先举一个简单的例子来根据定义计算状态价值，然后再介绍蒙特卡洛预测算法。

如图 1 所示，考虑智能体在 2×2 的网格中使用随机策略进行移动，以左上角为起点，右下角为终点，规定每次只能向右或向下移动，动作分别用 a_1 和 a_2 表示。用智能体的位置不同的状态，即 s_1, s_2, s_3, s_4 ，初始状态为 $S_0 = s_1$ 。设置每走一步接收到的奖励为 -1 ，折扣因子 $\gamma = 0.9$ ，目标是计算各个状态的价值函数 $V(s)$ 。

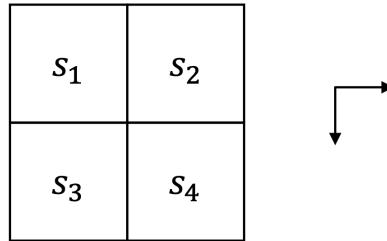


图 1: 2×2 网格示例

回顾状态价值函数的定义，如式 (1) 所示。

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\ &= \mathbb{E}_\pi[G_t | S_t = s] \end{aligned} \quad (1)$$

现在根据定义来分别计算各状态的价值，首先由于 s_4 是终止状态，因此 $V(s_4) = 0$ 。

接下来计算 s_2 的价值 $V(s_2)$ ，从 s_2 出发只能向下走到达终点 s_4 ，对应的轨迹为 $\tau_1 = \{s_2, a_2, r(s_2, a_2), s_4\}$ ，回报为 $G_{\tau_1} = r(s_2, a_2) = -1$ ，因此 $V(s_2) = G_{\tau_1} = -1$ 。

然后计算 s_3 的价值 $V(s_3)$ ，从 s_3 出发只能向右走到达终点 s_4 ，对应的轨迹为 $\tau_2 = \{s_3, a_1, r(s_3, a_1), s_4\}$ ，回报为 $G_{\tau_2} = r(s_3, a_1) = -1$ ，因此 $V(s_3) = G_{\tau_2} = -1$ 。

最后计算起始状态 s_1 的价值 $V(s_1)$ ，从 s_1 出发有两条可能的轨迹，其一是 $s_1 \rightarrow s_2 \rightarrow s_4$ ，其二是 $s_1 \rightarrow s_3 \rightarrow s_4$ ，对应的轨迹分别如式 (2) 和 (3) 所示。

$$\tau_3 = \{s_1, a_1, r(s_1, a_1), s_2, a_2, r(s_2, a_2), s_4\} \quad (2)$$

$$\tau_4 = \{s_1, a_2, r(s_1, a_2), s_3, a_1, r(s_3, a_1), s_4\} \quad (3)$$

相应地，对应的回报计算分别如式 (4) 和 (5) 所示。

$$G_{\tau_3} = r(s_1, a_1) + \gamma r(s_2, a_2) = (-1) + 0.9 * (-1) = -1.9 \quad (4)$$

$$G_{\tau_4} = r(s_1, a_2) + \gamma r(s_3, a_1) = (-1) + 0.9 * (-1) = -1.9 \quad (5)$$

由于智能体采用随机策略，因此两条轨迹的概率相等，均为 0.5。因此， $V(s_1)$ 可以表示为式 (6)。

$$V(s_1) = 0.5 * G_{\tau_3} + 0.5 * G_{\tau_4} = 0.5 * (-1.9) + 0.5 * (-1.9) = -1.9 \quad (6)$$

综上所述，各状态的价值函数结果如表 1 所示。

表 1: 各状态的价值函数

状态	s_1	s_2	s_3	s_4
价值	-1.9	-1.0	-1.0	0.0

下面将介绍蒙特卡洛方法是如何通过采样来估计状态价值的。

蒙特卡洛预测

蒙特卡洛估计

蒙特卡洛估计是一种用随机采样近似求期望、积分或概率分布特征的通用方法。换句话说，如果想求一个复杂的数学期望（或积分），而无法直接解析求解时，就可以用大量随机样本的平均值去逼近它。

假设我们想要估计某个函数 $f(x)$ 的期望，如式 (7) 所示。

$$\mathbb{E}[f(X)] = \int f(x)p(x)dx \quad (7)$$

其中 $p(x)$ 是随机变量 X 的概率密度函数。直接计算这个积分可能很复杂，但我们可以用蒙特卡洛采样来近似估计它。具体步骤如下：

1. 从概率分布 $p(x)$ 中采样 N 个独立同分布的样本 $\{x_1, x_2, \dots, x_N\}$ 。
2. 计算函数值的平均，如式 (8) 所示。

$$\hat{\mathbb{E}}[f(X)] = \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (8)$$

根据大数定律，当样本数量 N 足够大时，估计值 $\hat{\mathbb{E}}[f(X)]$ 会收敛到真实的期望值 $\mathbb{E}[f(X)]$ 。

为帮助理解，我们来演示如何通过蒙特卡洛方法来估计圆周率 π 。考虑一个单位正方形内切一个单位圆，圆的面积为 $\pi r^2 = \pi$ ，正方形的面积为 4。如果我们在正方形内随机撒点，落在圆内（即满足 $x^2 + y^2 \leq 1$ ）的点数与总点数的比例应该接近于圆的面积与正方形面积的比例，即 $\pi/4$ 。

用 `Python` 代码实现这个过程，如代码 1 所示。

代码 1: 使用蒙特卡洛方法估计策略

```

import random

def monte_carlo_pi(num_samples=1000000):
    count_in_circle = 0
    for _ in range(num_samples):
        x, y = random.random(), random.random()
        if x**2 + y**2 <= 1:
            count_in_circle += 1
    pi_estimate = 4 * count_in_circle / num_samples
    return pi_estimate

print("Estimated π:", monte_carlo_pi())

```

运行代码后，可以得到一个接近 π 的估计值。随着采样数量的增加，估计值会越来越精确。

蒙特卡洛预测 (Monte Carlo Prediction) 则指的是，在强化学习中，利用蒙特卡洛估计来预测给定策略 π 下的状态价值 $V_\pi(s)$ 。具体思路是多次完整地执行策略 π ，每次执行都会产生一条完整的轨迹（从初始状态到终止状态），然后根据这些轨迹来计算各个状态的回报，最后取平均作为该状态的价值估计，如式 (9) 所示。

$$V_\pi(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} G_t^{(i)} \quad (9)$$

增量式更新

在实际应用中，由于状态空间可能非常大，估计状态价值所需的轨迹数量可能上万甚至更多。一方面，轨迹是通过智能体与环境交互产生的，这一交互过程可能也会非常耗时；另一方面，存储和处理大量轨迹数据也会带来计算和内存的压力。

为了解决这些问题，蒙特卡洛预测通常采用增量式更新的方式来估计状态价值，即**边采样边更新**，而不是等采样完所有轨迹后再进行批量更新。

如式 (10) 所示，增量式更新的基本思想是每次采样到一个新的回报 G 后，立即用它来更新对应状态 s 的价值估计 $V(s)$ 。

$$V(s) \leftarrow V(s) + \frac{1}{N(s)} [G - V(s)] \quad (10)$$

其中 $N(s)$ 是状态 s 被访问的次数， G 是当前采样到的回报。或者使用常数步长，如式 (11) 所示。

$$V(s) \leftarrow V(s) + \alpha [G - V(s)] \quad (11)$$

其中 $\alpha \in (0, 1]$ 是学习率， α 越大，收敛速度很快但波动也较大； α 越小，收敛速度较慢但更稳定。

可以发现，增量式更新的核心思想如式 (12) 所示。

$$\text{新的估计值} \leftarrow \text{旧的估计值} + \text{步长} \times (\text{目标值} - \text{旧的估计值}) \quad (12)$$

首次访问蒙特卡洛

在增量式更新的基础上，蒙特卡洛方法主要分成两种算法，一种是首次访问蒙特卡洛 (first-visit Monte Carlo, FVMC) 方法，另外一种是每次访问蒙特卡洛 (every-visit Monte Carlo, EVMC) 方法。FVMC 方法主要包含两个步骤，首先是产生一个回合的完整轨迹，然后遍历轨迹计算每个状态的回报。

我们先来看首次访问蒙特卡洛 (FVMC) 方法的具体实现，算法流程如图 2 所示。

首次访问蒙特卡洛算法

- 1: 初始化：状态价值 $V(s)$, 回报字典 $\text{Returns}(s_t)$
- 2: **for** 回合数 $= 1, M$ **do**
- 3: 根据策略 π 采样一回合轨迹 $\tau = \{s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T, \}$
- 4: 初始化回报 $G \leftarrow 0$
- 5: **for** 时步 $t = T - 1, , T - 2, \dots, 0$ **do**
- 6: $G \leftarrow \gamma G + R_{t+1}$
- 7: **if** 状态 s_t 首次出现，即不在历史状态 s_0, \dots, s_{t-1} 中 **then**
- 8: 将 G 添加到 $\text{Returns}(s_t)$
- 9: $V(S_t) \leftarrow \text{average}(\text{Returns}(S_t))$
- 10: **end if**
- 11: **end for**
- 12: **end for**

图 2: 首次访问蒙特卡洛算法伪代码

假设我们已经采样得到了一条完整的轨迹 $\tau = \{S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T, S_T\}$, 其中 S_T 是终止状态。对于轨迹中的每个状态 S_t , 我们检查它是否是该状态在当前轨迹中的首次出现。如果是首次出现，我们计算从该时间步 t 开始的回报 G_t , 并将其添加到该状态的回报列表中，最后更新该状态的价值函数 $V(S_t)$ 为回报列表的平均值。

回头看我们前面的示例，可以用 FVMC 方法来实现状态价值函数的估计，如代码 2 所示。

代码 2: 首次访问蒙特卡洛方法估计状态价值函数

```
import numpy as np
from collections import defaultdict

# ----- 环境定义 -----
states = ['s1', 's2', 's3', 's4']
gamma = 0.9
R = -1
terminal = 's4'

# 状态转移（确定性）
transitions = {
    's1': {'right': 's2', 'down': 's3'},
    's2': {'down': 's4'},
    's3': {'right': 's4'},
}
}

# 策略 π: 在合法动作间随机选择
def policy(state):
    actions = list(transitions[state].keys())
    return np.random.choice(actions)

# 生成一条完整轨迹（从 s1 到 s4）
def generate_episode():
    episode = []
```

```

state = 's1'
while state != terminal:
    action = policy(state)
    next_state = transitions[state][action]
    episode.append((state, action, R))
    state = next_state
episode.append((terminal, None, 0)) # 终止
return episode

def first_visit_mc(num_episodes=1000):
    v = defaultdict(float)
    returns = defaultdict(list)

    for _ in range(num_episodes):
        episode = generate_episode()
        G = 0
        visited = set() # 用于记录首访

        # 反向遍历轨迹
        for state, action, reward in reversed(episode):
            G = gamma * G + reward
            if state not in visited:
                visited.add(state)
                returns[state].append(G)
                v[state] = np.mean(returns[state])
    return v

if __name__ == "__main__":
    v_first = first_visit_mc()
    print("First-Visit MC:")
    for s in states:
        print(f" {s}: {v_first[s]:.2f}")

```

运行结果如代码 3 所示。

代码 3: 首次访问蒙特卡洛方法估计状态价值函数结果

```

First-Visit MC:
s1: -1.90
s2: -1.00
s3: -1.00
s4: 0.00

```

可以发现，估计的状态价值函数与我们前面根据定义计算的结果是一致的。

注意，只在第一次遍历到某个状态时会记录并计算对应的回报，对应伪代码如图 2 所示。

每次访问蒙特卡洛

在 EVMC 方法中则不会忽略同一状态的多个回报，具体代码实现如代码 4 所示。

代码 4: 每次访问蒙特卡洛方法估计状态价值函数

```
import numpy as np
from collections import defaultdict

# ----- 环境定义 -----
states = ['s1', 's2', 's3', 's4']
gamma = 0.9
R = -1
terminal = 's4'

# 状态转移（确定性）
transitions = {
    's1': {'right': 's2', 'down': 's3'},
    's2': {'down': 's4'},
    's3': {'right': 's4'},
}

# 策略 π: 在合法动作间随机选择
def policy(state):
    actions = list(transitions[state].keys())
    return np.random.choice(actions)

# 生成一条完整轨迹（从 s1 到 s4）
def generate_episode():
    episode = []
    state = 's1'
    while state != terminal:
        action = policy(state)
        next_state = transitions[state][action]
        episode.append((state, action, R))
        state = next_state
    episode.append((terminal, None, 0)) # 终止
    return episode

def every_visit_mc(num_episodes=1000):
    V = defaultdict(float)
    returns = defaultdict(list)

    for _ in range(num_episodes):
        episode = generate_episode()
        G = 0

        # 反向遍历轨迹（每次出现都更新）
        for state, action, reward in reversed(episode):
            G = gamma * G + reward
            returns[state].append(G)
            V[state] = np.mean(returns[state])

    return V
```

```

    return v

if __name__ == "__main__":
    v_every = every_visit_mc()

    print("\nEvery-Visit MC:")
    for s in states:
        print(f" {s}: {v_every[s]:.2f}")

```

同样运行结果如代码 5 所示。

代码 5: 每次访问蒙特卡洛方法估计状态价值函数结果

```

Every-Visit MC:
s1: -1.90
s2: -1.00
s3: -1.00
s4: 0.00

```

总的来说，FVMC 是一种基于回合的增量式方法，具有无偏性和收敛快的优点，但是在状态空间较大的情况下，依然需要训练很多个回合才能达到稳定的结果。而 EVMC 则是更为精确的预测方法，但是计算的成本相对也更高。

蒙特卡洛估计动作价值

蒙特卡洛预测或者估计动作价值函数 $Q(s, a)$ 的方法与状态价值函数类似，只不过需要同时考虑状态和动作的组合。具体来说，蒙特卡洛动作价值估计的步骤如下：

1. **生成完整轨迹**：与状态价值函数相同，首先需要通过与环境的交互生成一条完整的轨迹，包括状态、动作和奖励。
2. **计算回报**：对于轨迹中的每个状态-动作对 (s, a) ，计算从该对开始的回报 G_t 。
3. **更新价值函数**：根据计算得到的回报更新动作价值函数 $Q(s, a)$ ，可以使用首次访问或每次访问的方式。具体的增量式更新公式与状态价值函数类似，如式 (13) 所示。

$$Q(s, a) \leftarrow Q(s, a) + \alpha[G - Q(s, a)] \quad (13)$$

其中 α 是学习率， G 是从状态 s 执行动作 a 后得到的回报。

使用 Python 代码实现蒙特卡洛动作价值估计来解决前面示例的问题，如代码 6 所示。

代码 6: 蒙特卡洛方法估计动作价值函数

```

import numpy as np
from collections import defaultdict

states = ['s1', 's2', 's3', 's4']
actions = ['right', 'down']
gamma = 0.9
R = -1
terminal = 's4'

# 转移定义

```

```

transitions = {
    ('s1', 'right'): 's2',
    ('s1', 'down'): 's3',
    ('s2', 'down'): 's4',
    ('s3', 'right'): 's4',
}

def policy(state):
    # 随机策略  $\pi(a|s)$ 
    legal = [a for (s,a) in transitions if s == state]
    return np.random.choice(legal)

def generate_episode():
    episode = []
    state = 's1'
    while state != terminal:
        action = policy(state)
        next_state = transitions[(state, action)]
        episode.append((state, action, R))
        state = next_state
    episode.append((terminal, None, 0))
    return episode

def mc_action_value(num_episodes=1000):
    Q = defaultdict(float)
    returns = defaultdict(list)

    for _ in range(num_episodes):
        episode = generate_episode()
        G = 0
        visited = set()
        for state, action, reward in reversed(episode):
            G = gamma * G + reward
            if action is not None and (state, action) not in visited:
                visited.add((state, action))
                returns[(state, action)].append(G)
                Q[(state, action)] = np.mean(returns[(state, action)])
    return Q

Q = mc_action_value()
for (s,a), v in Q.items():
    print(f"Q({s},{a}) = {v:.2f}")

```

运行结果如代码 7 所示。

代码 7: 蒙特卡洛方法估计动作价值函数结果

```

Q(s1,down) = -1.90
Q(s1,right) = -1.90
Q(s2,down) = -1.00
Q(s3,right) = -1.00

```

联系状态价值和动作价值的关系，如式 (14) 所示。

$$V_{\pi}(s) = \sum_a \pi(a|s) Q_{\pi}(s, a) \quad (14)$$

以状态 s_1 为例, 由于智能体采用随机策略, 即在动作 a_1 和 a_2 之间按同等概率选择, 因此可以计算出 $V(s_1)$ 如式(15)所示。

$$\begin{aligned} V(s_1) &= 0.5 * Q(s_1, a_1) + 0.5 * Q(s_1, a_2) \\ &= 0.5 * (-1.9) + 0.5 * (-1.9) \\ &= -1.9 \end{aligned} \quad (15)$$

可以发现, 计算结果与前面直接估计的状态价值是一致的。

蒙特卡洛控制

更复杂的蒙特卡洛预测示例

为了更好地理解蒙特卡洛控制方法, 先举一个稍微更复杂的蒙特卡洛预测示例 (虽然并没有复杂多少, 因为状态空间上仍然较小)。

如图 3 所示, 在前面 2×2 网格示例的基础上, 考虑智能体在 3×3 的网格中使用随机策略进行移动, 以左上角为起点, 右下角为终点, 同样规定每次只能向右或向下移动, 动作分别用 a_1 和 a_2 表示。用智能体的位置不同的状态, 即 s_1, s_2, \dots, s_9 , 初始状态为 $S_0 = s_1$, 终止状态为 s_9 。



图 3: 3×3 网格示例

除了每走一步接收 -1 的奖励之外, 这次我们在网格中增加了一些障碍物, 例如在位置 s_4 处设置了一个深坑, 智能体走到该位置时会受到一个额外的负奖励 -3 , 在位置 s_5 处设置了一个水洼, 智能体走到该位置时会受到一个额外的负奖励 -0.5 。折扣因子 $\gamma = 0.9$, 目标是计算各个状态的价值函数 $V(s)$ 。

用 Python 代码实现蒙特卡洛方法来估计状态价值函数, 如代码 8 所示。

代码 8: 使用蒙特卡洛方法估计 3×3 网格状态价值函数

```
import random
from collections import defaultdict
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ----- 参数 -----
gamma = 0.9
episodes = 30000
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
start = "s1"
```

```

coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2)
}

# ----- 环境 -----
def legal_actions(s):
    r, c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")
    return acts

def step(s, a):
    r, c = coords[s]
    if a == "right": r2, c2 = r, c + 1
    elif a == "down": r2, c2 = r + 1, c
    s2 = [k for k, v in coords.items() if v == (r2, c2)][0]
    reward = -1.0
    # 额外惩罚修改 ↓↓↓
    if s2 == "s4": reward -= 3.0      # 深坑
    if s2 == "s5": reward -= 0.5      # 水洼
    if s2 == "s9": reward += 1.0      # 终点 +1 → 净0
    done = (s2 == terminal)
    return s2, reward, done

def policy(s):
    acts = legal_actions(s)
    return random.choice(acts)

def generate_episode():
    episode = []
    s = start
    while True:
        if s == terminal:
            episode.append((s, None, 0))
            break
        a = policy(s)
        s2, r, done = step(s, a)
        episode.append((s, a, r))
        s = s2
    return episode

# ----- First-Visit Monte Carlo -----
def first_visit_mc(num_episodes=episodes, gamma=0.9):
    V = defaultdict(float)
    N = defaultdict(int)
    for _ in range(num_episodes):
        episode = generate_episode()
        G = 0
        visited = set()
        for s, a, r in reversed(episode):

```

```

G = gamma * G + r
if s not in visited:
    visited.add(s)
    N[s] += 1
    V[s] += (G - V[s]) / N[s]

return V

V = first_visit_mc()

# ----- 打印结果 -----
grid = np.array([[V[f"s{r*3+c+1}"] for c in range(3)] for r in range(3)])
df = pd.DataFrame(grid.round(3),
                  columns=["col1", "col2", "col3"],
                  index=["row1", "row2", "row3"])
print(df)

# ----- 热力图展示 -----
plt.figure(figsize=(5,5))
plt.imshow(grid, cmap='coolwarm', origin='upper')
for i in range(3):
    for j in range(3):
        plt.text(j, i, f"{grid[i,j]:.2f}", ha='center', va='center', color='black')
plt.title("Monte Carlo State Values")
plt.colorbar(label="V(s)")
plt.tight_layout()
plt.show()

```

执行代码后，除了打印出各状态的价值函数结果外，还会生成一个热力图来直观展示各状态的价值分布情况，如图 4 所示。

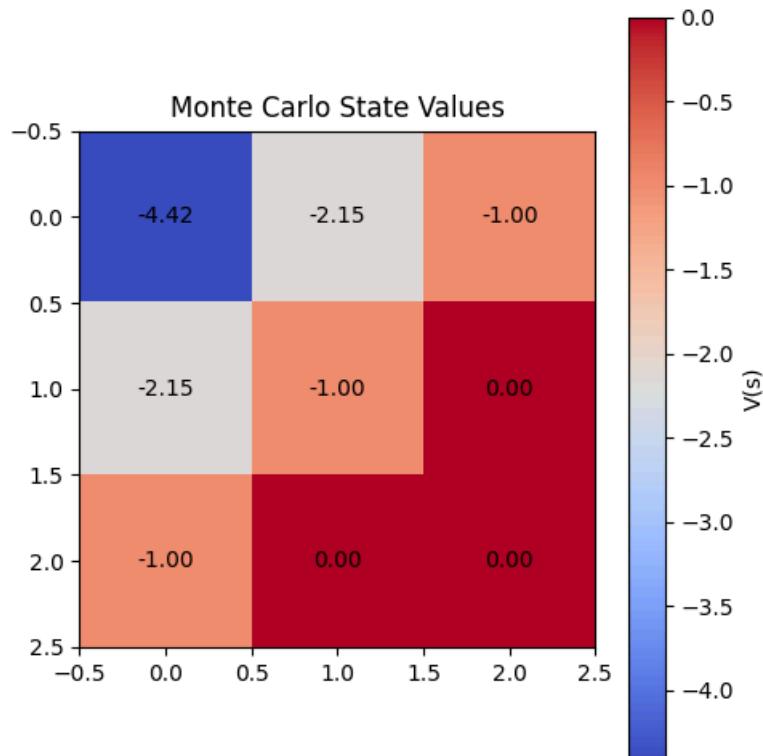


图 4: 3x3 网格中的状态价值热力图

初学者可能会有疑问，为什么平地状态例如 s_1 反而比有障碍物的状态 s_4 和 s_5 价值更低呢？直觉上，我们可能会认为障碍物的存在会降低对应位置的状态价值，但实际上注意状态价值的计算是基于从该状态出发的未来回报的期望，而不是仅仅考虑该状态本身的奖励。

由于智能体采用的是随机策略，且动作只能向右或向下移动，因此从起始状态 s_1 出发，智能体有较高的概率会经过障碍物位置 s_4 和 s_5 ，从而导致整体回报降低，因此 s_1 的状态价值较低。

而对于障碍物位置 s_4 和 s_5 ，虽然从别的状态走到它们时身会带来额外的负奖励，但从这些状态出发，例如从 s_5 出发，只能到达平地状态 s_6 、 s_8 和终点 s_9 ，这些状态的未来回报相对较高，因此 s_4 和 s_5 的状态价值反而可能高于一些平地位置。

同样地，可以用蒙特卡洛方法来估计动作价值函数 $Q(s, a)$ ，代码实现如代码 9 所示。

代码 9: 使用蒙特卡洛方法估计 3x3 网格动作价值函数

```
import random
from collections import defaultdict
import numpy as np
import matplotlib.pyplot as plt

# ----- 参数 -----
gamma = 0.9
episodes = 30000
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
start = "s1"

coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2)
}

# ----- 环境 -----
def legal_actions(s):
    r, c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")
    return acts

def step(s, a):
    r, c = coords[s]
    if a == "right": r2, c2 = r, c + 1
    elif a == "down": r2, c2 = r + 1, c
    s2 = [k for k, v in coords.items() if v == (r2, c2)][0]
    reward = -1.0
    if s2 == "s4": reward -= 3.0      # 深坑
    if s2 == "s5": reward -= 0.5      # 水洼
    if s2 == "s9": reward += 1.0      # 终点奖励 +1 → 净0
    done = (s2 == terminal)
    return s2, reward, done
```

```

def policy(s):
    acts = legal_actions(s)
    return random.choice(acts)

def generate_episode():
    episode = []
    s = start
    while True:
        if s == terminal:
            episode.append((s, None, 0))
            break
        a = policy(s)
        s2, r, done = step(s, a)
        episode.append((s, a, r))
        s = s2
    return episode

def first_visit_mc_Q(num_episodes=episodes, gamma=0.9):
    Q = defaultdict(float)
    N = defaultdict(int)
    for _ in range(num_episodes):
        episode = generate_episode()
        G = 0
        visited = set()
        for s, a, r in reversed(episode):
            G = gamma * G + r
            if a is None:
                continue
            if (s, a) not in visited:
                visited.add((s, a))
                N[(s, a)] += 1
                Q[(s, a)] += (G - Q[(s, a)]) / N[(s, a)]
    return Q

Q = first_visit_mc_Q()

# 构建Q值矩阵
q_right = np.full((3, 3), np.nan)
q_down = np.full((3, 3), np.nan)

for s in states:
    if "right" in legal_actions(s):
        r, c = coords[s]
        q_right[r, c] = Q[(s, "right")]
    if "down" in legal_actions(s):
        r, c = coords[s]
        q_down[r, c] = Q[(s, "down")]

for r in range(3):
    for c in range(3):
        s = [k for k, v in coords.items() if v == (r, c)][0]
        print(f"State {s}: Q(s, 'right') = {q_right[r,c]:.2f}, Q(s, 'down') = {q_down[r,c]:.2f}")

```

```

# 可视化 Q(s,a) 热力图
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

im1 = axes[0].imshow(q_right, cmap='coolwarm', origin='upper')
axes[0].set_title("Q(s1, a='right')")
for i in range(3):
    for j in range(3):
        if not np.isnan(q_right[i, j]):
            axes[0].text(j, i, f"{q_right[i, j]:.2f}", ha='center', va='center',
color='black')
fig.colorbar(im1, ax=axes[0])

im2 = axes[1].imshow(q_down, cmap='coolwarm', origin='upper')
axes[1].set_title("Q(s1, a='down')")
for i in range(3):
    for j in range(3):
        if not np.isnan(q_down[i, j]):
            axes[1].text(j, i, f"{q_down[i, j]:.2f}", ha='center', va='center',
color='black')
fig.colorbar(im2, ax=axes[1])

plt.suptitle("Monte Carlo Action Values")
plt.tight_layout()
plt.show()

```

执行代码后，可以得到各状态-动作对的动作价值函数结果，并生成两个热力图来直观展示各状态在不同动作下的价值分布情况，如图 5 所示。

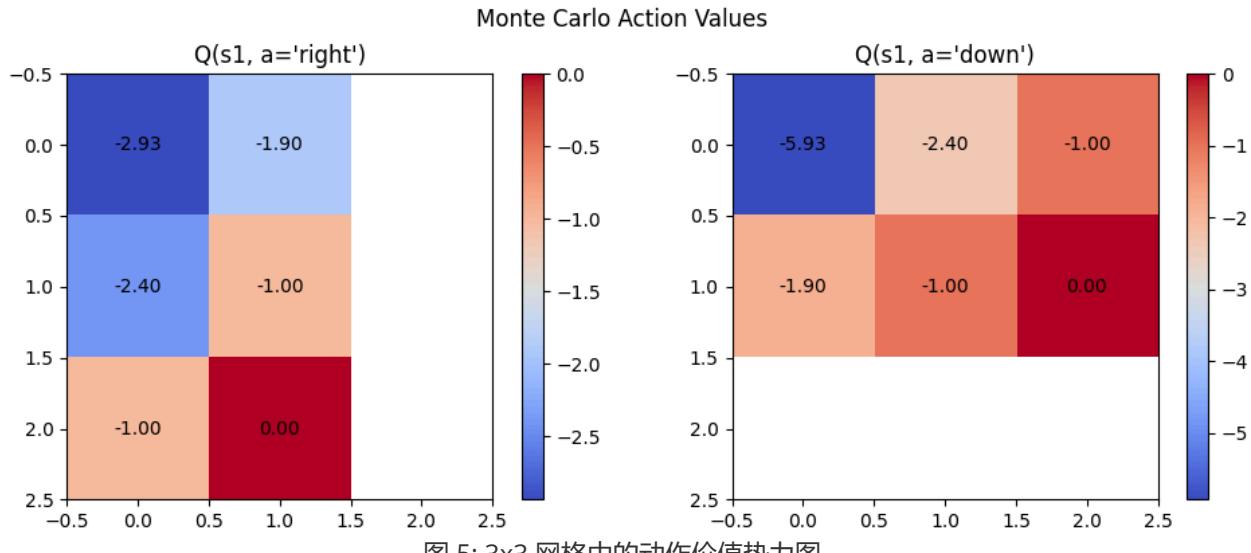


图 5: 3x3 网格中的动作价值热力图

可以发现，从状态 s_1 出发，选择向右移动（动作 a_1 ）和向下移动（动作 a_2 ）的动作价值分别为 $Q(s_1, a_1) = -2.93$ 和 $Q(s_1, a_2) = -5.93$ ，这表明在随机策略下，选择向右移动的动作相对更优一些，因为对应的动作价值更高。

蒙特卡洛控制算法

在计算动作价值的过程中，可以借鉴动态规划中策略迭代的思想，先进行策略评估，即预测动作价值，然后基于当前的动作价值函数进行策略改进，形成一个新的策略，如式 (16) 所示。

$$\pi_0 \rightarrow Q_{\pi_0} \rightarrow \pi_1 \rightarrow Q_{\pi_1} \rightarrow \pi_2 \rightarrow \dots \rightarrow \pi^* \rightarrow Q_{\pi^*} \quad (16)$$

这种利用蒙特卡洛方法交替进行策略评估和策略改进的过程，称为蒙特卡洛控制 (Monte Carlo Control)，算法流程如图 6 所示。

蒙特卡洛控制算法

```
1: 初始化：策略  $\pi(s)$ , 动作价值  $Q(s, a)$ , 回报字典  $\text{Returns}(s, a)$ 
2: for 回合数 = 1,  $M$  do
3:   根据当前策略  $\pi$  采样轨迹  $\tau = \{s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T, \}$ 
4:   初始化回报  $G \leftarrow 0$ 
5:   for 时步  $t = T - 1, , T - 2, \dots, 0$  do
6:      $G \leftarrow \gamma G + R_{t+1}$ 
7:     if 状态  $s_t$  首次出现，即不在历史状态  $s_0, \dots, s_{t-1}$  中 then
8:       将  $G$  添加到  $\text{Returns}(s_t, a_t)$ 
9:        $Q(S_t, A_t) \leftarrow \text{average}(\text{Returns}(S_t, A_t))$ 
10:      更新策略： $\pi(s_t) \leftarrow \arg \max_a Q(s_t, a)$ 
11:    end if
12:   end for
13: end for
```

图 6: 蒙特卡洛控制算法伪代码

在 3×3 网格的示例中，可以用蒙特卡洛控制方法来学习一个更优的策略，代码实现如代码 10 所示。

代码 10: 使用蒙特卡洛控制方法学习 3×3 网格最优策略

```
import random
from collections import defaultdict
import numpy as np
import matplotlib.pyplot as plt

# ----- 环境定义 -----
gamma = 0.9
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"

coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2)
}

def legal_actions(s):
    r, c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")
    if (r, c) != (2, 2): acts.append("up")
    if c > 0: acts.append("left")
    return acts
```

```

if r < 2: acts.append("down")
return acts

def step(s, a):
    r, c = coords[s]
    if a == "right": r2, c2 = r, c + 1
    elif a == "down": r2, c2 = r + 1, c
    s2 = [k for k, v in coords.items() if v == (r2, c2)][0]

    reward = -1.0
    if s2 == "s4": reward -= 3.0
    if s2 == "s5": reward -= 0.5
    if s2 == "s9": reward += 1.0
    done = (s2 == terminal)
    return s2, reward, done

# ----- Monte Carlo ES 控制 -----
def monte_carlo_es(num_episodes=10000, gamma=0.9):
    Q = defaultdict(float)
    Returns = defaultdict(list)
    policy = {}

    # 初始化策略 π 随机
    for s in states:
        actions = legal_actions(s)
        if actions:
            policy[s] = random.choice(actions)

    for episode in range(num_episodes):
        # Exploring Starts: 随机选起点 (s0, a0)
        s0 = random.choice([s for s in states if s != terminal])
        actions = legal_actions(s0)
        if not actions:
            continue
        a0 = random.choice(actions)

        # 生成一条完整轨迹
        episode_list = []
        s, a = s0, a0
        while True:
            s_next, r, done = step(s, a)
            episode_list.append((s, a, r))
            if done:
                break
            a = policy.get(s_next, random.choice(legal_actions(s_next)))
            s = s_next

        # 反向计算 G 并更新 Q
        G = 0
        visited = set()
        for s, a, r in reversed(episode_list):
            G = gamma * G + r
            if (s, a) not in visited:

```

```

        visited.add((s, a))
        Returns[(s,a)].append(G)
        Q[(s,a)] = np.mean>Returns[(s,a)])
        # 策略改进: 贪婪选择
        policy[s] = max(legal_actions(s), key=lambda x: Q[(s,x)])
    return policy, Q

policy, Q = monte_carlo_es()

# ----- 输出结果 -----
print("最优策略 π(s):")
for s in states:
    if s in policy:
        print(f"{s}: {policy[s]}")
print("\n示例部分 Q 值:")
for k in list(Q.keys())[:10]:
    print(k, "=", round(Q[k], 2))

import matplotlib.pyplot as plt

# ----- 绘制最优策略箭头图 (从 s1 出发) -----
fig, ax = plt.subplots(figsize=(5, 5))
ax.set_xlim(-0.5, 2.5)
ax.set_ylim(-0.5, 2.5)
ax.set_xticks(range(3))
ax.set_yticks(range(3))
ax.set_xticklabels(["col1", "col2", "col3"])
ax.set_yticklabels(["row1", "row2", "row3"])
ax.set_title("Optimal Policy Arrows (from s1)")

# 画方格背景 (起点/终点/水洼/深坑)
colors = {"s1": "#b2df8a", "s4": "#999999", "s5": "#80b1d3", "s9": "#fdb462"}
for s, (r, c) in coords.items():
    rect = plt.Rectangle((c-0.5, r-0.5), 1, 1,
                         facecolor=colors.get(s, "white"),
                         edgecolor='black', lw=1.5)
    ax.add_patch(rect)
    ax.text(c, r, s, ha='center', va='center', fontsize=12, fontweight='bold')

import matplotlib.pyplot as plt

# ----- 构造从 s1 出发的最优路径 -----
path = ["s1"]
s = "s1"
while s != "s9":
    a = policy[s]
    r, c = coords[s]
    if a == "right":
        s_next = [k for k, v in coords.items() if v == (r, c+1)][0]
    else: # down
        s_next = [k for k, v in coords.items() if v == (r+1, c)][0]
    path.append(s_next)
    s = s_next

```

```

# ----- 准备通用绘图函数 -----
def draw_policy(ax, highlight_path=False):
    ax.set_xlim(-0.5, 2.5)
    ax.set_ylim(-0.5, 2.5)
    ax.set_xticks(range(3))
    ax.set_yticks(range(3))
    ax.set_xticklabels(["col1", "col2", "col3"])
    ax.set_yticklabels(["row1", "row2", "row3"])
    ax.grid(True)

    # 背景方格: 起点、终点、水洼、深坑
    colors = {"s1": "#b2df8a", "s4": "#999999", "s5": "#80b1d3", "s9": "#fdb462"}
    for s, (r, c) in coords.items():
        rect = plt.Rectangle((c-0.5, r-0.5), 1, 1,
                             facecolor=colors.get(s, "white"),
                             edgecolor='black', lw=1.5)
        ax.add_patch(rect)
        ax.text(c, r, s, ha='center', va='center', fontsize=12, fontweight='bold')

    # 策略箭头 (黑色)
    for s, a in policy.items():
        if s == "s9": # 终点不画箭头
            continue
        r, c = coords[s]
        dx, dy = 0, 0
        if a == "right": dx = 0.4
        elif a == "down": dy = 0.4
        ax.arrow(c, r, dx, dy, head_width=0.15, head_length=0.15, fc='black', ec='black')

    # 若 highlight_path=True, 则绘制红色最优路径
    if highlight_path:
        for i in range(len(path)-1):
            s1, s2 = path[i], path[i+1]
            r1, c1 = coords[s1]
            r2, c2 = coords[s2]
            ax.arrow(c1, r1, c2-c1, r2-r1, head_width=0.18, head_length=0.18,
                     fc='red', ec='red', lw=2)

    ax.invert_yaxis()

print("最优路径: ", " → ".join(path))
# ----- 绘制对比图 -----
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
draw_policy(axes[0], highlight_path=False)
axes[0].set_title("Global Optimal Policy π(s)")

draw_policy(axes[1], highlight_path=True)
axes[1].set_title("Trajectory from s1 (Red Arrows)")

plt.suptitle("Monte Carlo ES Policy vs Optimal Path", fontsize=14)
plt.tight_layout()
plt.show()

```

执行代码后，可以得到最优策略 $\pi(s)$ 以及从起始状态 s_1 出发的最优路径，并生成对比图来展示全局最优策略和从 s_1 出发的最优路径，如图 7 所示。

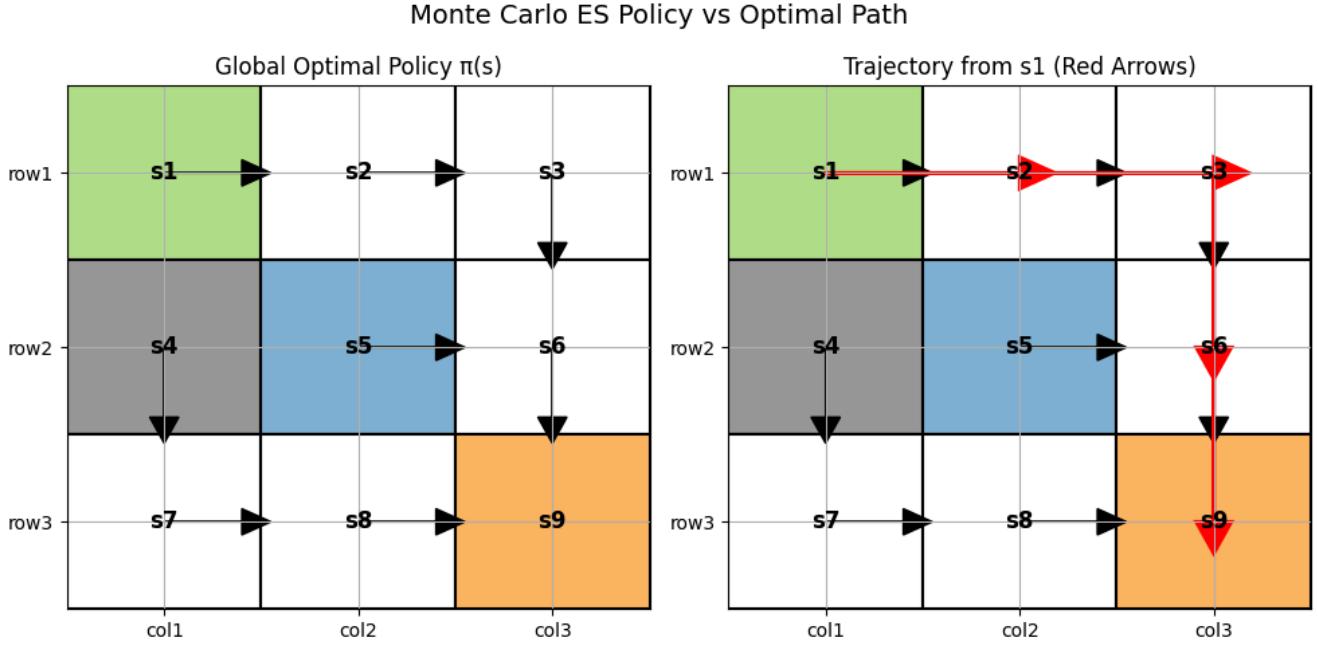


图 7: 3x3 网格中的蒙特卡洛控制最优策略与路径

左图展示了不同状态下的全局最优策略，右图用红色箭头高亮显示了从起始状态 s_1 出发的最优路径，如式 (17) 所示。

$$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_6 \rightarrow s_9 \quad (17)$$

可以看到，智能体在最优路径上避开了障碍物 s_4 和 s_5 ，选择了一条累计回报较高的路径到达终点 s_9 ，这验证了蒙特卡洛控制方法在学习最优策略方面的有效性。

时序差分方法

时序差分估计

回顾蒙特卡洛估计的更新，如式 (1) 所示。

$$V(s) \leftarrow V(s) + \alpha[G - V(s)] \quad (1)$$

其中回报 G 是从状态 s 开始一直到终止状态的完整回报，定义如式 (2) 所示。

$$G = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \quad (2)$$

尽管蒙特卡洛方法可以通过增量式更新来迭代预测状态价值函数 $V(s)$ ，但它有一个明显的缺点，即它必须等到回合结束后才能进行更新，因为只有在终止状态时才能计算出完整的回报 G 。这在某些情况下可能会导致学习过程变得缓慢，尤其是在回合较长或没有明确终止状态的环境中。

为了解决这个问题，时序差分 (Temporal Difference, TD) 估计方法被提出。时序差分方法结合了蒙特卡洛方法和动态规划的思想，允许在每个时间步进行更新，而不需要等待回合结束。具体来说，时序差分方法使用当前奖励和下一个状态的估计值来更新当前状态的价值估计，如式 (3) 所示。

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad (3)$$

其中， $R_{t+1} + \gamma V(s_{t+1})$ 被称为**时序差分目标**，它表示的是当前回报 G 的一个估计值。

当前估计与目标之间的差值则被称为**时序差分误差** (TD Error)，如式 (4) 所示。

$$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \quad (4)$$

这种使用当前估计来更新现有估计的方式被称为**自举** (bootstrapping)，这样做的好处是可以在每个时间步进行更新，不用等到回合结束拿到完整的回报 G 再更新。这样做的好处就是一方面结合了蒙特卡洛的无模型特性，即不需要知道环境的状态转移概率，另一方面也结合了动态规划的自举特性，即利用现有的估计来更新估计，从而提高了学习效率。

注意，实际应用中会考虑终止状态的特殊情况，由于终止状态没有下一个状态，因此在更新时需要单独处理，通常将终止状态的估计值设为 0，如式 (5) 所示。

$$\begin{cases} V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} - V(s_t)] & \text{对于终止状态 } V(s_t) \\ V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)] & \text{对于非终止状态 } V(s_t) \end{cases} \quad (5)$$

时序差分估计计算示例

回顾蒙特卡洛方法中讲到的 3×3 网格世界的例子，如图 1 所示。考虑智能体在 3×3 的网格中使用随机策略进行移动，以左上角为起点，右下角为终点，同样规定每次只能向右或向下移动，动作分别用 a_1 和 a_2 表示。用智能体的位置不同的状态，即 s_1, s_2, \dots, s_9 ，初始状态为 $S_0 = s_1$ ，终止状态为 s_9 。

s_1	s_2	s_3
s_4	s_5	s_6
s_7	s_8	s_9

	起点	a_1 向右	a_2 向下
	终点	$R(s_i) = -1$	
	水洼	$R(s_4) = R(s_i) - 3.0$	
	深坑	$R(s_5) = R(s_i) - 0.5$	
			$R(s_9) = R(s_i) + 1$

图 1: 3×3 网格示例

除了每走一步接收 -1 的奖励之外，这次我们在网格中增加了一些障碍物，例如在位置 s_4 处设置了一个深坑，智能体走到该位置时会受到一个额外的负奖励 -3 ，在位置 s_5 处设置了一个水洼，智能体走到该位置时会受到一个额外的负奖励 -0.5 。折扣因子 $\gamma = 0.9$ ，目标是计算各个状态的价值函数 $V(s)$ 。

用 Python 实现时序差分估计的代码并且对比蒙特卡洛估计的结果，如代码 1 所示。

代码 1：时序差分估计与蒙特卡洛估计对比

```
import random
from collections import defaultdict
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

# ----- 参数 -----
gamma = 0.9
alpha = 0.1           # 学习率（可试 0.05~0.2）
episodes = 100000
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
start = "s1"

coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2)
}

# ----- 环境 -----
def legal_actions(s):
    r, c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")
    return acts

def step(s, a):
    r, c = coords[s]
    if a == "right": r2, c2 = r, c + 1
    elif a == "down": r2, c2 = r + 1, c
    s2 = [k for k, v in coords.items() if v == (r2, c2)][0]
    reward = -1.0
    # 额外惩罚修改 ↓↓
    if s2 == "s4": reward -= 3.0      # 深坑
    if s2 == "s5": reward -= 0.5      # 水洼
    if s2 == "s9": reward += 1.0      # 终点 +1 → 净0
    done = (s2 == terminal)
    return s2, reward, done

def policy(s):
```

```

acts = legal_actions(s)
return random.choice(acts)

def generate_episode():
    episode = []
    s = start
    while True:
        if s == terminal:
            episode.append((s, None, 0))
            break
        a = policy(s)
        s2, r, done = step(s, a)
        episode.append((s, a, r))
        s = s2
    return episode

# ----- First-Visit Monte Carlo -----
def first_visit_mc(num_episodes=episodes, gamma=0.9):
    V = defaultdict(float)
    N = defaultdict(int)
    for _ in range(num_episodes):
        episode = generate_episode()
        G = 0
        visited = set()
        for s, a, r in reversed(episode):
            G = gamma * G + r
            if s not in visited:
                visited.add(s)
                N[s] += 1
                V[s] += (G - V[s]) / N[s]
    return V

# ----- TD(0) 预测 -----
def td0_value_prediction(episodes=episodes, alpha=alpha, gamma=gamma, start="s1"):
    V = defaultdict(float) # 初始全0
    for _ in range(episodes):
        s = start
        while s != terminal:
            a = policy(s)
            s2, r, done = step(s, a)
            # TD(0) 更新
            V[s] += alpha * (r + gamma * V[s2] - V[s])
            s = s2
    return V

s_t = time.time()
V_mc = first_visit_mc()
t_cost_mc = time.time() - s_t
print(f"First-Visit MC 用时: {t_cost_mc:.2f} 秒")

# ----- 打印结果 -----
grid_mc = np.array([[V_mc[f"s{r*3+c+1}"] for c in range(3)] for r in range(3)])
df_mc = pd.DataFrame(grid_mc.round(3),

```

```

        columns=["col1","col2","col3"],
        index=["row1","row2","row3"])

print(df_mc)

s_t = time.time()
v_td0 = td0_value_prediction()
t_cost_td0 = time.time() - s_t

print(f"TD(0) 用时: {t_cost_td0:.2f} 秒")

grid_td0 = np.array([[v_td0[f"s{r*3+c+1}"] for c in range(3)] for r in range(3)])
df_td0 = pd.DataFrame(grid_td0.round(3),
                      columns=["col1","col2","col3"],
                      index=["row1","row2","row3"])

print(df_td0)

```

运行代码后，得到结果如代码 2 所示。

代码 2: 时序差分估计与蒙特卡洛估计结果对比

```

First-Visit MC 用时: 0.57 秒
    col1   col2   col3
row1 -4.43 -2.149 -1.0
row2 -2.15 -1.000  0.0
row3 -1.00  0.000  0.0
TD(0) 用时: 0.47 秒
    col1   col2   col3
row1 -4.110 -2.216 -1.0
row2 -2.182 -1.000  0.0
row3 -1.000  0.000  0.0

```

可以发现，时序差分估计和蒙特卡洛估计得到的状态价值函数 $V(s)$ 非常接近，但时序差分估计的计算速度更快一些。这是因为时序差分方法能够在每个时间步进行更新，而不需要等待整个回合结束，从而提高了学习效率，并且在更加复杂的环境中，这种优势会更加明显。

时序差分目标的推导

那么时序差分目标 $R_{t+1} + \gamma V(s_{t+1})$ 是怎么定义来的呢？回顾贝尔曼方程的状态价值函数形式，如式 (6) 所示。

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi [G_t \mid S_t = s] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s] \end{aligned} \quad (6)$$

会发现，在给定策略 π 的期望公式中，回报和状态价值存在等价关系，如式 (7) 所示。

$$G_t \approx R_{t+1} + \gamma V(s_{t+1}) \quad (7)$$

代入到蒙特卡洛更新公式 (1) 中，就得到了时序差分更新公式 (3)。

注意，(7) 中的等价近似是有条件的，需要满足在“期望”的环境下。具体来说，首先需保证策略是稳定的（通常指策略能够收敛，感兴趣的读者也可参考后面要讲的策略梯度方法中的平稳分布概念），其次需要足够的探索，产生足够多样的轨迹，尽可能覆盖所有的回报情况。然后在此基础上通过不断地迭代更新 $V(s)$ ，使得 $V(s)$ 趋近于真实的状态价值 $V_\pi(s)$ ，在这个过程中，时序差分目标 $R_{t+1} + \gamma V(s_{t+1})$ 也会趋近于真实的期望回报 $\mathbb{E}_\pi[G_t \mid S_t = s]$ 。

换句话说，在保证策略收敛的前提下，使用时序差分方法时，最重要的是**足够多次的迭代更新和足够的探索性**，否则时序差分估计可能会一直不稳定，从而无法收敛到最优策略，这是在实际应用中需要特别注意的地方。

在策略收敛前，由于状态价值函数 $V(s)$ 还没有收敛到真实的值，是一个估计的量，因此时序差分目标 $R_{t+1} + \gamma V(s_{t+1})$ 也是一个近似的估计值。因此，在实际应用中，为了保证前期的估计更加稳定，会采用一些技巧，例如使用较小的学习率 α 、提高探索率等，来减小估计误差对学习过程的影响。很多时候，也会优化时序差分目标本身的表示形式，例如使用 $R_{t+1} + \gamma V(s_{t+1})$ 形式的变种，或者引入神经网络等函数逼近方法来提高估计的准确性。

n 步时序差分

式(7)中的时序差分目标 $R_{t+1} + \gamma V(s_{t+1})$ 实际上是向前自举了一步的结果，即单步时序差分(TD(0))。实际上我们也可以向前自举多步，得到n步时序差分目标，如式(8)所示。

$$G_t^{(n)} = \underbrace{R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n}}_{\text{实际回报 (采样得到)}} + \underbrace{\gamma^n V(s_{t+n})}_{\text{估计值 (引入自举)}} \quad (8)$$

可以看出该等价近似式中，前半部分是从时间步 t 开始，连续 n 步的实际奖励回报的折扣和，这部分是通过采样得到的实际值；后半部分则是从时间步 $t + n$ 开始的状态 s_{t+n} 的估计价值 $V(s_{t+n})$ ，这部分是通过自举引入的估计值。当 n 越大时， γ^{n-1} 的权重会越来越小，意味着估计值 $V(s_{t+n})$ 对整体目标的影响会减小，而实际回报部分的影响会增大，但同时计算复杂度也会增加；反之，当 n 越小时，估计值 $V(s_{t+n})$ 的影响会增大，而实际回报部分的影响会减小，但计算复杂度会降低。

如式(9)所示，当 n 趋近于无穷大时， n 步时序差分目标就变成了完整的回报 G ，即蒙特卡洛方法；当 $n = 1$ 时，则退化为单步时序差分目标

$$\begin{aligned} n = 1(\text{TD}) \quad G_t^{(1)} &= R_{t+1} + \gamma V(s_{t+1}) \\ n = 2 \quad G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 V(s_{t+2}) \\ n = \infty(\text{MC}) \quad G_t^{\infty} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T \end{aligned} \quad (9)$$

因此， n 步时序差分方法实际上是蒙特卡洛方法和单步时序差分方法之间的一种折中，通过调整 n 的取值，一方面在计算复杂度和估计准确性之间进行权衡，另一方面也在偏差和方差之间进行权衡。较小的 n 值通常会导致较高的偏差但较低的方差，而较大的 n 值则会导致较低的偏差但较高的方差。

动态规划、蒙特卡洛和时序差分的比较

回顾动态规划、蒙特卡洛和时序差分这三种算法，它们都是强化学习中比较重要的价值估计方法，并且各自有不同的特点和适用场景，具体比如表1所示。

表 1: 动态规划、蒙特卡洛和时序差分比较

方法	依赖环境模型	更新时机	估计方式	优缺点
动态规划	需要	每个时间步	自举	优点：收敛速度快；缺点：有模型方法，需要完整的环境模型
蒙特卡洛方法	不需要	回合结束后	采样	优点：无偏估计，无模型方法，适用于未知环境；缺点：需要完整回合，高方差，收敛速度慢
时序差分方法	不需要	每个时间步	自举和采样结合	优点：无模型方法，适用于未知环境，更新及时高效，低方差；缺点：有偏估计，可能不稳定，依赖于探索策略

Sarsa 算法

Sarsa 是一种基于时序差分的同策略 (on-policy) 控制算法，用于估计动作价值函数 $Q(s, a)$ ，它的名称来源于更新过程中涉及的五个元素：状态 S_t 、动作 A_t 、奖励 R_{t+1} 、下一个状态 S_{t+1} 和下一个动作 A_{t+1} 。

时序差分估计动作价值

在讲解 Sarsa 算法流程之前，我们先看时序差分如何估计动作价值。类似于状态价值，动作价值的时序差分更新如式 (10) 所示。

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (10)$$

Sarsa 算法流程

参考蒙特卡洛控制算法的流程，结合时序差分估计动作价值的方式，Sarsa 算法的具体流程如图 2 所示，[完整的代码实现](#)可参考实战部分的内容。

Sarsa 算法

- 1: 初始化 $Q(s, a)$ 为任意值，但对于终止状态即 $Q(s_T,) = 0$
- 2: **for** 回合数 $= 1, M$ **do**
- 3: 重置环境，获得初始状态 s_0
- 4: 根据 ε -greedy 策略采样初始动作 a_0
- 5: **for** 时步 $t = 1, \dots, T$ **do**
- 6: 环境根据 a_t 反馈奖励 r_t 和下一个状态 s_{t+1}
- 7: 根据 ε -greedy 策略 s_{t+1} 和采样动作 a_{t+1}
- 8: **更新策略：**
- 9: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
- 10: 更新状态 $s_{t+1} \leftarrow s_t$
- 11: 更新动作 $a_{t+1} \leftarrow a_t$
- 12: **end for**
- 13: **end for**

图 2: Sarsa 算法流程

可以看出，由于时序差分方法是在每个时间步进行更新的，因此 Sarsa 算法无需等待回合结束，而是在每个时间步根据当前状态 s_t 和动作 a_t ，以及环境反馈的奖励 r_t 和下一个状态 s_{t+1} ，并更新动作价值函数 $Q(s_t, a_t)$ 。

探索策略

前面讲到，为了保证时序差分估计的收敛性，最好进行足够的探索。在 Sarsa 算法中，通常使用 ε -贪婪策略 (ε -greedy policy) 来平衡探索和利用。

具体来说，在每个时间步，智能体以概率 $1 - \varepsilon$ 选择当前估计最优的动作（即具有最高 Q 值的动作），以概率 ε 随机选择一个动作进行探索，从而确保在学习过程中能够覆盖更多的状态和动作，具体如式 (11) 所示。

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}, & \text{if } a = \arg \max_{a'} Q(s, a') \\ \frac{\varepsilon}{|\mathcal{A}(s)|}, & \text{otherwise} \end{cases} \quad (11)$$

其中 ε 表示探索率，随着训练的进行，通常会逐渐减小 ε 的值，以便在初期进行更多的探索，而在后期更多地利用已经学到的知识，从而提高学习效率和最终的策略表现。

Q-learning 算法

Q-learning 是一种基于时序差分的异策略 (off-policy) 控制算法，。与 Sarsa 不同，Q-learning 在更新过程中使用的是下一个状态 s_{t+1} 的最优动作的估计值，而不是实际采取的动作 a_{t+1} ，这使得 Q-learning 能够学习到最优策略，而不依赖于当前的行为策略。具体的更新公式如式 (12) 所示。

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \quad (12)$$

类似的，算法流程如图 3 所示，完整的代码实现可参考实战部分的内容。

Q-learning 算法

- 1: 初始化 $Q(s, a)$ 为任意值，但对于终止状态即 $Q(s_T,) = 0$
- 2: **for** 回合数 $= 1, M$ **do**
- 3: 重置环境，获得初始状态 s_1
- 4: **for** 时步 $t = 1, \dots, T$ **do**
- 5: 根据 $\epsilon - greedy$ 策略采样动作 a_t
- 6: 环境根据 a_t 反馈奖励 r_t 和下一个状态 s_{t+1}
- 7: **更新策略：**
- 8: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
- 9: 更新状态 $s_{t+1} \leftarrow s_t$
- 10: **end for**
- 11: **end for**

图 3: Q-learning 算法流程

对比 Sarsa 算法，可以发现 Q-learning 使用了最优 Q 值 $\max_{a'} Q(s_{t+1}, a')$ 来更新当前的 $Q(s_t, a_t)$ ，而非当前策略的 Q 值。相比之下，Q-learning 更加注重学习最优策略，而不依赖于当前的行为策略，这使得它在某些情况下能够更快地收敛到最优策略，但也可能导致在某些环境中不稳定，特别是在探索不足的情况下。而 Sarsa 则更加稳健，因为它考虑了实际采取的动作，但可能收敛速度较慢。

具体对比总结如表 2 所示。

表 2 Sarsa 和 Q-learning 对比

	Q-learning	Sarsa
策略类型	异策略	同策略
更新目标	最优动作的估计值	实际采取动作的估计值
收敛速度	较快	较慢
稳定性	更激进，可能不稳定	较为稳健
适用场景	对于需要快速收敛到最优策略的场景如游戏智能体	对于需要稳健学习的场景如自动驾驶等

同策略与异策略

同策略 on-policy 和异策略 off-policy 是强化学习中两种不同的学习范式，主要区别在于行为策略和目标策略的关系。换句话说，在智能体在与环境交互的过程中，若所采用的策略（行为策略）与其试图学习或优化的策略（目标策略）相同，称为同策略学习；反之，若行为策略与目标策略不同，则称为异策略学习。

以学习开车为例，如图 4 所示，同策略中，驾驶者按照当前学到的驾驶技巧进行驾驶，同时也在学习和改进这些技巧；而在异策略中，驾驶者心里一直想着最优的驾驶方式，但实际驾驶时可能会因为交通状况、路况等因素而采取不同的驾驶策略。换句话说，同策略就是在“学自己”，而异策略则是在“看别人学或学理想中的自己”。

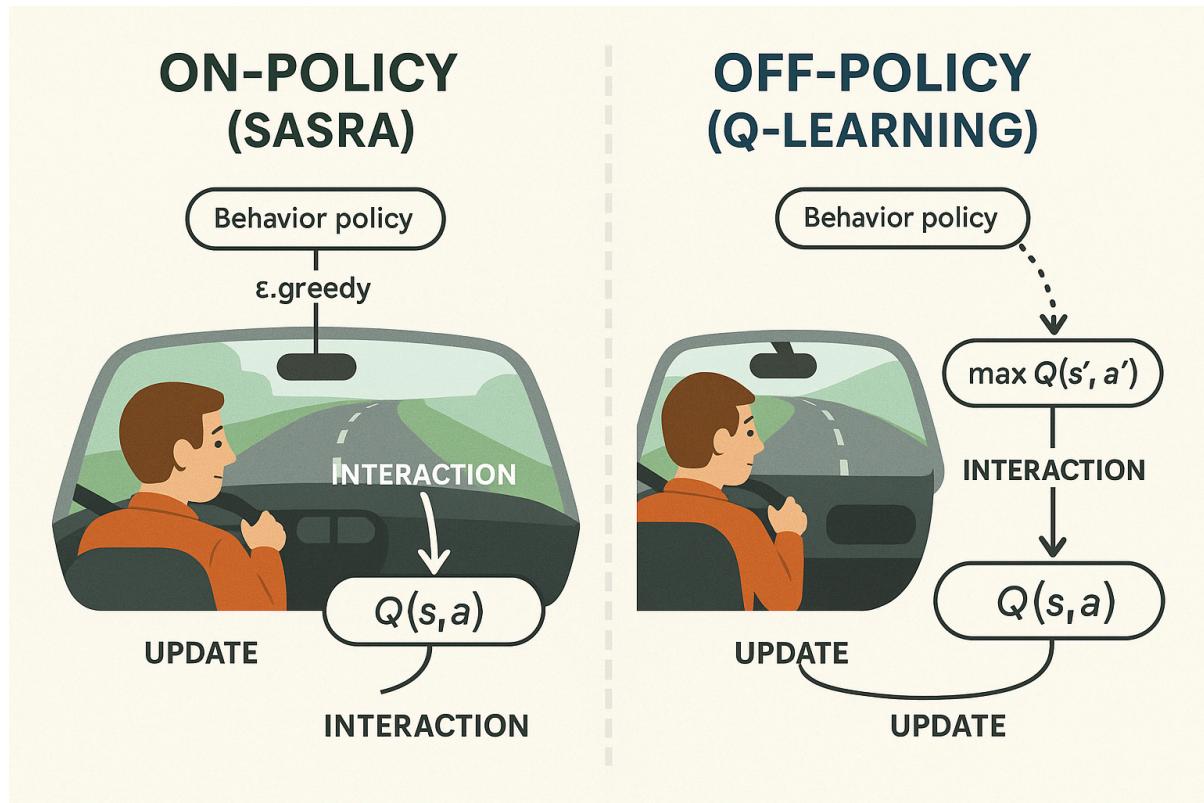


图 4: 同策略与异策略对比示意图

实际应用中，同策略方法通常更加稳健，因为它直接基于当前的行为策略进行学习，能够更好地适应环境的变化；而异策略方法则更具灵活性，能够利用不同的行为策略来探索环境，从而可能更快地找到最优策略，但也可能面临稳定性和收敛性的问题。

思考

蒙特卡洛和时序差分都是无模型方法吗？

是的，蒙特卡洛方法和时序差分方法都是无模型（model-free）方法。这意味着它们不需要对环境的动态模型进行显式建模，而是通过与环境的交互来学习价值函数或策略。

蒙特卡洛方法和时序差分方法的优劣势分别是什么？

蒙特卡洛方法优势：可以直接从经验中学习，不需要环境的转移概率；收敛性良好，可以保证在有限步内收敛到最优策略；可以处理长期回报，对于无折扣情况也可以使用。**蒙特卡洛方法劣势：**需要等到一条完整的轨迹结束才能更新价值函数，因此效率较低；对于连续状态空间和动作空间的问题，蒙特卡洛方法难以处理。**时序差分优势：**可以在交互的过程中逐步更新价值函数，效率较高；可以处理连续状态空间和动作空间的问题；可以结合函数逼近方法使用，对于高维状态空间的问题有很好的应用。**时序差分劣势：**更新过程中存在一定的方差，可能会影响收敛速度和稳定性；对于无折扣情况，需要采取一些特殊的方法来保证收敛。总的来说，蒙特卡洛方法对于离散状态空间的问题，特别是存在长期回报的问题有很好的适用性，但是其效率较低。时序差分方法则可以高效地处理连续状态空间和动作空间的问题，但是其更新过程中存在方差问题。在实际应用中需要根据问题的特点和实际情况选择合适的方法。

什么是 Q 值的过估计？有什么缓解的方法？

值的过估计 (overestimation of values) 是指在强化学习中，由于采样数据的不充分或者算法本身的限制，导致学习到的状态或动作价值函数高估了它们的真实值。值的过估计会影响强化学习算法的性能和稳定性，因此需要采取相应的缓解措施。一些缓解值的过估计的方法有：双重 Q 学习 (Double Q -learning)：将一个 Q 函数的更新过程分为两步，分别用来更新行动值函数和目标值，从而避免了 Q 函数的过估计；优先经验回放 (Prioritized Experience Replay)：在经验回放中，根据每条经验的 TD 误差大小来选择回放的概率，使得 TD 误差大的经验更有可能被回放，从而更好地修正价值函数；目标网络 (Target Network)：使用一个目标网络来计算目标值，目标网络的参数较稳定，不会随着每次更新而改变，从而减缓了价值函数的过估计问题；随机探索策略 (Exploration Strategy)：采用一些随机的探索策略，如 ϵ - greedy、高斯噪声等，可以使得智能体更多地探索未知的状态和动作，从而减少了价值函数的过估计问题。这些方法可以在不同的强化学习算法中使用，比如 DQN、DDQN、Dueling DQN 等。选择合适的方法可以有效地缓解价值函数的过估计问题。

on-policy 与 off-policy 之间的区别是什么？

on-policy 指的是学习一个策略时，使用同一策略来收集样本，并且利用这些样本来更新该策略。即学习的策略和探索的策略是相同的。off-policy 指的是学习一个策略时，使用不同于目标策略的行为策略来收集样本，并且利用这些样本来更新目标策略。即学习的策略和探索的策略是不同的。在强化学习中，通常使用 Q-learning、SARSA 等算法来实现 off-policy 学习。而使用 policy gradient 等算法来实现 on-policy 学习。on-policy 学习的优点是可以较好地处理连续动作空间的问题，并且可以保证学习到的策略收敛到最优策略。但是其缺点是样本的利用效率较低，因为样本只能用于更新当前策略，不能用于更新其他策略。off-policy 学习的优点是样本的利用效率较高，因为可以使用不同的行为策略来收集样本，并且利用这些样本来更新目标策略。但是其缺点是可能会出现样本不一致的问题，即目标策略和行为策略不同，会导致学习的不稳定性。因此，在实际应用中需要根据具体问题的特点和实际情况选择合适的学习方式。

为什么需要探索策略？

探索策略是强化学习中非常重要的一个概念，原因有：强化学习的目标是学习一个最优策略，但初始时我们并不知道最优策略，因此需要通过探索来发现更优的策略；在强化学习中，往往存在许多未知的状态和动作，如果智能体只采用已知的策略，那么它将无法探索到未知状态和动作，从而可能会错过更优的策略；探索策略可以帮助智能体避免陷入局部最优解，从而更有可能找到全局最优解。探索策略可以提高智能体的鲁棒性，使其对环境的变化更加适应。常用的探索策略包括 ϵ - greedy 策略、softmax 策略、高斯噪声等。

深度学习基础

在教程前面部分我们介绍了强化学习的基础内容，包含马尔可夫决策过程、预测与控制等，其中经典的预测与控制算法主要包括动态规划、蒙特卡洛方法和时序差分方法等。

这些算法在解决一些简单的强化学习问题时表现良好，但在面对高维度和复杂环境时，传统方法往往力不从心。为了解决这些问题，深度强化学习应运而生，它结合了深度学习的强大功能，使得强化学习能够处理更复杂的任务。

价值函数表格

在强化学习中，价值函数（如状态价值函数 $V(s)$ 和动作价值函数 $Q(s, a)$ ）是评估状态或状态-动作对好坏的关键工具，怎么表示这些函数是强化学习中的一个重要问题。

对于小规模的问题，例如前面章节中介绍的网格世界（Grid World）示例，我们可以使用表格形式来存储对应的状态或动作价值。如表 1 所示，我们可以使用一个二维表格来表示动作价值函数 $Q(s, a)$ ，其中行表示不同的动作 a ，列表示不同的状态 s 。表格中的每个元素 $Q(s, a)$ 存储了在状态 s 下采取动作 a 所获得的预期回报。

表 1: Q 表格

	s_1	s_2	s_3	s_4
a_1	-1.9	-1.0	-1.0	0.0
a_2	-1.3	-0.9	-0.7	-0.5

表格实际上是一种数学抽象形式，在程序实现上可以使用 Python 数组或者字典来表示，如代码 1 所示。

代码 1: 表格的程序实现示例

```
import numpy as np
# 使用二维数组表示 Q 表格
# 维度为 (状态数, 动作数)，例如 Q[0, 1] 表示状态 s1 下采取动作 a2 的价值
Q_table = np.array([[-1.9, -1.0, -1.0, 0.0],
                    [-1.3, -0.9, -0.7, -0.5]])
# 使用字典表示 Q 表格
# 键为 (状态, 动作) 的元组，值为对应的动作价值，例如 Q_dict[(0, 1)] 表示状态 s1 下采取动作 a2 的价值
Q_dict = {
    (0, 0): -1.9, (0, 1): -1.0, (0, 2): -1.0, (0, 3): 0.0,
    (1, 0): -1.3, (1, 1): -0.9, (1, 2): -0.7, (1, 3): -0.5
}
# 为便于理解可以使用str元组表示状态和动作
Q_dict_str = {
    ('s_1', 'a_1'): -1.9, ('s_2', 'a_1'): -1.0,
    ('s_3', 'a_1'): -1.0, ('s_4', 'a_1'): 0.0,
    ('s_1', 'a_2'): -1.3, ('s_2', 'a_2'): -0.9,
    ('s_3', 'a_2'): -0.7, ('s_4', 'a_2'): -0.5
}
```

然而，随着问题规模的增大，状态和动作空间变得庞大甚至连续，使用表格来存储价值函数变得不可行。一方面，表格的存储内存会随着状态和动作数量的增加而指数级增长，导致计算开销过大。另一方面，表格方法无法处理连续状态或动作空间，因为无法为每一个可能的状态-动作对分配一个独立的表格项。

函数近似表示

为了解决上述问题，我们可以使用函数近似的方法来表示价值函数。函数近似通过参数化的函数（如线性函数、神经网络等）来估计价值函数，从而避免了存储整个表格的需求。

在深度学习出现之前，线性函数是常用的函数近似方法。线性函数通过将状态和动作映射到一个特征空间，并使用线性组合来估计价值函数。例如，动作价值函数 $Q(s, a)$ 可以表示式 (1)。

$$Q(s, a; \theta) = \theta^T \phi(s, a) \quad (1)$$

其中 $Q(s, a; \theta)$ 有时也协作 $Q_\theta(s, a)$ ， $\phi(s, a)$ 是状态-动作对的特征向量， θ 是参数向量。通过调整参数 θ ，我们可以使得函数近似更好地拟合实际的价值函数。

但是，线性函数在处理复杂的非线性关系时表现有限，参数调整也较为困难，这限制了其在复杂强化学习任务中的应用。

除了线性函数，其他传统的函数近似方法还包括决策树、支持向量机等，这些方法在某些特定任务中也有一定的应用价值，具体总结如表 2 所示。

表 2: 典型函数近似方法总结

方法类型	核心思想	表达能力	优缺点	示例算法
线性近似	$V(s) = \theta^T \phi(s)$	弱	高效但不能表达非线性	Linear TD, LSTD
RBF 网络	高斯核函数组合	中等	可表示局部非线性，计算较重	RBF-Q
决策树/ GBDT	树状划分状态空间	中高	可解释、难与在线RL结合	Fitted Q-Iteration
局部加权回归	局部样本加权平均	中高	局部准确，全局难泛化	LWPR
Tile Coding	分层网格离散化	中	稀疏更新，高维困难	Sarsa(λ) with Tile
神经网络	层级非线性映射	很强	表达力最强，但有时也不稳定	DQN, PPO, SAC

从表中可以看出，神经网络在表达能力上最强，能够处理复杂的非线性关系，因此，现在更常用的是神经网络作为函数近似器，它能够捕捉复杂的非线性关系，并且通过反向传播算法进行高效的参数更新，具体内容将在下文展开讲解。

梯度下降

相比于表格方法，函数近似具有更好的泛化能力，但同时也引入额外的参数，因此在训练中除了考虑价值函数预测的准确性（或者说最小化预测误差）之外，还需要考虑参数的优化问题。

参数优化通常使用梯度下降（Gradient Descent）方法，通过计算损失函数关于参数的梯度，并沿着梯度的反方向更新参数，从而逐步逼近最优解。具体来说，假设我们有一个损失函数 $L(\theta)$ ，表示预测值与真实值之间的差异，那么参数更新的公式可以表示为式 (2)。

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} L(\theta) \quad (2)$$

其中 α 是学习率，控制参数更新的步长， $\nabla_{\theta}L(\theta)$ 是损失函数关于参数的梯度。

注意，梯度下降的目标是最小化损失函数，有时可能需要最大化某个目标函数（例如累积奖励），此时可使用梯度上升（Gradient Ascent）方法，如式 (3) 所示。

$$\theta \leftarrow \theta + \alpha \nabla_{\theta}J(\theta) \quad (3)$$

但在实际应用中，梯度上升可以通过优化负的损失函数来实现，即式 (4)。

$$\theta \leftarrow \theta - \alpha \nabla_{\theta}(-J(\theta)) \quad (4)$$

换句话说，梯度下降和梯度上升在数学上是很容易互相转换的，出于统一性和习惯性考虑，通常都会把问题转化为最小化损失函数的问题，然后使用梯度下降方法来进行参数优化。

在标准的梯度下降基础上，一些改进的优化算法，如随机梯度下降（SGD）、动量法（Momentum）、自适应学习率方法（如 Adam）等，这些方法在实际应用中能够提高收敛速度和稳定性。

除了梯度下降方法之外，还有其他优化方法，如牛顿法、拟牛顿法等，**这些方法通过利用二阶导数信息来加速收敛，但计算复杂度较高，通常在大规模强化学习中不常用**，具体总结如表 3 所示。

表 2: 典型参数优化方法总结

方法类型	核心思想	优点	缺点	常见算法
梯度下降	按误差方向更新参数	简单、通用	收敛慢、需调学习率	TD(λ), SARSA
最小二乘法	一次性解析求解	快速收敛	不适合在线	Monte Carlo Fit
LSTD	TD + 最小二乘	稳定、无学习率	需计算矩阵逆	LSTD(λ), LSPI
RLS	在线更新最小二乘	快、稳定	计算略复杂	Adaptive TD
共轭梯度	近似解线性系统	快、节省内存	需矩阵操作	LSTD-CG
岭回归	带正则的最小二乘	稳定性高	λ 需调	L2-TD
贝叶斯线性回归	估计参数分布	不确定性估计	计算复杂	Bayesian TD

在强化学习中，以状态价值为例，将这里的 $V(s)$ 用线性函数近似来替换原来的表格表示，如式 (5) 所示。

$$V_{\theta}(s) = \theta^T \phi(s) \quad (5)$$

其中 $\phi(s)$ 是状态 s 的特征向量， θ 是参数向量。由于是线性函数，因此对应的梯度非常简单，如式 (???) 所示。

$$\nabla_{\theta}V_{\theta}(s) = \phi(s) \quad (6)$$

如何更新参数呢？可以通过最小化函数近似的状态价值与真实价值之间的均方误差来实现，如式 (7) 所示。

$$L(\theta) = \frac{1}{2} \mathbb{E} \left[(V^{\pi}(s) - V_{\theta}(s))^2 \right] \quad (7)$$

这里 $V^{\pi}(s)$ 是在策略 π 下状态 s 的真实价值， $V_{\theta}(s)$ 是函数近似的估计值，加上 $\frac{1}{2}$ 是为了在计算梯度时方便抵消平方项的系数 2。其中真实价值 $V^{\pi}(s)$ 通常是未知的，因此我们需要使用采样得到的目标值来替代它。

具体来说，若使用蒙特卡洛估计，可以将目标值设为完整回合的累积奖励 $V^{\pi}(s_t) \approx G_t$ ，结合复合函数求导法则，得到损失函数的梯度为式 (8)。

$$\nabla_{\theta}L(\theta) = \mathbb{E} [(G_t - V_{\theta}(s_t))(-\nabla_{\theta}V_{\theta}(s_t))] \quad (8)$$

注意这里目标值 G_t 不依赖于参数 θ , 因此梯度只包含一个部分, 对应的梯度下降更新公式为式 (9)。

$$\begin{aligned}\theta &\leftarrow \theta - \alpha [G_t - V_\theta(s_t)](-\nabla_\theta V_\theta(s_t)) \\ &= \theta - \alpha [V_\theta(s_t) - G_t]\nabla_\theta V_\theta(s_t) \\ &= \theta - \alpha [V_\theta(s_t) - G_t]\phi(s_t)\end{aligned}\quad (9)$$

若使用时序差分估计, 可以将目标值设为单步奖励加上下一个状态的估计价值 $V^\pi(s_t) \approx R_{t+1} + \gamma V_\theta(s_{t+1})$, 则梯度如式 (10) 所示。

$$\nabla_\theta L(\theta) = \mathbb{E} [(R_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t))(\gamma \nabla_\theta V_\theta(s_{t+1}) - \nabla_\theta V_\theta(s_t))] \quad (10)$$

注意, 这里目标值 $R_{t+1} + \gamma V_\theta(s_{t+1})$ 也依赖于参数 θ , 因此梯度包含了两个部分。但是由于实际应用中通常忽略目标值对参数的依赖, 即半梯度更新, 对应的梯度下降更新公式为式 (11)。

$$\begin{aligned}\theta &\leftarrow \theta - \alpha [V_\theta(s_t) - (R_{t+1} + \gamma V_\theta(s_{t+1}))]\nabla_\theta V_\theta(s_t) \\ &= \theta - \alpha [V_\theta(s_t) - R_{t+1} + \gamma V_\theta(s_{t+1})]\phi(s_t)\end{aligned}\quad (11)$$

半梯度更新 TD(0) 方法虽然忽略了目标值对参数的依赖, 但这样的做法已经被证明可以收敛到一个较好的解, 且计算更为简单。此外如果直接使用完整梯度, 反而会破坏时序差分的递推结果, 从而导致不稳定。

梯度下降示例

本节将继续以前面章节中的 3×3 网格世界为例, 演示如何使用线性函数近似和梯度下降, 并结合时序差分方法来估计状态价值函数 $V(s)$ 。

先回顾一下网格世界的环境设置, 如图 1 所示。考虑智能体在 3×3 的网格中使用随机策略进行移动, 以左上角为起点, 右下角为终点, 同样规定每次只能向右或向下移动, 动作分别用 a_1 和 a_2 表示。用智能体的位置不同的状态, 即 s_1, s_2, \dots, s_9 , 初始状态为 $S_0 = s_1$, 终止状态为 s_9 。

s_1	s_2	s_3	起点	a_1 向右	a_2 向下
s_4	s_5	s_6	终点	$R(s_i) = -1$	
s_7	s_8	s_9	水洼	$R(s_4) = R(s_i) - 3.0$	
			深坑	$R(s_5) = R(s_i) - 0.5$	
				$R(s_9) = R(s_i) + 1$	

图 1: 3×3 网格示例

除了每走一步接收 -1 的奖励之外, 这次我们在网格中增加了一些障碍物, 例如在位置 s_4 处设置了一个深坑, 智能体走到该位置时会受到一个额外的负奖励 -3 , 在位置 s_5 处设置了一个水洼, 智能体走到该位置时会受到一个额外的负奖励 -0.5 。折扣因子 $\gamma = 0.9$, 目标是计算各个状态的价值函数 $V(s)$ 。

在这个网格示例中, 我们可以设计一个简单的特征映射 $\phi(s)$, 将状态 s 映射为一个包含位置坐标和障碍物信息的特征向量。如何定义这个特征向量的过程叫作**特征工程**, 设计特征时需要考虑特征的表达能力和计算复杂度之间的平衡, 往往也会需要一些编码技巧。

在本例中, 状态 s 的特征向量可以设计如表 4 所示。

, 状态 s 的特征向量可以设计如表 4 所示。

表 4: 状态特征映射示例

状态 s	行坐标	列坐标	行列乘积	行坐标平方	列坐标平方	是否深坑	是否水洼
s_1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
s_2	0.0	0.5	0.0	0.0	0.25	0.0	0.0
s_3	0.0	1.0	0.0	0.0	1.0	0.0	0.0
s_4	0.5	0.0	0.0	0.25	0.0	1.0	0.0
s_5	0.5	0.5	0.25	0.25	0.25	0.0	1.0
s_6	0.5	1.0	0.5	0.25	1.0	0.0	0.0
s_7	1.0	0.0	0.0	1.0	0.0	0.0	0.0
s_8	1.0	0.5	0.5	1.0	0.25	0.0	0.0
s_9	1.0	1.0	1.0	1.0	1.0	0.0	0.0
特征表示	r	c	$r \times c$	r^2	c^2	is_pit	is_puddle
权重表示	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7

注意行列坐标为了避免数值过大，均做了归一化处理，取值范围为 $[0, 1]$ 。此外，还引入了两个二值特征，分别表示当前状态是否为深坑 (is_pit) 或水洼 (is_puddle)。

然后再加上偏置项，总共八个特征。这样，状态 s 的价值函数估计 (5) 可以展开如式 (12) 所示。

$$V_\theta(s) = \theta_0 + \theta_1 \cdot r + \theta_2 \cdot c + \theta_3 \cdot (r \times c) + \theta_4 \cdot r^2 + \theta_5 \cdot c^2 + \theta_6 \cdot \text{is_pit} + \theta_7 \cdot \text{is_puddle} \quad (12)$$

基于上述设置，我们可以实现一个简单的半梯度 TD(0) 算法来估计状态价值函数 $V(s)$ ，具体如代码 2 所示。

代码 2: 使用线性函数近似和半梯度 TD(0) 估计状态价值函数

```

import numpy as np
import pandas as pd
import random

# ----- MDP Setup -----
gamma = 0.9
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2),
}

def legal_actions(s):
    r,c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")

```

```

    return acts

def step(s, a):
    r,c = coords[s]
    if a == "right": r2,c2 = r, c+1
    else:             r2,c2 = r+1, c
    # next state
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    reward = -1.0
    if s2=="s4": reward -= 3.0      # pit
    if s2=="s5": reward -= 0.5      # puddle
    if s2=="s9": reward += 1.0      # terminal bonus -> net 0
    done = (s2==terminal)
    return s2, reward, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ----- feature map -----
def phi(s):
    r, c = coords[s]
    rn, cn = r/2.0, c/2.0
    is_pit = 1.0 if s=="s4" else 0.0
    is_puddle = 1.0 if s=="s5" else 0.0
    return np.array([1.0, rn, cn, rn*cn, rn*rn, cn*cn, is_pit, is_puddle], dtype=float)

# ----- semi-gradient TD(0) -----
def td0_linear_value(episodes=20000, alpha=0.05, gamma=0.9):
    w = np.zeros(len(phi("s1")), dtype=float)
    for _ in range(episodes):
        s = "s1"
        while s != terminal:
            a = random_policy(s)
            s2, r, done = step(s, a)
            v_s = float(w @ phi(s))
            v_s2 = 0.0 if done else float(w @ phi(s2))
            td_error = r + gamma*v_s2 - v_s
            w += alpha * td_error * phi(s)
            s = s2
    return w

w = td0_linear_value()

def v_hat(s): return float(w @ phi(s)) if s!="s9" else 0.0
grid = np.array([[v_hat(f"s{r*3+c+1}") for c in range(3)] for r in range(3)])
print(pd.DataFrame(np.round(grid,3),
                  index=["row1","row2","row3"],
                  columns=["col1","col2","col3"]))
print("\nweights:", np.round(w,3))

```

执行代码后，可以得到结果如代码 3 所示。注意由于随机性的存在，每次运行结果会有些许差异，但整体趋势是一致的。这个现象在所有的迭代更新中都存在，尤其是在使用随机梯度下降方法时更为明显。

代码 3: 线性函数近似估计的状态价值函数结果

```
    col1  col2  col3
row1 -4.456 -2.148 -0.906
row2 -2.030 -0.866  0.117
row3 -0.875  0.185  0.000

weights: [-4.456  5.501  5.679 -2.494 -1.921 -2.13   0.155 -0.364]
```

可以看到，状态价值函数 $V(s)$ 的估计结果与之前使用表格方法得到的结果较为接近，说明线性函数近似结合梯度下降和时序差分方法在这个示例问题中能够有效地估计状态价值函数。

独热编码

前面在示例中，我们使用了手工设计的特征映射 $\phi(s)$ 来表示状态 s ，这种方法在状态空间较小且结构明确的情况下效果较好，但在更复杂的环境中，手工设计特征可能变得困难且不够灵活，因此需要更通用的状态表示方法。

注意到，状态 s 在这个示例中是离散的，是不是可以直接用整数来表示状态呢？例如用 $1, 2, 3, \dots$ 来分别表示状态 s_1, s_2, s_3, \dots 。虽然这样做简单直接，但可能会导致神经网络难以学习到有效的特征。因为这些状态之间并没有实际的数值关系，更多的是不同的类别或标签，直接使用整数表示可能会导致神经网络难以学习到有效的特征。

对于离散的状态，为了兼顾表达的通用性和神经网络的学习难度，我们可以使用独热编码（One-Hot Encoding）来表示状态 s 。独热编码将每个离散状态映射为一个高维向量，其中只有对应状态的位置为 1，其他位置为 0。例如，在 3×3 网格示例中，假设有九个离散状态 s_1, s_2, \dots, s_9 ，它们的独热编码表示如式 (13) 所示。

$$\begin{aligned} s_1 &: [1, 0, 0, 0, 0, 0, 0, 0, 0] \\ s_2 &: [0, 1, 0, 0, 0, 0, 0, 0, 0] \\ s_3 &: [0, 0, 1, 0, 0, 0, 0, 0, 0] \\ s_4 &: [0, 0, 0, 1, 0, 0, 0, 0, 0] \\ s_5 &: [0, 0, 0, 0, 1, 0, 0, 0, 0] \\ s_6 &: [0, 0, 0, 0, 0, 1, 0, 0, 0] \\ s_7 &: [0, 0, 0, 0, 0, 0, 1, 0, 0] \\ s_8 &: [0, 0, 0, 0, 0, 0, 0, 1, 0] \\ s_9 &: [0, 0, 0, 0, 0, 0, 0, 0, 1] \end{aligned} \tag{13}$$

对应的函数表达式为式 (14)。

$$\phi(s_i) = [0, 0, \dots, 1, \dots, 0]^T \in \mathbb{R}^n, \quad (\text{第 } i \text{ 个位置为 1, 其余为 0}) \tag{14}$$

其中 n 是状态的总数。使用独热编码后，函数近似的状态价值函数可以表示为式 (15)。

$$V_\theta(s_i) = \boldsymbol{\theta}^T \phi(s_i) = \theta_i \tag{15}$$

也就是说，使用独热编码时，线性函数近似实际上等价于表格方法，每个状态对应一个独立的参数 θ_i ，从而实现对每个状态价值的单独估计。

基于独热编码，我们同样可以实现一个简单的半梯度 TD(0) 算法来估计状态价值函数 $V(s)$ ，具体如代码 4 所示。

代码 4: 使用独热编码和半梯度 TD(0) 估计状态价值函数

```
import numpy as np
import random
```

```

import pandas as pd

# ----- 环境 -----
gamma = 0.9
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2),
}
}

def legal_actions(s):
    r,c = coords[s]
    acts=[]
    if c<2: acts.append("right")
    if r<2: acts.append("down")
    return acts

def step(s,a):
    r,c = coords[s]
    if a=="right": r2,c2 = r, c+1
    else:           r2,c2 = r+1, c
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    reward = -1.0
    if s2=="s4": reward -= 3.0      # 深坑
    if s2=="s5": reward -= 0.5      # 水洼
    if s2=="s9": reward += 1.0      # 终点净0
    done = (s2==terminal)
    return s2, reward, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ----- One-hot 特征 -----
def phi(s):
    vec = np.zeros(9)
    vec[int(s[1:])-1] = 1.0
    return vec

d = 9                      # 参数维度
theta = np.zeros(d)          # 线性权重
alpha = 0.1
episodes = 20000

def v_hat(s):
    return np.dot(theta, phi(s))

# ----- TD(0) 半梯度 -----
for ep in range(episodes):
    s = "s1"
    while s != terminal:
        a = random_policy(s)

```

```

s2, r, done = step(s, a)
target = r + (0 if done else gamma * v_hat(s2))
delta = target - v_hat(s)
theta += alpha * delta * phi(s)    # 线性半梯度更新
s = s2

# ----- 输出 -----
v_est = {s: (0.0 if s==terminal else v_hat(s)) for s in states}
grid = np.array([[v_est[f"s{r*3+c+1}"] for c in range(3)] for r in range(3)])
df = pd.DataFrame(np.round(grid,3), index=["row1","row2","row3"], columns=
["col1","col2","col3"])

print("线性函数近似 (one-hot) + TD(0) 学到的状态价值: ")
print(df)

print("\n参数向量 θ (对应每个状态的估计值) :")
print(pd.Series(np.round(theta,3), index=states))

```

执行代码后，参考结果如代码 5 所示。

代码 5: 独热编码估计的状态价值函数结果

线性函数近似 (one-hot) + TD(0) 学到的状态价值:

	col1	col2	col3
row1	-4.445	-2.09	-1.0
row2	-2.128	-1.00	0.0
row3	-1.000	0.00	0.0

参数向量 θ (对应每个状态的估计值) :

s1	-4.445
s2	-2.090
s3	-1.000
s4	-2.128
s5	-1.000
s6	0.000
s7	-1.000
s8	0.000
s9	0.000

dtype: float64

可以看到，得到的结果跟之前的状态价值估计方法是接近的，说明独热编码作为一种通用的状态表示方法，能够有效地支持线性函数近似和梯度下降方法来估计状态价值函数。

然而当状态空间变得更大或更复杂时，独热编码的维度也会随之增加，导致计算和存储开销变大，并且无法捕捉状态之间的相似性和结构信息。因此，在更复杂的环境中，需要不同的编码方式，例如嵌入式表示（embedding）等，然后再输入到函数近似模型或神经网络中进行处理，这样可以更有效地利用状态之间的关系和特征。

神经网络近似

前面讲到，线性函数近似或者拟合价值函数在某些简单问题中表现良好，但在面对复杂的环境和高维状态空间时，线性函数的表达能力有限，难以捕捉复杂的非线性关系。

随着深度学习的发展，神经网络成为了更强大的函数近似工具。神经网络通过多层非线性变换，能够捕捉复杂的模式和关系，从而更准确地估计价值函数，如式 (7) 所示。

$$Q_{\theta}(s, a) = \text{NN}(s, a; \theta) \quad (16)$$

神经网络一般包含三个主要部分：输入层、隐藏层和输出层，其中隐藏层可以有多层，每层包含多个神经元，通过激活函数实现非线性变换。如图 2 所示，假设我们使用一个简单的前馈神经网络来近似动作价值函数 $Q(s, a)$ 。输入层包含状态 s （有时也包含动作 a ）的特征表示，经过隐藏层的非线性变换，最终输出对应的动作价值估计。

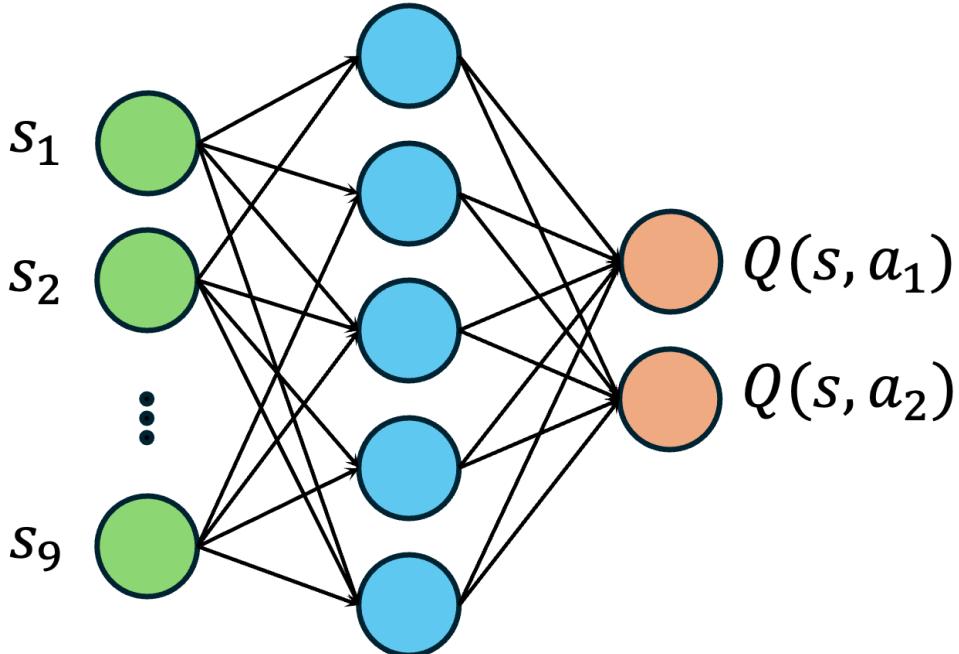


图 2: 使用神经网络近似动作价值函数 $Q(s, a)$

对于线性输入，即一维特征向量，一般可以用全连接层（fully connected layer，简称 FC）将输入映射到隐藏层，然后通过激活函数（如 ReLU、Sigmoid 等）引入非线性，最后再通过输出层得到价值估计。对应的图示可以简化为图 3 所示。

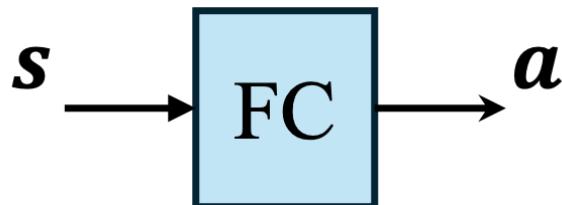


图 3: 神经网络全连接层示意图

再回到前面的 3×3 网格示例，我们可以使用神经网络来近似状态价值函数 $V(s)$ ，并结合半梯度 TD(0) 方法进行训练。具体实现如代码 6 所示。

代码 6: 使用神经网络和半梯度 TD(0) 估计状态价值函数

```
"""
3x3 网格: NN 近似 v_pi(s)
- 动作: right / down (越界不可选)
- 奖励: 每步 -1; 进 s4 额外 -3; 进 s5 额外 -0.5; 到 s9 +1 (等价进终点净0)
- 策略: 在可行动作间均匀随机
- 训练: TD(0) 半梯度 (也可切换成蒙特卡洛回归)
"""

import math, random, numpy as np
import torch, torch.nn as nn, torch.optim as optim
```

```

from collections import defaultdict

# ----- 环境 -----
gamma = 0.9
states = [f"s{i}" for i in range(1,10)]
terminal = "s9"
coords = {
    "s1":(0,0), "s2":(0,1), "s3":(0,2),
    "s4":(1,0), "s5":(1,1), "s6":(1,2),
    "s7":(2,0), "s8":(2,1), "s9":(2,2),
}
def legal_actions(s):
    r,c = coords[s]
    acts=[]
    if c<2: acts.append("right")
    if r<2: acts.append("down")
    return acts

def step(s,a):
    r,c = coords[s]
    if a=="right": r2,c2=r,c+1
    else:           r2,c2=r+1,c
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    # 奖励
    rwd = -1.0
    if s2=="s4": rwd -= 3.0
    if s2=="s5": rwd -= 0.5
    if s2=="s9": rwd += 1.0    # 到终点净0
    done = (s2==terminal)
    return s2, rwd, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ----- 特征（可换成 one-hot） -----
def phi(s):
    r,c = coords[s]
    rn, cn = r/2.0, c/2.0                      # 归一化到 [0,1]
    is_pit   = 1.0 if s=="s4" else 0.0
    is_puddle = 1.0 if s=="s5" else 0.0
    return np.array([1.0, rn, cn, rn*cn, rn*rn, cn*cn, is_pit, is_puddle],
    dtype=np.float32)

feat_dim = len(phi("s1"))

# ----- 神经网络 -----
class valueNet(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 64), nn.ReLU(),
            nn.Linear(64, 64), nn.ReLU(),
            nn.Linear(64, 1)

```

```

)
def forward(self, x):    # x: [B, in_dim]
    return self.net(x).squeeze(-1)

device = torch.device("cpu")
net = ValueNet(feat_dim).to(device)
opt = optim.Adam(net.parameters(), lr=1e-3)

# ----- 训练开关 -----
USE_TD0 = True    # True: TD(0); False: 蒙特卡洛回归

# ----- 训练 -----
episodes = 40000
batch_buf_s, batch_buf_target = [], []

def tensorify(ss): # list[str] -> tensor
    feats = np.stack([phi(s) for s in ss], axis=0)
    return torch.tensor(feats, dtype=torch.float32, device=device)

for ep in range(episodes):
    # Exploring starts 也行; 此处固定从 s1
    s = "s1"
    traj = [] # for MC
    while s != terminal:
        a = random_policy(s)
        s2, rwd, done = step(s, a)

        if USE_TD0:
            # TD(0) 半梯度: target = r + γ v(s'), 并 detach v(s')
            with torch.no_grad():
                v_sp = 0.0 if done else net(tensorify([s2]))[0].item()
            target = rwd + gamma * v_sp
            preds = net(tensorify([s]))
            loss = nn.MSELoss()(preds, torch.tensor([target], dtype=torch.float32,
device=device))
            opt.zero_grad(); loss.backward(); opt.step()
        else:
            # 先记轨迹, 等 episode 结束再做 MC 回归
            traj.append((s, rwd))
            s = s2

    if not USE_TD0:
        # 蒙特卡洛: 从后往前算 G, 并回到到 v(s)
        G = 0.0
        for s, rwd in reversed(traj):
            G = rwd + gamma*G
            batch_buf_s.append(s); batch_buf_target.append(G)
        # 小批量更新
        if len(batch_buf_s) >= 64:
            x = tensorify(batch_buf_s)
            y = torch.tensor(batch_buf_target, dtype=torch.float32, device=device)
            pred = net(x)
            loss = nn.MSELoss()(pred, y)

```

```

        opt.zero_grad(); loss.backward(); opt.step()
        batch_buf_s.clear(); batch_buf_target.clear()

# ----- 评估与打印 -----
with torch.no_grad():
    grid = np.zeros((3,3), dtype=np.float32)
    for r in range(3):
        for c in range(3):
            sid = f"s{r*3+c+1}"
            grid[r,c] = 0.0 if sid==terminal else net(tensorify([sid]))[0].item()
    print("Estimated V_pi(s) by NN (rows=row1..row3):")
    print(np.round(grid, 3))

# 热力图:
# import matplotlib.pyplot as plt
# plt.imshow(grid, origin='upper');
# for i in range(3):
#     for j in range(3): plt.text(j,i,f"{grid[i,j]:.2f}",ha='center',va='center')
# plt.title("NN-approximated V(s)"); plt.colorbar(); plt.show()

```

执行代码后，参考结果如代码 7 所示。

代码 7: 神经网络估计的状态价值函数结果

```

Estimated V_pi(s) by NN (rows=row1..row3):
[[-4.223 -2.105 -1.029]
 [-2.089 -1.019  0.01 ]
 [-1.004  0.034  0.    ]]

```

可以看到，神经网络成功地近似了状态价值函数 $V(s)$ ，并且结果与之前的方法相似，说明神经网络作为一种强大的函数近似工具，能够有效地支持强化学习中的价值函数估计任务。

我们再使用独热编码来表示状态输入，并使用相同的神经网络结构进行训练，具体实现如代码 8 所示。

代码 8: 使用独热编码和神经网络估计状态价值函数

```

"""
TD(0) 半梯度 + 神经网络 + one-hot 状态表示
3x3 网格: right/down; 深坑-3, 水洼-0.5, 每步-1, 终点+1 (净0)
"""

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random

# ----- 环境 -----
gamma = 0.9
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
}

```

```

        "s7": (2,0), "s8": (2,1), "s9": (2,2),
    }

def legal_actions(s):
    r,c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")
    return acts

def step(s,a):
    r,c = coords[s]
    if a == "right": r2,c2 = r, c+1
    else:             r2,c2 = r+1, c
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    reward = -1.0
    if s2 == "s4": reward -= 3.0
    if s2 == "s5": reward -= 0.5
    if s2 == "s9": reward += 1.0 # 到终点净0
    done = (s2 == terminal)
    return s2, reward, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ----- One-hot 状态编码 -----
def onehot(s):
    idx = int(s[1:]) - 1 # s1->0, s2->1, ...
    vec = np.zeros(9, dtype=np.float32)
    vec[idx] = 1.0
    return vec

# ----- 神经网络 -----
class ValueNet(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 32), nn.ReLU(),
            nn.Linear(32, 1)
        )
    def forward(self, x):
        return self.net(x).squeeze(-1)

device = torch.device("cpu")
net = ValueNet(9).to(device)
opt = optim.Adam(net.parameters(), lr=1e-3)

# ----- 训练 -----
episodes = 40000
alpha = 0.1

def tensorify(slist):
    feats = np.stack([onehot(s) for s in slist], axis=0)

```

```

    return torch.tensor(feats, dtype=torch.float32, device=device)

for ep in range(episodes):
    s = "s1"
    while s != terminal:
        a = random_policy(s)
        s2, r, done = step(s, a)

        with torch.no_grad():
            v_next = 0.0 if done else net(tensorify([s2]))[0].item()
            target = r + gamma * v_next
            v_pred = net(tensorify([s]))
            loss = (v_pred - target) ** 2

            opt.zero_grad()
            loss.backward()
            opt.step()
            s = s2

# ----- 结果可视化 -----
with torch.no_grad():
    grid = np.zeros((3,3))
    for r in range(3):
        for c in range(3):
            sid = f"s{r*3+c+1}"
            grid[r,c] = 0.0 if sid==terminal else net(tensorify([sid]))[0].item()

print("Estimated v_pi(s) by NN with onehot (rows=row1..row3):")
print(np.round(grid, 3))

# ----- 热力图展示 -----
# import matplotlib.pyplot as plt
# plt.imshow(grid, cmap='coolwarm', origin='upper')
# for i in range(3):
#     for j in range(3):
#         plt.text(j,i,f"{grid[i,j]:.2f}",ha='center',va='center',color='black')
# plt.title("Value Approximation with One-hot Encoding (TD(0))")
# plt.colorbar(label="V(s)")
# plt.tight_layout()
# plt.show()

```

执行代码后，参考结果如代码 9 所示。

代码 9: 独热编码神经网络估计的状态价值函数结果

```

Estimated v_pi(s) by NN with onehot (rows=row1..row3):
[[-4.525e+00 -2.156e+00 -9.900e-01]
 [-2.092e+00 -1.004e+00 -1.300e-02]
 [-1.005e+00  2.000e-03  0.000e+00]]

```

可以看到，使用独热编码作为状态表示，神经网络同样能够有效地近似状态价值函数 $V(s)$ ，并且结果与之前的方法是相似的。

到目前为止，结合前面章节内容，对于 3×3 网格示例，我们使用了多种方法来估计状态价值函数 $V(s)$ ，包括表格方法、线性函数近似（手工特征和独热编码）以及神经网络近似（手工特征和独热编码），并主要使用时序差分方法 (TD(0)) 进行训练迭代。这些方法的结果都较为接近，说明它们在这个简单环境中都能有效地估计价值函数。

然而，随着环境复杂度的增加，例如状态空间变大、状态和动作的关系更复杂，简单的表格或者线性函数近似可能无法捕捉到足够的信息，而神经网络由于其强大的表达能力，往往能够更好地适应复杂环境中的价值函数估计任务。

神经网络拓展

前面我们介绍了使用基础的前馈神经网络来近似价值函数的方法，这种神经网络通常由若干个全连接层组成，并使用非线性激活函数来增强表达能力，也就是我们常说的多层感知机 (multi-layer perceptron, MLP) 或全连接网络 (fully connected network)。这种神经网络结构适用于处理低维的、结构化的输入数据，例如前面示例中的手工设计特征或者独热编码等。

然而，在实际应用中，针对不同类型的数据和任务，可能需要使用更复杂的神经网络结构来更好地捕捉数据的特征和模式。例如，卷积神经网络 (convolutional neural network, CNN) 适用于处理图像数据，能够有效地提取空间特征；循环神经网络 (recurrent neural network, RNN) 适用于处理序列数据，能够捕捉时间依赖关系，具体总结如表 1 所示。

表 1: 常用神经网络类型及其特点

神经网络类型	适用场景	主要特点
全连接网络 (MLP)	低维结构化数据	多层全连接层，适用于一般函数近似
卷积神经网络 (CNN)	图像、视频等网格数据	局部感受野、权重共享、池化层
循环神经网络 (RNN)	序列数据（文本、时间序列）	循环连接，捕捉时间依赖关系
LSTM / GRU	长序列数据	门机制，解决梯度消失问题
Transformer	序列数据（文本、时间序列）	自注意力机制，适合并行计算
图神经网络 (GNN)	图结构数据	节点和边的特征传播

在强化学习中，选择合适的神经网络结构对于成功应用深度强化学习算法至关重要。此外，复杂的神经网络通常需要更多的数据和计算资源来进行训练，因此在设计神经网络时需要权衡模型复杂度和计算效率，以确保模型能够在合理的时间内收敛并达到良好的性能。

DQN 算法

DQN 算法的核心思想是在 Q-learning 算法的基础上引入深度神经网络来近似动作价值函数 $Q(s, a)$ ，从而能够处理高维的状态空间。此外，DQN 算法还引入了一些技巧，如经验回放和目标网络等，来提高训练的稳定性和效果。

Q 网络

在深度学习基础章节中，我们演示了如何使用神经网络来近似状态价值函数 $V(s)$ 并使用梯度下降和 TD 误差来更新网络参数。类似地，我们也可以使用神经网络来近似动作价值函数 $Q(s, a)$ ，回顾 Q-learning 算法的更新公式，如式 (1) 所示。

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q'(s_{t+1}, a) - Q(s_t, a_t)] \quad (1)$$

其中 $y_t = r_t + \gamma \max_a Q'(s_{t+1}, a)$ 表示期望或目标的 Q 值， $Q(s_t, a_t)$ 表示实际的 Q 值， α 是学习率。引入神经网络近似 Q 函数后，如式 (2) 所示。

$$Q_\theta(s_t, a_t) \leftarrow Q_\theta(s_t, a_t) + \alpha[y_t - Q_\theta(s_t, a_t)] \quad (2)$$

其中参数 θ 是神经网络的参数，可以通过梯度下降的方式来优化。具体来说，可以以最小化 TD 误差为目标，即最小化目标 Q 值和实际 Q 值之间的差距，如式 (3) 所示。

$$\begin{aligned} L(\theta) &= (y_t - Q_\theta(s_t, a_t))^2 \\ \theta &\leftarrow \theta - \alpha \nabla_\theta L(\theta) \end{aligned} \quad (3)$$

注意，由于是基于 TD 更新的，因此依然需要判断终止状态，如式 (4) 所示。

$$y_t = \begin{cases} r_t & \text{对于终止状态 } s_t \\ r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') & \text{对于非终止状态 } s_t \end{cases} \quad (4)$$

经验回放

在 Q-learning 算法中，我们每次执行一步交互得到一个样本之后，就立即用这个最新的样本去更新 Q 函数，如式 (5) 所示。

$$(s_t, a_t, s_{t+1}, r_{t+1}) \rightarrow \text{update } Q(s_t, a_t) \quad (5)$$

这种更新方式在引入神经网络后会带来一些问题。首先，每次用单个样本去迭代网络参数很容易导致训练的不稳定，从而影响模型的收敛。其次，每次迭代的样本都是从环境中实时交互得到的，这样的样本是在时间上是连续的，即是有关联的，这与梯度下降法的数据假设不符，即**训练集中的样本必须是独立同分布的**。此外，每个样本只被使用一次就被丢弃，数据使用率较低，而在强化学习中与环境交互产生样本的成本往往较高，**较低的数据使用率会严重影响训练效率**。

为了解决这些问题，DQN 算法引入了经验回放机制。具体来说，我们会将每次与环境交互得到的样本 $(s_t, a_t, s_{t+1}, r_{t+1})$ 存储在一个经验回放池（Replay Buffer）中，如式 (6) 所示。

$$(s_t, a_t, s_{t+1}, r_{t+1}) \rightarrow D \quad (6)$$

然后在每次更新网络参数时，我们会从经验回放池中随机抽取一个小批量的样本进行训练，如式 (7) 所示。

$$D \rightarrow \{(s_i, a_i, s_{i+1}, r_{i+1})\}_{i=1}^N \quad (7)$$

其中下标 i 表示从经验回放池中随机抽取的样本，意味着不再跟时间相关， N 是小批量的样本数量。利用这些样本，我们可以计算对应的目标 Q 值 y_i ，并使用梯度下降的方法来更新网络参数 θ ，如式 (8) 所示。

$$L(\theta) = \mathbb{E}_{(s_i, a_i, s_{i+1}, r_{i+1}) \sim D} [(y_i - Q_\theta(s_i, a_i))^2]$$

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$
(8)

如图 1 所示，不同于 Q-learning 算法每次用最新的样本直接喂入神经网络去更新网络模型，DQN 算法会把每次与环境交互得到的样本都存储在一个经验回放中，然后每次从经验池中随机抽取一批样本来训练网络。

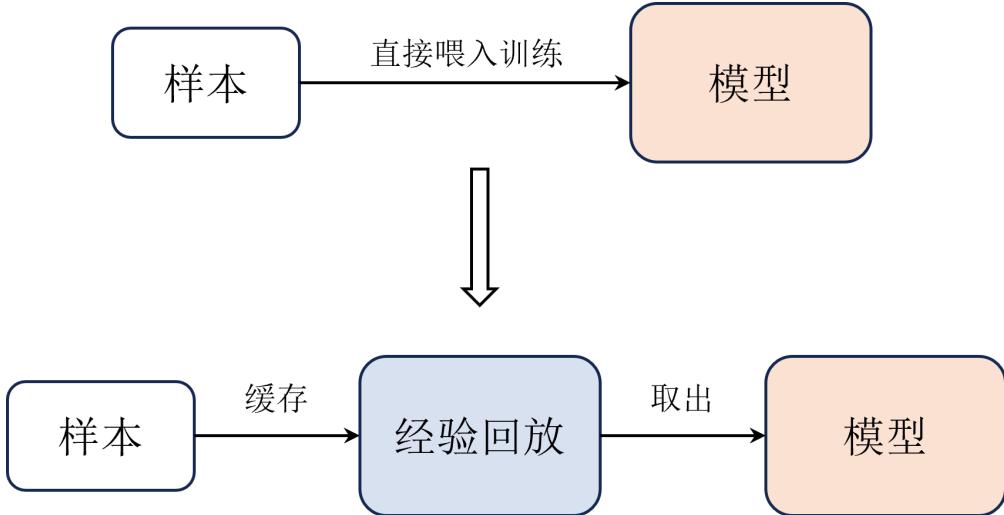


图 1: 经验回放示例

目标网络

注意到在式 (4) 中，目标值 y_t 的参数 θ 与实际值即当前网络 Q_θ 的参数是相同的。这意味着网络要学习的目标会在每次更新时发生变化，等价于我们在追逐一个不断移动的目标，这很容易造成训练的不稳定，甚至损失发散导致无法收敛（即训练失败）。

为了解决这个问题，DQN 算法引入了目标网络（Target Network）的概念。具体来说，我们会维护一个与当前网络结构相同但参数不同的目标网络 Q_{θ^-} ，并使用目标网络来计算目标 Q 值，如式 (9) 所示。

$$y_t = \begin{cases} r_t & \text{对于终止状态 } s_t \\ r_t + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a') & \text{对于非终止状态 } s_t \end{cases} \quad (9)$$

目标网络的参数 θ^- 会定期地从当前网络的参数 θ 复制过来，如式 (10) 所示。

$$\theta^- \leftarrow \theta \quad \text{Update every } C \text{ steps} \quad (10)$$

其中 C 是一个超参数，表示每隔多少步更新一次目标网络的参数。这样一来，目标网络的参数在会在一定时间内保持不变，从而使得目标 Q 值相对稳定，避免了追逐不断移动的目标的问题，提高了训练的稳定性和收敛性，如图 2 所示。

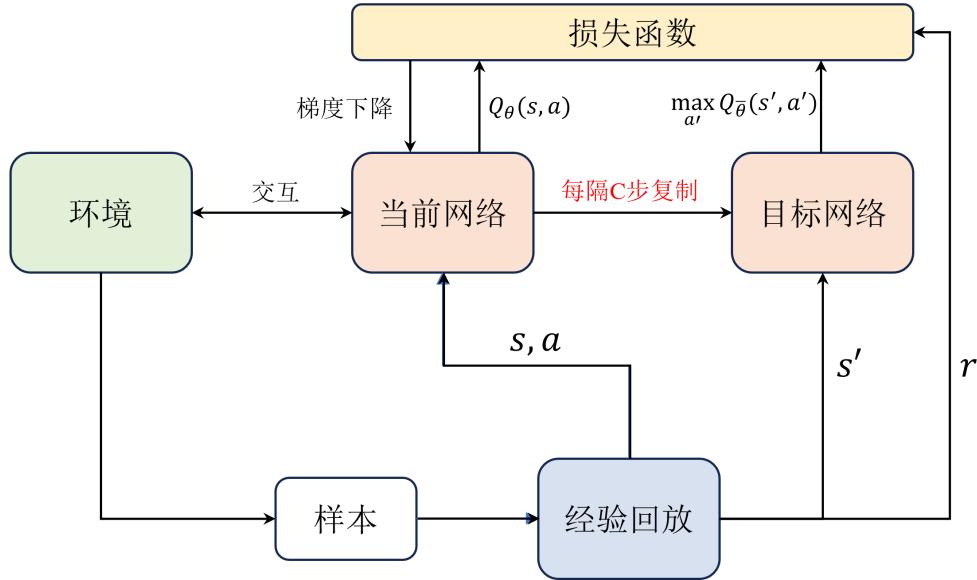


图 2: 目标网络示例

另外，式 (10) 中的更新方式被称为硬更新 (Hard Update)，即每隔 C 步直接将当前网络的参数复制给目标网络。除此之外，还有一种软更新 (Soft Update) 的方式，即每次更新时将目标网络的参数向当前网络的参数靠近一点，如式 (11) 所示。

$$\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^- \quad (11)$$

其中 τ 是一个超参数，通常取值很小（如 0.001），表示每次更新时目标网络参数向当前网络参数靠近的比例。注意，软更新的方式在 DQN 算法中并不常用，原因是硬更新已经能够很好地解决目标不稳定的问题，但在一些后续的强化学习算法例如 DDPG 和 TD3 中被广泛采用。

算法流程

DQN 算法的完整流程如图 3 所示，包括初始化网络和经验回放池、与环境交互采样、存储样本到经验回放池、从经验回放池中随机采样小批量样本、计算目标 Q 值和损失函数、更新网络参数以及定期更新目标网络参数等步骤，完整的代码实现可参考实战部分的内容。

DQN 算法

```
1: 初始化当前网络参数  $\theta$  和目标网络参数  $\theta^- \leftarrow \theta$ 
2: 初始化经验回放  $D$ 
3: for 回合数  $m = 1, 2, \dots, M$  do
4:   重置环境，获得初始状态  $s_0$ 
5:   for 时步  $t = 1, 2, \dots, T$  do
6:     交互采样：
7:       根据  $\varepsilon - greedy$  策略采样动作  $a_t$ 
8:       环境根据  $a_t$  反馈奖励  $r_t$  和下一个状态  $s_{t+1}$ 
9:       存储样本  $(s_t, a_t, r_t, s_{t+1})$  到经验回放  $D$  中
10:      更新状态  $s_{t+1} \leftarrow s_t$ 
11:      策略更新：
12:        从  $D$  中随机采样一个批量的样本
13:        计算  $Q$  的期望值，即  $y_i = r_t + \gamma \max_{a_{i+1}} Q_{\theta^-}(s_{i+1}, a)$ 
14:        计算损失  $L(\theta) = (y_i - Q_{\theta}(s_i, a_i))^2$  并梯度下降更新参数  $\theta$ 
15:        每  $C$  步复制参数到目标网络  $\theta^- \leftarrow \theta$ 
16:    end for
17: end for
```

图 3: DQN 算法流程

思考

相比于 Q-learning 算法，DQN 算法做了哪些改进？

答：主要包括：**引入深度神经网络**：Q-learning 算法中使用的是表格法来存储动作价值函数，但对于状态空间较大的问题，表格法会变得不可行。DQN 通过引入深度神经网络来近似动作价值函数，能够处理高维连续状态空间的问题。**经验回放**：传统的 Q-learning 算法每次更新时只使用当前状态和动作的信息，但这种方式可能会导致样本之间的相关性和不稳定性。DQN 采用经验回放机制，将所有的状态、动作、奖励、下一状态组成的经验存储在经验池中，然后从经验池中随机取样进行训练，可以缓解样本相关性和不稳定性的问题。**目标网络**：DQN 还引入了目标网络来解决动作价值函数的不稳定性问题。目标网络是一个与当前神经网络结构相同的网络，但其参数被固定一段时间。在训练时，使用目标网络来计算目标 Q 值，从而减少当前神经网络参数对目标 Q 值的影响，提高训练稳定性。**奖励裁剪**：在某些情况下，奖励值可能非常大或非常小，这可能会导致训练不稳定。DQN 采用奖励裁剪，将奖励值限制在一个较小的范围内，从而在一定程度上提高了训练稳定性。

为什么要在 DQN 算法中引入 $\varepsilon - greedy$ 策略？

答：目的是为了平衡探索和利用的关系。具体来说， $\varepsilon - greedy$ 策略会以一定的概率 ε 随机选择动作，以一定的概率 $1 - \varepsilon$ 选择当前状态下具有最大 Q 值的动作，从而在训练过程中保证一定的探索性，使得智能体能够尝试一些未知的状态和动作，从而获得更多的奖励。如果在训练过程中完全按照当前状态下的最大 Q 值选择动作，可能会导致智能体过于保守，无法获得更多的奖励。而如果完全随机选择动作，可能会导致智能体无法学习到更优的策略，从而影响学习效果。因此，引入 $\varepsilon - greedy$ 策略可以在探索和利用之间进行平衡，从而在训练过程中获得更好的性能。需要注意的是， $\varepsilon - greedy$ 策略中的 ε 值是一个重要的超参数，需要根据具体问题进行调整。如果 ε 值过小，可能会导致智能体无法充分探索环境；如果 ε 值过大，可能会导致智能体无法有效地利用已有的经验。因此，需要根据具体问题进行调参。

DQN 算法为什么要多加一个目标网络？

答：目标网络的作用是为了解决动作价值函数的不稳定性问题。目标网络是一个与当前神经网络结构相同的网络，但其参数被固定一段时间。在训练时，使用目标网络来计算目标 Q 值，从而减少当前神经网络参数对目标 Q 值的影响，提高训练稳定性。具体来说，当使用当前神经网络来计算目标 Q 值时，当前神经网络的参数和目标 Q 值的计算都是基于同一批数据的，这可能导致训练过程中出现不稳定的情况。而使用目标网络来计算目标 Q 值时，目标网络的参数是固定的，不会受到当前神经网络的训练过程的影响，因此可以提高训练的稳定性。同时，目标网络的更新也是基于一定的规则进行的。在每个训练步骤中，目标网络的参数被更新为当前网络的参数的加权平均值，其权重由一个超参数 τ 控制。通过这种方式，目标网络的更新过程可以更加平稳，避免了训练过程中出现剧烈的波动，从而提高了训练的效率和稳定性。因此，引入目标网络是 DQN 算法的一个重要改进，可以显著提高算法的性能和稳定性。

经验回放的作用是什么？

答：经验回放主要作用在于缓解样本相关性和不稳定性问题，提高算法的训练效率和稳定性。**缓解样本相关性问题**：在深度强化学习中，每个样本通常都是与前几个样本高度相关的。如果直接使用当前样本进行训练，可能会导致样本之间的相关性过高，从而影响算法的训练效果。经验回放机制通过从经验池中随机取样，可以打破样本之间的相关性，提高训练的效果。**缓解不稳定性问题**：在深度强化学习中，每个样本的值函数都是基于当前神经网络的参数计算的。由于神经网络的参数在每个训练步骤中都会发生变化，因此每个样本的值函数也会随之变化。这可能会导致算法的训练过程不稳定，经验回放机制可以通过随机取样的方式，减少每个训练步骤中样本值函数的变化，从而提高训练的稳定性。

DQN 算法进阶

DQN 算法虽然在很多任务中取得了不错的效果，但它仍然存在一些问题，例如 Q 值的过估计、探索效率低下等。为了解决这些问题，研究人员提出了许多改进的 DQN 算法。本章将介绍其中的一些重要改进方法，包括 Double DQN、Noisy DQN、Dueling DQN、PER DQN、C51 以及 Rainbow DQN 等。

Double DQN 算法

回顾 DQN 算法的更新公式，如式 (1) 所示。

$$\begin{aligned} Q_{\theta}(s_i, a_i) &\leftarrow Q_{\theta}(s_i, a_i) + \alpha[y_i - Q_{\theta}(s_i, a_i)] \\ y_i &= r_i + \gamma \max_{a'} Q_{\theta^*}(s_{i+1}, a') \end{aligned} \quad (1)$$

注意到目标值 y_i 中的最大化操作用来同时选择动作和评估价值，如果目标网络对部分动作的价值估计偏高，在选择动作时会进一步放大这种偏差，导致 Q 值的过估计问题。这种过估计会影响策略的学习效果，甚至导致训练不稳定。

为了解决这个问题，Double DQN 算法提出了将动作选择和动作评估分离的思路。具体来说，Double DQN 使用当前网络来选择动作，而使用目标网络来评估该动作的价值，如式 (2) 所示。

$$y_i = r_i + \gamma Q_{\theta^*}(s_{i+1}, \arg \max_a Q_{\theta}(s_{i+1}, a)) \quad (2)$$

通过这种方式，避免了同一个网络既当“裁判”又当“选手”，从而减轻了过估计的问题。

在算法流程上，Double DQN 与 DQN 基本相同，只是在计算目标 Q 值时采用了不同的方式，[完整的代码实现可参考实战部分的内容](#)。

Dueling DQN 算法

在强化学习中的很多状态下，某些动作的选择对最终的回报影响并不大，或者说某些动作之间的回报差异并不显著，例如等红灯时由于无法前进，无论选择等待还是尝试向左或右转，最终的回报都不会有太大差别。基础的 DQN 算法由于直接学习 $Q(s, a)$ 函数，可能无法有效地捕捉这种状态下动作之间的差异，从而影响学习效率和策略质量。

为了解决这个问题，Dueling DQN 算法的核心思路是将 Q 函数分解为两个独立的部分：状态价值函数 $V(s)$ 和优势函数 $A(s, a)$ ，即式 (3) 所示。

$$Q(s, a) = V(s) + A(s, a) \quad (3)$$

其中， $V(s)$ 表示在状态 s 下的整体价值，而 $A(s, a)$ 则表示在状态 s 下选择动作 a 相对于其他动作的优势。通过这种分解，即使某些动作的选择对回报影响不大，网络仍然可以通过 $V(s)$ 来学习状态的整体价值，从而提高学习效率。

具体如何实现呢？首先回顾一下 DQN 算法中的 Q 网络结构。如图 1 所示，它通常是一个基础的多层感知机，接受状态作为输入，经过若干隐藏层后输出每个动作对应的 Q 值，输出维度等于动作数。

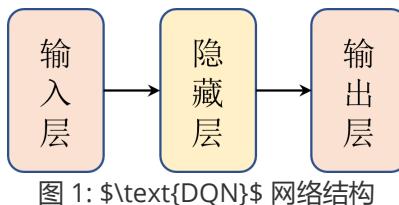


图 1: \$\\text{DQN}\$ 网络结构

在 Dueling DQN 算法中，我们对网络结构进行了修改。如图 2 所示，在输出层之前，网络被分为两个分支：一个用于计算状态价值 $V(s)$ ，另一个用于计算优势函数 $A(s, a)$ ，并且网络的前几层依然是共享的。这样，网络可以同时学习状态的整体价值和各个动作的优势，从而更好地捕捉状态下动作之间的差异。

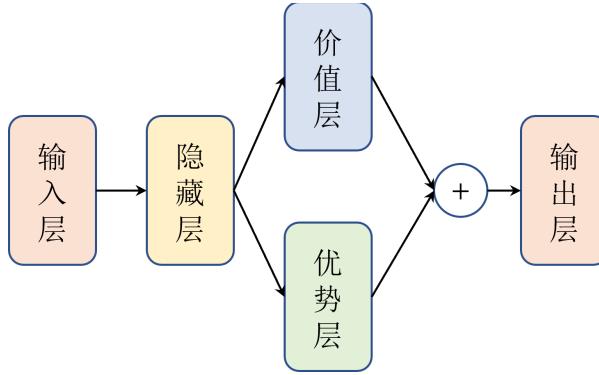


图 2: Dueling DQN 网络结构

相应地，式(3)可以改写为式(4)。

$$Q_{\theta,\alpha,\beta}(s, a) = A_{\theta,\alpha}(s, a) + V_{\theta,\beta}(s) \quad (4)$$

其中， $A_{\theta,\alpha}(s, a)$ 表示优势层的输出， $V_{\theta,\beta}(s)$ 表示价值层的输出， θ 表示共享隐藏层的参数， α 和 β 分别表示优势层和价值层的参数。

此外，为了解决不可识别性 (identifiability) 问题，Dueling DQN 通常会对优势函数进行中心化处理，即减去优势函数的均值，如式(5)所示。

$$Q_{\theta,\alpha,\beta}(s, a) = \left(A_{\theta,\alpha}(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A_{\theta,\alpha}(s, a') \right) + V_{\theta,\beta}(s) \quad (5)$$

同样地，在算法流程中，Dueling DQN 与 DQN 基本相同，只是在计算 Q 值时采用了不同的网络结构，**完整的代码实现可参考实战部分的内容**。

Noisy DQN 算法

为了平衡探索与利用的问题，DQN 算法通常采用 ε -greedy 策略来增加探索能力，即以一定概率选择随机动作，如式(6)所示。

$$a = \begin{cases} \text{random action} & \text{with probability } \varepsilon \\ \arg \max_a Q(s, a) & \text{with probability } 1 - \varepsilon \end{cases} \quad (6)$$

然而，这种方式一方面需要人为设定和退火 (annealing) ε 的值，另一方面所有状态都共用一个 ε 值，而实际上不同状态下的探索需求可能是不同的。

为了解决这些问题，Noisy DQN 算法直接在网络结构中引入了噪声层，使得网络本身具备探索能力，从而无需单纯依赖 ε -greedy 策略来增加探索性。

具体来说，Noisy DQN 的关键是使用带有噪声的线性层 (Noisy Linear Layer) 来替换传统的线性层，如式(7)所示。

$$y = (W + \Sigma_W \odot \epsilon_W)x + b + \Sigma_b \odot \epsilon_b \quad (7)$$

其中， W 和 b 分别是线性层可学习的权重和偏置， Σ_W 和 Σ_b 是对应的噪声标准差参数，也是可学习的， ϵ_W 和 ϵ_b 是从某个分布 (通常是高斯分布) 中采样的噪声， \odot 表示逐元素乘法。因此，网络不再是一个确定性的函数，而是一个随机性的，如式(8)所示。

$$Q_{\theta+\epsilon}(s, a) \quad (8)$$

同样在算法流程中，Noisy DQN 与 DQN 相同，只是在网络结构中引入了噪声层，**完整的代码实现可参考实战部分的内容**。

PER DQN 算法

在标准 DQN 算法中，我们使用经验回放（experience replay）来存储智能体与环境交互的经验。在训练时，我们会从经验回放中均匀随机采样一批样本来进行更新策略。

然而，实际上对于强化学习训练来说，不同样本的重要性是不同的，均匀随机采样的方法无法有效利用这些差异，可能会导致一些特别少但是价值可能特别高的经验或者样本没有被高效地利用到，从而影响算法的收敛性。

为此，PER DQN 算法通过引入优先经验回放（prioritized experience replay）的机制，使得经验回放中的样本可以根据其重要性来进行采样，例如价值越高的样本被采样的概率越大，从而提升算法的效果并加快学习速度。

最自然的衡量样本重要性的方法是使用 TD 误差（temporal-difference error），如式 (9) 所示。

$$\delta_i = |y_i - Q_\theta(s_i, a_i)| \quad (9)$$

在此基础上，定义样本 i 的优先级 p_i 如式 (10) 所示。

$$p_i = (\delta_i + \epsilon)^\alpha \quad (10)$$

其中， ϵ 是一个小的常数，防止优先级为零而无法被采样导致样本丢失， α 是一个超参数，用于控制优先程度，通常在 $(0, 1)$ 的区间内。当 $\alpha = 0$ 时，采样概率为均匀分布，退化为标准 DQN 算法。当 $\alpha = 1$ 时，完全按照 TD 误差进行优先采样。

这样一来，样本被采样的概率 $P(i)$ 如式 (11) 所示。

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (11)$$

重要性采样权重

尽管优先经验回放能够提升采样效率，但由于采样概率不再是均匀分布，这会引入采样偏差，影响估计的无偏性。为了解决这个问题，PER DQN 引入了重要性采样权重（importance-sampling weight），如式 (12) 所示。

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^\beta \quad (12)$$

其中， N 是经验回放中的样本总数， $P(i)$ 是样本 i 的采样概率， β 是一个超参数，用于控制重要性采样的程度，通常在 $(0, 1)$ 的区间内。当 $\beta = 0$ 时，不使用重要性采样权重；当 $\beta = 1$ 时，完全按照采样概率进行重要性采样。

归一化后，重要性采样权重如式 (13) 所示。

$$\tilde{w}_i = \frac{w_i}{\max_j w_j} \quad (13)$$

并且在更新网络参数时，使用重要性采样权重来调整损失函数，如式 (14) 所示。

$$L = \frac{1}{B} \sum_{i=1}^B \tilde{w}_i \cdot (y_i - Q_\theta(s_i, a_i))^2 \quad (14)$$

实现优先经验回放

虽然优先经验回放的思想原理比较简单，但在实际实现中存在一些挑战，通常有两种主要的数据结构可以用来实现优先经验回放，分别是基于堆（heap）的数据结构和基于树（tree）的数据结构。

基于堆的数据结构实现相对简单，但是在采样和更新优先级时的时间复杂度较高，通常接近 $O(N)$ 。而基于树的数据结构，例如 SumTree，基本上能在 $O(\log N)$ 的时间复杂度内完成采样和更新操作，更加高效。

如图 3 所示，每个父节点的值等于左右两个子节点值之和。在强化学习中，所有的样本只保存在最下面的叶子节点中，并且除了保存样本数据之外，还会保存对应的优先级，即对应叶子节点中的值（例如图中的 31,13,14 以及 8 等，也对应样本的 TD 误差）。并且根据叶子节点的值，我们从 0 开始依次划分采样区间。然后在采样中，例如这里根节点值为 66，那么我们就可以在 [0, 66) 这个区间均匀采样，采样到的值落在哪个区间中，就说明对应的样本就是我们要采样的样本。例如我们采样到了 25 这个值，即对应区间 [0, 31)，那么我们就采样到了第一个叶子节点对应的样本。

注意到，第一个样本对应的区间也是最长的，这意味着第一个样本的优先级最高，也就是 TD 误差最大，反之第四个样本的区间最短，优先级也最低。这样一来，我们就可以通过采样来实现优先经验回放的功能。

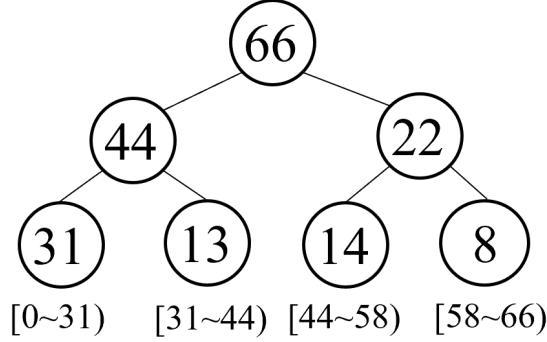


图 3: SumTree 结构

每个叶子节点保存一个样本以及对应的优先级值，而每个非叶子节点保存其两个子节点的优先级之和。这样，在采样时，我们可以通过从根节点开始，根据采样值递归地选择左子节点或右子节点，直到到达叶子节点，从而高效地实现优先经验回放。

同样在算法流程中，PER DQN 与 DQN 是相似的，只是在经验回放时采用了优先经验回放的机制，[完整的代码实现可参考实战部分的内容](#)。

C51 算法

C51 算法，又称 Categorical DQN，是一种值分布强化学习(Distributed RL)算法。经典基于值的强化学习算法如 DQN 等使用期望值对累计回报进行建模，表示为状态价值函数 $V(s)$ 或动作价值函数 $Q(s, a)$ 。然而，这种建模方式丢失了完整的分布信息，**降低了价值估计的精度和稳定性**。为了解决这个问题，C51 的使用值分布 $Z(x, a)$ 来替代原有的值函数 $Q(s, a)$ ，这样就能更好地处理值函数估计不准确以及离散动作空间的问题。

⑤ 论文链接：<https://arxiv.org/abs/1707.06887>

在之前讲到的经典强化学习算法中我们优化的其实是值分布的均值，也就是 Q 函数，但实际上由于状态转移的随机性、函数近似等原因，智能体与环境之间也存在着随机性，这也导致了最终累积的回报也会是一个随机变量，使用一个确定的均值会忽略整个分布所提供的信息。。因此，我们可以将值函数 Q 看成是一个随机变量，它的期望值就是 Q 函数，而它的方差就是 Q 函数的不确定性，公式表示如下：

$$Q^\pi(x, a) := \mathbb{E}Z^\pi(x, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(x_t, a_t)\right]$$

其中状态分布 $x_t \sim P(\cdot | x_{t-1}, a_{t-1})$, $a_t \sim \pi(\cdot | x_t)$, $x_0 = x$, $a_0 = a$.

优缺点

C51 算法的主要优点是：

- **提高价值估计的准确性：**通过建模值函数的分布而非单一 Q 值，能够捕捉回报的波动性和不确定性，从而提高估计的精度
- **提高算法的鲁棒性：**值分布建模能够让算法对噪声和环境变化更具鲁棒性，有助于减少过拟合和提高泛化能力。

C51 算法的主要缺点是：

- **计算复杂度增加：**由于需要维护和更新每个状态-动作对的完整概率分布，因此增加了算法的计算复杂度和内存开销。
- **参数调优困难：**原子分布的数量 N 需要事先确定，这个数量的选择对算法的性能有很大影响，但很难事先确定一个合适的值。
- **实践性困难：**理论上值分布是能够有效学习回报分布，但实际上由于环境的复杂性和神经网络的局限性，值分布的学习往往并不理想。

思考

DQN 算法为什么会产生 Q 值的过估计问题？

原因主要有：**数据相关性：**每次更新神经网络时，使用的都是之前采集到的数据，这些数据之间存在相关性。这导致神经网络的训练过程不稳定，可能会导致 Q 值的过估计问题。**最大化操作：**DQN 算法在更新目标 Q 值时，使用的是当前神经网络在下一个状态下具有最大 Q 值的动作。这种最大化操作可能会导致某些状态和动作的 Q 值被过估计。为了解决这个问题，可以采用一些技术，如 Double DQN 和 Dueling DQN。Double DQN 通过使用一个神经网络来估计当前状态下各个动作的 Q 值，使用另一个神经网络来计算目标 Q 值，从而减少 Q 值的过估计问题。Dueling DQN 则通过将 Q 值分解为状态值和优势值两部分，从而更准确地估计 Q 值，减少 Q 值的过估计问题。这些技术可以有效地减少 Q 值的过估计问题，提高 DQN 算法的性能。

同样是提高探索，Noisy DQN 和 ε – greedy 策略有什么区别？

ε – greedy 策略是一种基于概率的探索策略，其思想是在每个时间步中，以概率 ε 选择一个随机动作，以概率 $1 - \varepsilon$ 选择当前状态下具有最大 Q 值的动作。当随机动作被选择时，智能体有一定的概率探索新的状态和动作，从而提高探索能力。 ε – greedy 策略的优点是简单易用，但可能存在随机性过高或过低的问题，影响探索效果。

Noisy DQN 是一种基于网络权重噪声的探索策略，其思想是在神经网络中添加一定的权重噪声，以增加探索的随机性。在每个时间步中，神经网络中的权重噪声会随机地改变神经元的输出，从而改变智能体选择动作的概率分布。

Noisy DQN 的优点是能够自适应地控制探索随机性的大小，从而更加有效地提高探索能力。

策略梯度方法

策略参数化

基于策略梯度的方法首先需要将策略参数化，即直接将策略 π 参数化为 $\pi_\theta(a|s)$ ，其中 θ 是策略的参数，表示在状态 s 下选择动作 a 的概率，并且处处可微。简而言之，**参数化策略是一个处处可微的概率分布**。

然后，目标函数就可表示为 $J(\pi_\theta)$ ，即是一个关于参数 θ 的函数。为了最大化目标函数 $J(\pi_\theta)$ ，可以使用梯度上升法，即通过计算目标函数关于参数 θ 的梯度 $\nabla_\theta J(\pi_\theta)$ 来更新参数 θ ，如式 (1) 所示。

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta) \quad (1)$$

通常为了方便，会将梯度上升法转化为梯度下降法，即通过最小化目标函数的负值来更新参数 θ ，如式 (2) 所示。

$$\theta \leftarrow \theta - \alpha \nabla_\theta (-J(\pi_\theta)) \quad (2)$$

也就是说，只要能定义出目标函数 $J(\pi_\theta)$ 并求出其梯度 $\nabla_\theta J(\pi_\theta)$ ，就能利用梯度下降法来更新参数 θ ，从而使得策略 π_θ 逐步逼近最优策略 π^* 。

怎么定义关于策略的目标函数 $J(\pi_\theta)$ 呢？可以围绕**最大化长期回报**这一核心思想来展开。具体地，可以从两个角度来定义，一是基于轨迹概率密度的方式，二是基于平稳分布或状态访问分布的方式，也叫做占用测度（occupancy measure）推导，下面将分别介绍这两种推导方式。

基于轨迹推导

轨迹概率密度

智能体与环境交互过程中，首先环境会返回一个初始状态 s_0 ，然后智能体观测到当前状态并执行动作 a_0 。与此同时，环境会反馈一个奖励 r_0 ，并返回下一个状态 s_1 ，智能体再相应地执行动作 a_1 ，环境返回奖励 r_1 和下一个状态 s_2 ，如此反复进行下去，直至终止状态 s_T 。注意，这里通常假设是有终止状态的，即有限马尔可夫决策过程（Finite MDP）。

这样完整的有限步数的交互过程，称为一个**回合**（episode），回合最大步数用 T 表示（也叫作 Horizon^②）。把所有状态、动作和奖励组合起来的一个序列，称为**轨迹**（trajectory），如式 (3) 所示。

$$\tau = \{s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_T, a_T, r_T\} \quad (3)$$

② 参考自CS234，该词出现频率不算高，了解即可

为了计算轨迹产生的概率，我们可以先具体展开轨迹产生的路径。如图 1 所示，首先环境会从初始状态分布中采样出一个初始状态 s_0 ，对应的概率为 $\rho_0(s_0)$ 。然后智能体在状态 s_0 下根据策略 $\pi_\theta(a|s)$ 采样出一个动作 a_0 ，对应的概率就是策略函数对应的值，即 $\pi_\theta(a_0|s_0)$ ，接着环境根据状态转移概率 $P(s'|s, a)$ 采样出下一个状态 s_1 ，对应的概率为 $P(s_1|s_0, a_0)$ 。此时对应的轨迹序列为 $\tau = \{s_0, a_0, s_1\}$ ，根据条件概率可知，该轨迹产生的概率为 $\Pr(\tau) = \rho_0(s_0)\pi_\theta(a_0|s_0)P(s_1|s_0, a_0)$ 。

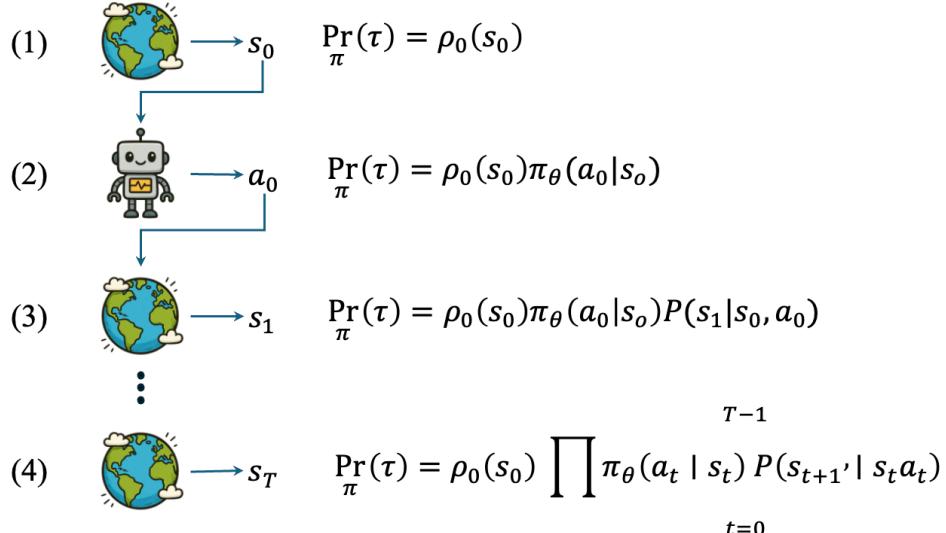


图 1 轨迹概率的计算

以此类推，可得完整轨迹的概率计算如式 (4) 所示。

$$\Pr_{\pi}(\tau) = \rho_0(s_0) \prod_{t=0}^{T-1} \pi_{\theta}(a_t | s_t) P(s_{t+1} | s_t, a_t) \quad (4)$$

可以看出，轨迹概率确实可以写成关于策略 $\pi_{\theta}(a|s)$ 或者策略参数 θ 的函数，如式 (5) 所示。

$$\Pr_{\pi}(\tau) = p_{\theta}(\tau) \quad (5)$$

给定策略 π_{θ} ，产生的轨迹可能会有很多种，因此对于式 (5) 更准确的表述是轨迹概率的分布，即轨迹概率密度。记每条的轨迹对应的回报为 $R(\tau)$ ，根据全概率公式可知，目标函数 $J(\pi_{\theta})$ 可以表示为轨迹概率密度与对应回报的乘积在所有轨迹上的积分，如式 (6) 所示。

$$J(\pi_{\theta}) = \int_{\tau} p_{\theta}(\tau) R(\tau) d\tau = \mathbb{E}_{\tau \sim p_{\theta}(\tau)} [R(\tau)] \quad (6)$$

对数导数技巧

为了最大化目标函数 $J(\pi_{\theta})$ ，可以使用梯度上升法，即通过计算目标函数关于参数 θ 的梯度 $\nabla_{\theta} J(\pi_{\theta})$ 来更新参数 θ ，如式 (7) 所示。

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\pi_{\theta}) \quad (7)$$

实际运用中为了方便，会将梯度上升法转化为梯度下降法，即通过最小化目标函数的负值来更新参数 θ ，如式 (8) 所示。

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} (-J(\pi_{\theta})) \quad (8)$$

但无论哪种方式，关键都是要计算梯度 $\nabla_{\theta} J(\pi_{\theta})$ 。根据式 (6) 可知，梯度 $\nabla_{\theta} J(\pi_{\theta})$ 的计算如式 (9) 所示。

$$\nabla_{\theta} J(\pi_{\theta}) = \nabla_{\theta} \int_{\tau} p_{\theta}(\tau) R(\tau) d\tau = \int_{\tau} \nabla_{\theta} p_{\theta}(\tau) R(\tau) d\tau \quad (9)$$

其中 $R(\tau)$ 一般不会是参数 θ 的函数，因此可以直接提到积分号外面。接下来，关键是要计算 $\nabla_{\theta} p_{\theta}(\tau)$ 。然而，根据式 (5) 可知，轨迹概率密度 $p_{\theta}(\tau)$ 是一个连乘积项，直接对其求导会比较复杂。为此，可以使用 **对数导数技巧** (log-derivative trick) 来简化计算过程。该技巧的核心思想是通过对数函数的链式法则来将导数从概率密度函数 $p_{\theta}(\tau)$ 转移到对数概率密度函数 $\log p_{\theta}(\tau)$ 上，从而简化计算。具体来说，利用对数函数的导数性质，有式 (10)。

$$\begin{aligned} \nabla_{\theta} \log p_{\theta}(\tau) &= \frac{\nabla_{\theta} p_{\theta}(\tau)}{p_{\theta}(\tau)} \\ \implies \nabla_{\theta} p_{\theta}(\tau) &= p_{\theta}(\tau) \nabla_{\theta} \log p_{\theta}(\tau) \end{aligned} \quad (10)$$

将式 (10) 代入式 (9) 中, 可得梯度 $\nabla_{\theta} J(\pi_{\theta})$ 的计算如式 (11) 所示。

$$\begin{aligned}\nabla_{\theta} J(\pi_{\theta}) &= \int_{\tau} p_{\theta}(\tau) \nabla_{\theta} \log p_{\theta}(\tau) R(\tau) d\tau \\ &= \mathbb{E}_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log p_{\theta}(\tau) R(\tau)]\end{aligned}\quad (11)$$

根据式 (4) 可知, 轨迹概率密度 $p_{\theta}(\tau)$ 中唯一与参数 θ 相关的项是策略 $\pi_{\theta}(a|s)$, 因此可以将对数概率密度函数 $\log p_{\theta}(\tau)$ 展开, 如式 (12) 所示。

$$\log p_{\theta}(\tau) = \sum_{t=0}^{T-1} \log \pi_{\theta}(a_t|s_t) + \log P(s_{t+1}|s_t, a_t) \quad (12)$$

由于环境的状态转移概率 $P(s'|s, a)$ 与参数 θ 无关, 因此对其求导结果为零, 即 $\nabla_{\theta} \log P(s_{t+1}|s_t, a_t) = 0$ 。这样一来, 式 (12) 可简化为式 (13) 所示。

$$\log p_{\theta}(\tau) = \sum_{t=0}^{T-1} \log \pi_{\theta}(a_t|s_t) \quad (13)$$

将式 (13) 代入式 (11) 中, 可得梯度 $\nabla_{\theta} J(\pi_{\theta})$ 的计算如式 (14) 所示。

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim p_{\theta}(\tau)} \left[\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) R(\tau) \right] \quad (14)$$

其中回报 $R(\tau)$ 的表示方式实际上可以有多种选择, 最简单的表示是轨迹中所有步数奖励的和, 如式 (15) 所示。

$$R(\tau) = \sum_{t=0}^{T-1} r_t \quad (15)$$

将式 (15) 代入式 (14) 中, 并交换求和次序, 可得梯度 $\nabla_{\theta} J(\pi_{\theta})$ 的计算如式 (16) 所示。

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim p_{\theta}(\tau)} \left[\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) \sum_{t=0}^{T-1} r_t \right] = \mathbb{E}_{\tau \sim p_{\theta}(\tau)} \left[\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) R_t \right] \quad (16)$$

占用测度推导

回顾状态价值相关部分, 设环境初始状态为 s_0 , 那么目标函数 $J(\pi)$ 可以表示为初始状态分布 ρ_0 与对应状态价值 $V^{\pi}(s_0)$ 的乘积在所有初始状态上的积分, 如式 (17) 所示。

$$J(\pi) = \int_{s_0} \rho_0(s_0) V^{\pi}(s_0) ds_0 = \mathbb{E}_{s_0 \sim \rho_0} [V^{\pi}(s_0)] \quad (17)$$

其中 ρ_0 是初始状态分布, $V^{\pi}(s)$ 是状态价值。状态价值指从状态 s 开始, 智能体在策略 π 指导下所能获得的未来 (折扣) 回报的期望, 定义如式 (18) 所示。

$$V^{\pi}(s) = \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right] \quad (18)$$

根据贝尔曼期望方程 (Bellman Expectation Equation) 可知, 状态价值 $V^{\pi}(s)$ 还可以通过动作价值函数表示, 如式 (19) 所示。

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(a|s)} [Q^{\pi}(s, a)] = \sum_a \pi(a|s) Q^{\pi}(s, a) \quad (19)$$

这样一来, 目标函数 $J(\pi)$ 就可以写成关于策略 $\pi(a|s)$ 或者策略参数 θ 的函数, 如式 (20) 所示。

$$J(\pi_\theta) = \int_{s_0} \rho_0(s_0) \sum_a \pi_\theta(a|s_0) Q^{\pi_\theta}(s_0, a) ds_0 \quad (20)$$

乍看初始状态分布 ρ_0 似乎与策略参数 θ 无关，因此在计算梯度 $\nabla_\theta J(\pi_\theta)$ 时可以将其视为常数项直接提到积分号外面。然而，实际上初始状态分布 ρ_0 会影响智能体后续的状态访问分布（state visitation distribution），进而影响目标函数 $J(\pi_\theta)$ 的值。

因此，在计算梯度 $\nabla_\theta J(\pi_\theta)$ 时，不能简单地将初始状态分布 ρ_0 视为常数项。为此，需要引入 **平稳分布**（stationary distribution）的概念来更好地理解状态访问分布与策略参数 θ 之间的关系。

平稳分布

在引入平稳分布概念之前，先来看一个例子。如图 2 所示，假设有一个简单的马尔可夫过程（Markov Process），包含三个状态 s_1, s_2, s_3 ，每个状态之间的转移概率如图中所示。

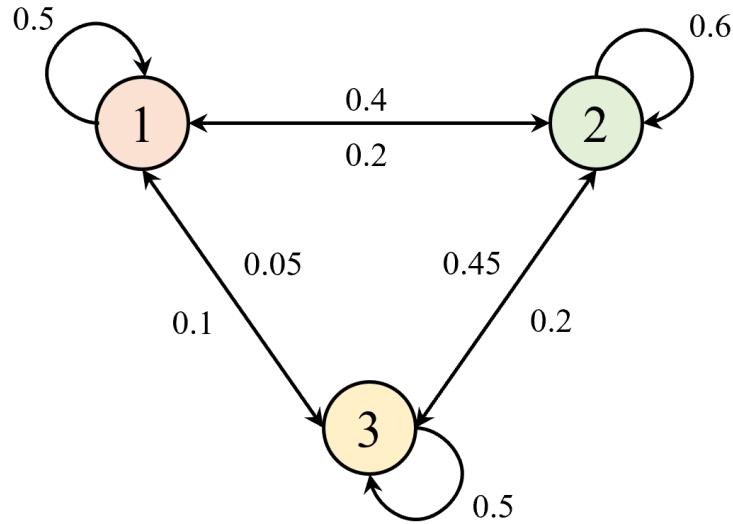


图 2 马尔可夫过程示例

从图中可以该马尔可夫过程的状态转移矩阵 P 如式 (21) 所示。

$$P = \begin{bmatrix} 0.5 & 0.4 & 0.1 \\ 0.2 & 0.6 & 0.2 \\ 0.05 & 0.45 & 0.5 \end{bmatrix} \quad (21)$$

设初始状态分布为 $\rho_0 = [0.15, 0.62, 0.23]$ ，表示初始时刻状态 s_1 的概率为 0.15，状态 s_2 的概率为 0.62，状态 s_3 的概率为 0.23。那么经过一步状态转移或者说一次状态迭代后，新的状态分布 ρ_1 可通过初始状态分布 ρ_0 与状态转移矩阵 P 相乘得到，如式 (22) 所示。

$$\begin{aligned} \rho_1 &= \rho_0 P = [0.15, 0.62, 0.23] \begin{bmatrix} 0.5 & 0.4 & 0.1 \\ 0.2 & 0.6 & 0.2 \\ 0.05 & 0.45 & 0.5 \end{bmatrix} \\ &= [0.21, 0.536, 0.254] \end{aligned} \quad (22)$$

同理，经过两次状态转移或者说两次状态迭代后，新的状态分布 ρ_2 可通过 ρ_1 与状态转移矩阵 P 相乘得到，如式 (23) 所示。

$$\begin{aligned} \rho_2 &= \rho_1 P = [0.21, 0.536, 0.254] \begin{bmatrix} 0.5 & 0.4 & 0.1 \\ 0.2 & 0.6 & 0.2 \\ 0.05 & 0.45 & 0.5 \end{bmatrix} \\ &= [0.225, 0.52, 0.255] \end{aligned} \quad (23)$$

那么经过多次状态转移或者说多次状态迭代后，状态分布会发生什么变化呢？我们可以通过编程来模拟一下，如代码 1 所示。

代码 1 状态分布迭代模拟

```
import numpy as np
pi_0 = np.array([[0.15, 0.62, 0.23]])
P = np.array([[0.5, 0.4, 0.1], [0.2, 0.6, 0.2], [0.05, 0.45, 0.5]])
for i in range(1, 10+1):
    pi_0 = pi_0.dot(P)
    print(f"第{i}次迭代后状态分布为: {np.around(pi_0, 3)}")
```

运行结果如代码 2 所示。

代码 2 状态分布迭代结果1

```
第1次迭代后状态分布为: [[0.21  0.536 0.254]]
第2次迭代后状态分布为: [[0.225 0.52  0.255]]
第3次迭代后状态分布为: [[0.229 0.517 0.254]]
第4次迭代后状态分布为: [[0.231 0.516 0.253]]
第5次迭代后状态分布为: [[0.231 0.516 0.253]]
第6次迭代后状态分布为: [[0.231 0.516 0.253]]
第7次迭代后状态分布为: [[0.232 0.516 0.253]]
第8次迭代后状态分布为: [[0.232 0.516 0.253]]
第9次迭代后状态分布为: [[0.232 0.516 0.253]]
第10次迭代后状态分布为: [[0.232 0.516 0.253]]
```

可以看出，经过多次迭代后，**状态分布逐渐趋于稳定，并最终收敛到一个固定的分布**，即 $\rho = [0.232, 0.516, 0.253]$ 。把初始状态分布改成其他的任意值，例如 $\rho_0 = [0.9, 0.05, 0.05]$ ，再运行代码 1，结果如代码 3 所示，状态分布依然会收敛到同一个固定的分布 $\rho = [0.232, 0.516, 0.253]$ 。

代码 3 状态分布迭代结果1

```
第1次迭代后状态分布为: [[0.462 0.413 0.125]]
第2次迭代后状态分布为: [[0.32  0.489 0.191]]
第3次迭代后状态分布为: [[0.267 0.507 0.225]]
第4次迭代后状态分布为: [[0.246 0.513 0.241]]
第5次迭代后状态分布为: [[0.238 0.515 0.248]]
第6次迭代后状态分布为: [[0.234 0.515 0.251]]
第7次迭代后状态分布为: [[0.233 0.516 0.252]]
第8次迭代后状态分布为: [[0.232 0.516 0.252]]
第9次迭代后状态分布为: [[0.232 0.516 0.252]]
第10次迭代后状态分布为: [[0.232 0.516 0.253]]
```

也就是说，无论初始状态分布如何变化，经过多次迭代后，状态分布最终都会收敛到同一个固定的分布 $\rho = [0.232, 0.516, 0.253]$ 。这个固定的分布就称为**平稳分布**（stationary distribution），通常用 $d^\pi(s)$ 表示，表示在策略 π 指导下，从任意初始状态 s_0 开始，经过足够长时间后，系统处于状态 s 的概率，如式 (24) 所示。

$$d^\pi(s) = \lim_{t \rightarrow \infty} Pr(s_t = s | s_0, \pi_\theta) \quad (24)$$

简单来说，它描述了系统在长期运行后，处于各状态的概率分布。需要注意的是，平稳分布的存在是有前提条件的，必须是遍历（ergodic）的马尔可夫过程，遍历包含两个性质：不可约（irreducible）和非周期（aperiodic）。不可约表示从任意状态出发，都有可能到达其他任意状态，有时也叫作连通性（communicative）；非周期表示系统不会陷入某种固定的循环模式。而通常情况下，**强化学习中的马尔可夫过程都是遍历的，因此平稳分布是存在的**。

平稳分布的存在性推导

本节内容主要从数学上来推导说明为什么平稳分布是存在的，换句话说为什么马尔可夫过程在长期运行后会收敛到一个固定的分布，即“不动点”，如式(25)所示。

$$d_{t+1}^\pi(s) = \sum_{s'} d_t^\pi(s') P(s|s', \pi_\theta), \text{且当 } d_t^\pi(s) = d^*(s) \text{ 时, } d_{t+1}^\pi(s) = d^*(s) \quad (25)$$

用矩阵的语言来表示，就是转移算子 P 存在一个不动点，如式(26)所示。

$$d^T = d^T P \quad (26)$$

两边转置，并结合矩阵转置的性质 $(AB)^T = B^T A^T$ ，可得式(27)。

$$d = P^T d \quad (27)$$

由于状态转移矩阵本身就是一个特殊的矩阵，即随机矩阵（stochastic matrix），即满足式(28)。

$$\sum_i P(s_i|s_j, a) = 1, \forall j, a, \text{ 其中 } P(s_i|s_j, a) \geq 0 \quad (28)$$

有时也写作式(29)。

$$\sum_{s'} P(s'|s, a) = 1, \forall s, a, \text{ 其中 } P(s'|s, a) \geq 0 \quad (29)$$

用矩阵语言来表示，就是每一列的元素和为1，且每个元素都非负，如式(30)所示。

$$P^T \mathbf{1} = \mathbf{1} \quad (30)$$

回顾线性代数相关知识，对于方阵 A ，如果存在一个非零向量 v 和一个标量 λ ，使得 $Av = \lambda v$ ，那么就称 λ 是矩阵 A 的一个特征值， v 是对应的右特征向量。同时，也会有左特征向量的概念，即如果存在一个非零向量 u 和一个标量 λ ，使得 $u^T A = \lambda u^T$ ， u 是对应的左特征向量。左特征向量和右特征向量表达的是同一个概念，只是左特征向量是行向量，右特征向量是列向量。

回到式(30)，可以看出，向量 $\mathbf{1}$ 是矩阵 P^T 的右特征向量，且对应的特征值为1。因此，矩阵 P 也必然存在对应的左特征向量 d ，即如式(26)和式(27)所示，也就是我们要找的平稳分布。

目标函数梯度

回顾式(20)，虽然初始状态分布 ρ_0 会影响状态访问分布，但经过多次迭代后，状态访问分布会逐渐趋于平稳分布 $d^\pi(s)$ 。因此，可以将目标函数 $J(\pi_\theta)$ 中的初始状态分布 ρ_0 替换为平稳分布 $d^\pi(s)$ ，即表示为平稳分布与对应状态价值的乘积和，如式(31)所示。

$$J(\pi_\theta) = \sum_s d^\pi(s) V^\pi(s) = \sum_s d^\pi(s) \pi_\theta(a|s) Q^\pi(s, a) \quad (31)$$

写成期望的形式，如式(32)所示。

$$J(\pi_\theta) = \mathbb{E}_{s \sim d^\pi(s)} [V^\pi(s)] = \mathbb{E}_{s \sim d^\pi(s), a \sim \pi_\theta(a|s)} [Q^\pi(s, a)] \quad (32)$$

同样地，只要能求出目标函数的梯度，就能利用梯度上升法来更新参数 θ ，从而使得策略 π_θ 逐步逼近最优，即使得目标函数的值最大化。梯度 $\nabla_\theta J(\pi_\theta)$ 的计算如式(33)所示，详细推导见下节内容。

$$\begin{aligned}
\nabla_{\theta} J(\pi_{\theta}) &= \nabla_{\theta} \sum_s d^{\pi}(s) \pi_{\theta}(a|s) Q^{\pi}(s, a) \\
&\approx \sum_s d^{\pi}(s) \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) \\
&= \sum_s d^{\pi}(s) \pi_{\theta}(a|s) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} Q^{\pi}(s, a) \\
&= \mathbb{E}_{s \sim d^{\pi}(s), a \sim \pi_{\theta}(a|s)} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s, a)]
\end{aligned} \tag{33}$$

其中近似符号 \approx 表示忽略了平稳分布 $d^{\pi}(s)$ 关于参数 θ 的导数项，即 $\nabla_{\theta} d^{\pi}(s)$ 。这种近似在实际应用中是合理的，因为平稳分布通常变化较慢，对梯度的影响较小，因此可以忽略不计。

目标函数梯度的推导

先看价值函数部分，如式 (34) 所示。

$$\begin{aligned}
\nabla_{\theta} V^{\pi}(s) &= \nabla_{\theta} \left(\sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s, a) \right) \\
&= \sum_{a \in \mathcal{A}} (\nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \nabla_{\theta} Q^{\pi}(s, a)) \text{ 分部积分法} \\
&= \sum_{a \in \mathcal{A}} \left(\nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \nabla_{\theta} \sum_{s', r} P(s', r|s, a) (r + V^{\pi}(s')) \right) \text{ 贝尔曼方程} \\
&= \sum_{a \in \mathcal{A}} \left(\nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \sum_{s', r} P(s', r|s, a) \nabla_{\theta} V^{\pi}(s') \right) \text{ 消去无关项} \\
&= \sum_{a \in \mathcal{A}} \left(\nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \sum_{s'} P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \right) \text{ 转移矩阵等价}
\end{aligned} \tag{34}$$

这样就得到了一个迭代公式，如式 (35) 所示。

$$\nabla_{\theta} V^{\pi}(s) = \sum_{a \in \mathcal{A}} \left(\nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \sum_{s'} P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \right) \tag{35}$$

其中 $V^{\pi}(s')$ 可以看作是下一个状态 s' 的价值函数，因此 $\nabla_{\theta} V^{\pi}(s')$ 也是一个类似的迭代公式。定义一个从状态 s 出发，经过 k 步转移后到达状态 x 的概率为 $\rho^{\pi}(s \rightarrow x, k)$ 的量，通常叫做状态访问序列的概率。对应的访问序列 (state visitation sequence) 表示如式 (36)。

$$s \xrightarrow{a \sim \pi_{\theta}(\cdot|s)} s' \xrightarrow{a \sim \pi_{\theta}(\cdot|s')} s'' \xrightarrow{a \sim \pi_{\theta}(\cdot|s'')} \dots \tag{36}$$

其中：

- 当 $k = 0$ 时， $\rho^{\pi}(s \rightarrow x, 0) = 1$
- 当 $k = 1$ 时， $\rho^{\pi}(s \rightarrow s', 1) = \sum_a \pi_{\theta}(a|s) P(s'|s, a)$
- 以此类推，可得 $\rho^{\pi}(s \rightarrow x, k+1) = \sum_{s'} \rho^{\pi}(s \rightarrow s', k) \rho^{\pi}(s' \rightarrow x, 1)$

为了简化，设 $\phi(s) = \sum_a \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a)$ ，那么根据式 (35) 可知， $\nabla_{\theta} V^{\pi}(s)$ 可展开为式 (37) 所示。

$$\begin{aligned}
\nabla_{\theta} V^{\pi}(s) &= \phi(s) + \sum_a \pi_{\theta}(a|s) \sum_{s'} P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \nabla_{\theta} V^{\pi}(s') \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \left(\phi(s') + \sum_{s''} \rho^{\pi}(s' \rightarrow s'', 1) \nabla_{\theta} V^{\pi}(s'') \right) \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \phi(s') + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \sum_{s''} \rho^{\pi}(s' \rightarrow s'', 1) \nabla_{\theta} V^{\pi}(s'') \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \phi(s') + \sum_{s''} \rho^{\pi}(s \rightarrow s'', 2) \nabla_{\theta} V^{\pi}(s'') \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \phi(s') + \sum_{s''} \rho^{\pi}(s \rightarrow s'', 2) \left(\phi(s'') + \sum_{s'''} \rho^{\pi}(s'' \rightarrow s''', 1) \nabla_{\theta} V^{\pi}(s''') \right) \\
&= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \phi(s') + \sum_{s''} \rho^{\pi}(s \rightarrow s'', 2) \phi(s'') + \sum_{s'''} \rho^{\pi}(s \rightarrow s''', 3) \nabla_{\theta} V^{\pi}(s''') \\
&= \dots \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \rho^{\pi}(s \rightarrow x, k) \phi(x)
\end{aligned} \tag{37}$$

接下来，回到目标函数梯度 $\nabla_{\theta} J(\pi_{\theta})$ 的计算，如式 (38) 所示。

$$\begin{aligned}
\nabla_{\theta} J(\pi_{\theta}) &= \nabla_{\theta} \sum_s d^{\pi}(s) V^{\pi}(s) \\
&= \sum_s (\nabla_{\theta} d^{\pi}(s) V^{\pi}(s) + d^{\pi}(s) \nabla_{\theta} V^{\pi}(s)) \\
&\approx \sum_s d^{\pi}(s) \nabla_{\theta} V^{\pi}(s) \quad \text{忽略 } \nabla_{\theta} d^{\pi}(s) \text{ 项} \\
&= \sum_s d^{\pi}(s) \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \rho^{\pi}(s \rightarrow x, k) \phi(x) \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \phi(x) \sum_s d^{\pi}(s) \rho^{\pi}(s \rightarrow x, k) \\
&= \sum_{x \in \mathcal{S}} d^{\pi}(x) \phi(x) \quad \text{平稳分布定义} \\
&= \sum_s d^{\pi}(s) \sum_a \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) \\
&= \sum_s d^{\pi}(s) \sum_a \pi_{\theta}(a|s) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} Q^{\pi}(s, a) \\
&= \mathbb{E}_{s \sim d^{\pi}(s), a \sim \pi_{\theta}(a|s)} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s, a)]
\end{aligned} \tag{38}$$

至此，目标函数梯度 $\nabla_{\theta} J(\pi_{\theta})$ 的推导完毕。

两种推导方式的等价性

对比基于轨迹概率密度推导出的目标函数和基于状态值函数推导出的目标函数，会发现两者形式上是类似的，区别在于一个是从时间步 t 的角度来表示，另一个是从状态 s 的角度来表示。实际上，这两个表达式确实是等价的，可以相互转换。下面将展示如何从状态值函数的表达式出发，推导出轨迹概率密度的表达式。

首先，回顾状态值函数的策略梯度表达式，如式 (39) 所示。

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{s \sim d^{\pi}(s), a \sim \pi_{\theta}(a|s)} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s, a)] \tag{39}$$

接下来，展开期望的定义，如式 (40) 所示。

$$\begin{aligned}\nabla_{\theta} J(\pi_{\theta}) &= \sum_s d^{\pi}(s) \sum_a \pi_{\theta}(a|s) \nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s, a) \\ &= \sum_s d^{\pi}(s) \sum_a \pi_{\theta}(a|s) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} Q^{\pi}(s, a)\end{aligned}\quad (40)$$

再将状态值函数 $Q^{\pi}(s, a)$ 展开成轨迹的形式，如式 (41) 所示。

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau \sim p_{\theta}(\tau|s_0=s, a_0=a)} \left[\sum_{t=0}^T r(s_t, a_t) \right] \quad (41)$$

将式 (41) 代入到式 (40) 中，如式 (42) 所示。

$$\begin{aligned}\nabla_{\theta} J(\pi_{\theta}) &= \sum_s d^{\pi}(s) \sum_a \pi_{\theta}(a|s) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} \mathbb{E}_{\tau \sim p_{\theta}(\tau|s_0=s, a_0=a)} \left[\sum_{t=0}^T r(s_t, a_t) \right] \\ &= \sum_s d^{\pi}(s) \sum_a \nabla_{\theta} \pi_{\theta}(a|s) \mathbb{E}_{\tau \sim p_{\theta}(\tau|s_0=s, a_0=a)} \left[\sum_{t=0}^T r(s_t, a_t) \right]\end{aligned}\quad (42)$$

接下来，将状态分布 $d^{\pi}(s)$ 和动作分布 $\pi_{\theta}(a|s)$ 展开成轨迹的形式，如式 (43) 所示。

$$\begin{aligned}d^{\pi}(s) &= \sum_{t=0}^T \gamma^t P(s_t = s | \pi_{\theta}) \\ \pi_{\theta}(a|s) &= P(a_t = a | s_t = s, \pi_{\theta})\end{aligned}\quad (43)$$

将式 (43) 代入到式 (42) 中，并结合轨迹概率密度的定义，最终得到轨迹概率密度的策略梯度表达式，如式 (44) 所示。

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim p_{\theta}(\tau)} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R(\tau) \right] \quad (44)$$

策略梯度通用表达式

在 GAE 论文^③ 中，提出了一种更为通用的策略梯度表达式，如式 (45) 所示。

$$g = \mathbb{E} \left[\sum_{t=0}^{\infty} \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right] \quad (45)$$

^③ <https://arxiv.org/pdf/1506.02438>

其中 Ψ_t 是一个通用的回报估计，可以根据具体的算法选择不同的形式。下面列举几种常见的 Ψ_t 形式：

- $R_t = \sum_{t=0}^{\infty} r_t$ ：表示整条轨迹的累计奖励
- $\sum_{t'=t}^{\infty} r_{t'}$ ：表示动作 a_t 之后的累计奖励
- G_t ：表示折扣回报，也是后面要讲的 REINFORCE 算法形式
- $\sum_{t'=t}^{\infty} r_{t'} - b(s_{t'})$ ：引入基线函数 $b(s_{t'})$ 来减少方差
- $Q^{\pi}(s_t, a_t)$ ：动作价值函数
- $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$ ：优势函数，表示动作相对于平均水平的好坏
- $r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$ ：时序差分误差 (TD error)

公式 (45) 中的 $\nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$ 用于衡量策略参数 θ 变化对动作选择概率的影响，相当于一个演员 (Actor) 的角色，而 Ψ_t 则提供了一个衡量动作好坏的信号，相当于一个评论家 (Critic) 的角色。通过选择不同的 Ψ_t 形式，即对动作不同的评价方式，可以得到不同的算法，例如 REINFORCE、PPO 等，这就是比较流行的演员-评论家 (Actor-Critic) 方法的基本思想。

策略函数建模

前面讲到，在策略梯度方法中，策略函数 $\pi_\theta(a|s)$ 是直接对策略进行参数化的函数，参数为 θ ，并且必须是一个概率分布。如何对策略函数进行建模呢？对于不同类型的动作空间（离散和连续），策略函数的建模方式也有所不同。对于离散动作空间，可以用多项式分布（Categorical Distribution）进行建模，而对于连续动作空间，则可以用高斯分布（Gaussian Distribution）。下面将分别展开说明。

离散动作空间

对于离散动作空间，例如动作集合 $\mathcal{A} = \{a_0, a_1, a_2, \dots, a_{n-1}\}$ ，通常使用多项式分布（Categorical Distribution）进行建模，而多项式分布需要一个概率向量 $p_\theta(s) = \{p_0, p_1, \dots, p_{n-1}\}$ 作为参数，其中每个元素 p_i 表示在状态 s 下选择动作 a_i 的概率，且满足 $\sum_{i=0}^{n-1} p_i = 1$ 。从多项式分布采样动作的过程如式（46）

$$a \sim \text{Categorical}(p_\theta(s)) \quad (46)$$

然而，神经网络一般不能直接输出满足概率分布要求的向量，因此需要对神经网络的输出进行转换，转换的方式必须是可微分的，以便能够计算梯度并进行优化。为此，可以采用软最大化（Softmax）函数来实现这一转换，如式（47）所示。

$$\pi_\theta(a|s) = \frac{\exp(f_\theta(s, a))}{\sum_{a'} \exp(f_\theta(s, a'))} = \frac{e^{z_a}}{\sum_{a'} e^{z_{a'}}} \quad (47)$$

其中 $z = f_\theta(s, a)$ 通常是神经网络的输出，表示在状态 s 下选择动作 a 的偏好值（preference），也写作 logits。

对应的梯度计算如式（48）所示。

$$\begin{aligned} \nabla_\theta \log \pi_\theta(a|s) &= \nabla_\theta f_\theta(s, a) - \sum_{a'} \pi_\theta(a'|s) \nabla_\theta f_\theta(s, a') \\ &= \nabla_\theta z_a - \sum_{a'} \pi_\theta(a'|s) \nabla_\theta z_{a'} \end{aligned} \quad (48)$$

再代入到前面推导出的目标函数梯度（例如式（33））中，即可得到完整的梯度计算公式。另外，Softmax 的梯度也可以通过链式法则和雅可比矩阵来计算，如式（49）所示，感兴趣的读者可自行了解。

$$\begin{aligned} \frac{\partial \pi_\theta(a|s)}{\partial z_{a'}} &= \frac{\partial y_a}{\partial z_{a'}} = y_a (\delta_{aa'} - y_{a'}) \\ &= \begin{cases} y_a(1 - y_a), & \text{if } a' = a \\ -y_a y_{a'}, & \text{if } a' \neq a \end{cases} \end{aligned} \quad (49)$$

注意到，由于使用了指数函数，如果某个动作的得分 z_i 较高，对应的 $\exp(z_i)$ 就会成倍增加，换句话说，这会让策略更倾向于“高分动作”。然而，如果得分 z 过大，可能会导致指数函数的输出超出计算机的表示范围，从而导致数值不稳定的问题。

为了解决这个问题，通常会在计算 Softmax 函数时，对所有的得分 z 减去一个常数 $\max(z)$ ，这样可以避免指数函数的输入过大，同时不会改变概率分布的相对关系，如式（50）所示。

$$\pi_\theta(a|s) = \frac{\exp(f_\theta(s, a) - \max_{a'} f_\theta(s, a'))}{\sum_{a'} \exp(f_\theta(s, a') - \max_{a''} f_\theta(s, a''))} \quad (50)$$

为帮助理解，我们使用 Numpy 模块实现一个 Softmax 函数并从中采样动作，如代码 4 所示。

代码 4 Softmax 函数 Numpy 实现

```
import numpy as np

# softmax 函数
def softmax(z):
    z = np.array(z)
```

```

e = np.exp(z - np.max(z))
return e / e.sum()

# softmax 函数 梯度
def softmax_grad(y):
    # 输入 y 是 softmax 输出向量
    return np.diag(y) - np.outer(y, y)

# logits -> 概率
logits = [2.0, 1.0, 0.1]
probs = softmax(logits)

# 从 Categorical 分布中采样动作
action = np.random.choice(len(probs), p=probs)

# 对数概率
log_prob = np.log(probs[action])

print(f"动作索引: {action}")
print(f"动作概率: {probs[action]:.3f}")
print(f"log π(a|s): {log_prob:.3f}")

```

运行结果如代码 5 所示。

代码 5 Softmax 函数运行结果

```

动作索引: 0
动作概率: 0.659
log π(a|s): -0.417

```

接下来，我们使用 PyTorch 模块实现一个基于 Softmax 函数的策略网络，并计算对应的梯度，如代码 6 所示。

代码 6 Softmax 策略网络 PyTorch 实现

```

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# 定义策略网络
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(PolicyNetwork, self).__init__()
        self.fc = nn.Linear(state_dim, action_dim)

    def forward(self, x):
        return F.softmax(self.fc(x), dim=-1)

# 创建策略网络实例
state_dim = 4 # 状态维度
action_dim = 3 # 动作维度
policy_net = PolicyNetwork(state_dim, action_dim)
optimizer = optim.Adam(policy_net.parameters(), lr=0.01)

# 示例状态输入
state = torch.FloatTensor([1.0, 0.5, -0.5, 2.0])
# 前向传播计算动作概率
action_probs = policy_net(state)

```

```

# 从动作概率中采样动作
action_dist = torch.distributions.Categorical(action_probs)
action = action_dist.sample()
log_prob = action_dist.log_prob(action)

# 假设一个示例回报
reward = torch.FloatTensor([1.0])
# 计算损失(负的策略梯度)
loss = -log_prob * reward
# 反向传播计算梯度
optimizer.zero_grad()
loss.backward()
optimizer.step()

print(f"动作索引: {action.item()}")
print(f"动作概率: {action_probs[action].item():.3f}")
print(f"log π(a|s): {log_prob.item():.3f}")

```

运行结果如代码 7 所示。

代码 7 Softmax 策略网络运行结果

```

动作索引: 2
动作概率: 0.123
log π(a|s): -2.095

```

连续动作空间

连续动作空间的策略建模通常使用高斯分布 (Gaussian Distribution)。根据动作维度的不同，高斯分布可以分为标量高斯分布和向量高斯分布两种情况。具体来讲，当动作维度为 1 维，即 $a \in \mathbb{R}$ 时，表示一个连续动作，例如机器人推力大小，此时用标量高斯分布建模。当动作维度为多维，即动作集合 $\mathcal{A} \subseteq \mathbb{R}^n$ 时，表示多个动作同时输出，例如控制多个电机，此时用向量高斯分布。两种高斯分布最后推导出的形式是类似的，初学者可以先理解标量高斯分布，再类比理解向量高斯分布，下面分别进行说明。

标量高斯分布

在标量高斯分布中，需要将均值 $\mu_\theta(s)$ 和协方差 $\sigma_\theta(s)$ 作为参数，其中均值表示在状态 s 下动作的期望值，协方差表示动作的分布范围和形状。从高斯分布采样动作的过程如式 (51) 所示。

$$a \sim \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)) \quad (51)$$

对应的策略函数如式 (52) 所示，注意这里为了展示方便，隐去了等式右边的参数 θ 和状态 s 。

$$\pi_\theta(a|s) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right) \quad (52)$$

对应的概率公式，如式 (53) 所示。

$$\log \pi_\theta(a|s) = -\frac{1}{2} \left(\frac{(a-\mu(s))^2}{\sigma^2(s)} + \log \sigma^2(s) + \log(2\pi) \right) \quad (53)$$

利用偏微分，分别求出对 μ 和 σ 的梯度，如式 (54) 所示。

$$\frac{\partial \log \pi_\theta(a|s)}{\partial \mu} = \frac{(a-\mu)}{\sigma^2}, \quad \frac{\partial \log \pi_\theta(a|s)}{\partial \sigma} = \frac{(a-\mu)^2}{\sigma^3} - \frac{1}{\sigma} \quad (54)$$

对于 σ 的梯度计算，通常使用 $\omega = \log \sigma$ 进行参数化(简单理解就是将协方差转成带 \log 的形式)，这样可以确保 σ 始终为正值。根据链式法则，有式 (55)。

$$\nabla \theta \log \pi_\theta(a|s) = \frac{\partial \log \pi_\theta(a|s)}{\partial \mu} \nabla_\theta \mu_\theta(s) + \frac{\partial \log \pi_\theta(a|s)}{\partial \omega} \nabla_\theta \omega_\theta(s) \quad (55)$$

其中由于 $\frac{\partial \sigma}{\partial \omega} = \sigma$, , 结合式 (54), 可得式 (56)。

$$\begin{aligned} \frac{\partial \log \pi_\theta(a|s)}{\partial \omega} &= \frac{\partial \log \pi_\theta(a|s)}{\partial \sigma} \frac{\partial \sigma}{\partial \omega} \\ &= \frac{\partial \log \pi_\theta(a|s)}{\partial \sigma} \sigma \\ &= \left(\frac{(a - \mu)^2}{\sigma^2} - 1 \right) \end{aligned} \quad (56)$$

向量高斯分布

对于多维连续动作空间, 例如 $a \in \mathbb{R}^d$, 通常使用向量高斯分布进行建模。与标量高斯分布类似, 需要将均值向量 $\mu_\theta(s) \in \mathbb{R}^d$ 和协方差矩阵 $\sigma_\theta(s) \in \mathbb{R}^{d \times d}$ 作为参数, 其中均值向量表示在状态 s 下每个动作维度的期望值, 协方差矩阵表示动作各维度之间的相关性和分布范围。从高斯分布采样动作的过程如式 (57) 所示。

$$a \sim \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)) \quad (57)$$

相应地, 策略函数如式 (58) 所示。

$$\begin{aligned} \pi_\theta(a|s) &= \frac{1}{(2\pi)^{d/2} |\sigma|^{1/2}} \exp \left(-\frac{1}{2} (a - \mu)^T \sigma^{-1} (a - \mu) \right) \\ &= \prod_{i=1}^d \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp \left(-\frac{(a_i - \mu_i)^2}{2\sigma_i^2} \right) \quad \text{展开成标量形式} \end{aligned} \quad (58)$$

其中 $|\sigma|$ 表示协方差矩阵的行列式。此时策略对应的概率密度公式, 如式 (59) 所示。

$$\begin{aligned} \log \pi_\theta(a|s) &= -\frac{1}{2} ((a - \mu(s))^T \sigma^{-1}(s) (a - \mu(s)) + \log |\sigma(s)| + d \log(2\pi)) \\ &= \sum_{i=1}^d \log \mathcal{N}(a_i | \mu_i(s), \sigma_i^2(s)) \\ &= -\frac{1}{2} \sum_{i=1}^d \left(\frac{(a_i - \mu_i(s))^2}{\sigma_i^2(s)} + \log \sigma_i^2(s) + \log(2\pi) \right) \quad \text{展开成标量形式} \end{aligned} \quad (59)$$

再求梯度, 如式 (60) 所示。

$$\nabla_\theta \log \pi_\theta(a|s) = \sum_{i=1}^n \left(\frac{(a_i - \mu_{\theta,i}(s))}{\sigma_{\theta,i}^2(s)} \nabla_\theta \mu_{\theta,i}(s) + \left(\frac{(a_i - \mu_{\theta,i}(s))^2}{\sigma_{\theta,i}^3(s)} - \frac{1}{\sigma_{\theta,i}^2(s)} \right) \nabla_\theta \sigma_{\theta,i}(s) \right) \quad (60)$$

可以看出, 式 (60) 与标量高斯分布的梯度计算 (式 (55)) 是类似的, 只不过是对每个动作维度分别计算梯度, 然后求和。

为帮助理解, 同样使用 `Numpy` 模块来实现高斯分布的采样和梯度求解, 如代码 8 所示。

代码 8 高斯分布 Numpy 实现

```
import numpy as np

def gaussian_policy_and_grads(mu, log_std):
    """
    mu, log_std: np.ndarray, 形状 [B, d] 或 [d] 或标量。
    返回:
        a: 采样动作 a ~ N(mu, sigma^2)
        logp: [B] 或标量, log pi(a|s)
```

```

dmu: ∂logπ/∂μ
dlogstd: ∂logπ/∂logσ
"""

mu = np.asarray(mu)
log_std = np.asarray(log_std)
std = np.exp(log_std)

# 1 采样动作 (重参数化技巧)
eps = np.random.randn(*mu.shape)      # ε ~ N(0, I)
a = mu + std * eps                   # a = μ + σ·ε

# 2 计算 log π(a|s)
quad = ((a - mu) / std) ** 2
logp_dims = -0.5 * (quad + 2 * log_std + np.log(2 * np.pi))
logp = np.sum(logp_dims, axis=-1) if logp_dims.ndim > 0 else logp_dims

# 3 计算梯度
var = std ** 2
dmu = (a - mu) / var                # ∂logπ/∂μ
dlogstd = ((a - mu) ** 2 / var) - 1.0    # ∂logπ/∂logσ

return a, logp, dmu, dlogstd

# ===== 示例 =====
# 批量 = 3, 动作维度 = 2
mu = np.array([[0.3, -0.1],
               [0.8, 0.1],
               [0.1, -0.1]])

log_std = np.array([[[-0.7, -0.7],
                     [-0.7, -0.7],
                     [-0.7, -0.7]]])    # log σ ≈ -0.7 → σ ≈ 0.496

a, logp, dmu, dlogstd = gaussian_policy_and_grads(mu, log_std)

print("采样动作 a:\n", np.round(a, 3))
print("log π(a|s):\n", np.round(logp, 4))
print("∂logπ/∂μ:\n", np.round(dmu, 3))
print("∂logπ/∂logσ:\n", np.round(dlogstd, 3))

```

运行结果如代码 9 所示。

代码 9 高斯分布运行结果

```

采样动作 a:
[[ 0.231 -0.317]
 [ 0.991  0.122]
 [ 0.067  0.118]]

log π(a|s):
[-1.1136 -1.0822 -1.1314]

∂logπ/∂μ:
[[ 0.278  1.111]
 [-0.786 -0.183]
 [ 0.134 -0.914]]

```

```

 $\frac{\partial \log \pi}{\partial \log \sigma}:$ 
[[-0.923 -0.198]
 [-0.382 -0.932]
 [-0.928 -0.507]]

```

接下来，我们使用 PyTorch 模块来实现一个基于高斯分布的策略网络，并计算对应的梯度，如代码 10 所示。

代码 10 高斯分布策略网络 PyTorch 实现

```

import torch
import torch.nn as nn
import torch.nn.functional as F

class GaussianPolicy(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.fc1 = nn.Linear(state_dim, 128)
        self.fc_mu = nn.Linear(128, action_dim)
        self.fc_logstd = nn.Linear(128, action_dim)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        mu = self.fc_mu(x)
        log_std = self.fc_logstd(x).clamp(-20, 2) # 避免数值不稳定
        std = log_std.exp()
        return mu, std

    def get_action(self, state):
        mu, std = self.forward(state)
        dist = torch.distributions.Normal(mu, std)
        action = dist.sample() # 采样动作
        log_prob = dist.log_prob(action).sum(dim=-1)
        return action, log_prob

```

REINFORCE 算法

REINFORCE 算法，又称为蒙特卡洛策略梯度（Monte Carlo Policy Gradient）算法，是最早提出的一种策略梯度方法。它的核心思想是通过采样轨迹来估计策略梯度，从而更新策略参数 θ 。回顾基于轨迹概率密度推导出的目标函数梯度式 (14)，在 REINFORCE 算法中，通常会使用回报 G_t (带折扣的未来奖励和) 来替代轨迹回报 $R(\tau)$ ，如式 (61) 所示。

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim p_{\theta}(\tau)} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) G_t \right] \quad (61)$$

使用蒙特卡洛方法来估计梯度，即通过多次采样轨迹 $\tau^{(i)}$ 来计算梯度的近似值，如式 (62) 所示。

$$\nabla_{\theta} J(\pi_{\theta}) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t^{(i)} | s_t^{(i)}) G_t^{(i)} \quad (62)$$

在实际应用中，REINFORCE 算法的流程如图 3 所示。首先，初始化策略参数 θ ，然后在每个迭代周期中，采样 N 条轨迹 $\{\tau^{(i)}\}_{i=1}^N$ (这个过程称为 rollout，出于简便图中只展示了一条轨迹的采样)，计算每条轨迹的回报 $G_t^{(i)}$ ，最后根据式 (62) 计算梯度并更新策略参数 θ 。

REINFORCE 算法

```
1: 初始化策略参数  $\theta$ 
2: for 迭代次数 = 1,  $M$  do
3:   根据策略  $\pi_\theta$  采样轨迹:  $\tau = \{s_0, a_0, r_0, \dots, s_T, a_T, r_T\}$ 
4:   for 时步  $t = 0, 1, \dots, T - 1$  do
5:     计算回报  $G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} r_k$ 
6:     更新策略  $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi_\theta(a_t | s_t)$ 
7:   end for
8: end for
```

图 3 REINFORCE 算法流程

小结

本文首先介绍策略参数化的基本概念，即通过参数化策略函数 $\pi_\theta(a|s)$ 来直接表示策略，读者须记住参数化策略是一个处处可微的分布。

然后通过基于轨迹密度概率和占用测度两种方式推导出策略梯度的目标函数及梯度表达式，分别为式 (14) 和式 (33)。两种推导方式是等价的，前者更直观易懂，后者更严谨且便于扩展。

在占用测度推导中，读者须谨记平稳分布在强化学习问题中是一定存在的，即随着迭代次数增加，状态分布会收敛到一个固定的分布，这个分布与初始状态分布无关。

接着，介绍了如何对策略函数进行建模，分别针对离散动作空间和连续动作空间，使用多项式分布和高斯分布进行建模，对于入门者来说，只需掌握相关的公式形式以及代码实现即可，不需要深入理解对应分布的梯度计算细节。

然后，介绍了 REINFORCE 算法的基本原理和实现流程，作为策略梯度方法的入门算法，读者需要理解其核心思想，即通过采样轨迹来估计策略梯度，并使用蒙特卡洛方法进行优化。但由于这类纯策略梯度方法使用率较低，可更多地将其作为理解更复杂算法的理论基础，而不用过于关注其实际应用和代码实现部分。

思考

为什么使用 Softmax 函数来建模离散动作空间的策略？

主要考虑以下因素：

- 该函数是可微分的，这是计算梯度的必要条件
- 一方面可以确保概率为正数，即对于任意神经网络的输出 z_i ， $\exp(z_i) > 0$ ，另一方面可以将任意实数映射到 $[0, 1]$ 之间，并且所有动作的概率之和为 1，符合概率分布的要求
- 指数函数能够放大差距，如果某个动作的得分 z_i 较高，对应的 $\exp(z_i)$ 就会成倍增加，从而让策略更倾向于“高分动作”。这样一来，策略能够更有效地利用当前的知识，选择更优的动作，降低无意义的探索，让回报更加稳定。然而这一点并非总是好事，过度放大差距可能导致探索不足，容易收敛到局部最优。因此实际运用时需要辩证看待。

REINFORCE 算法是无偏的吗？为什么？

REINFORCE 算法是无偏的，因为它使用了蒙特卡洛方法来估计策略梯度，所得到的梯度估计是对真实梯度的无偏估计。然而，由于使用了采样方法，估计的方差可能较大，这可能会影响学习的稳定性和效率。核心思想是直接对策略进行参数化，并通过梯度上升法来优化策略参数，从而最大化预期回报。

如何改进 REINFORCE 算法以提高样本效率？

可以通过以下几种方法来改进 REINFORCE 算法以提高样本效率: 1) 使用基线函数来减少梯度估计的方差; 2) 采用时间差分 (TD) 方法来替代蒙特卡洛估计, 从而减少对完整轨迹的依赖; 3) 使用经验回放 (Experience Replay) 技术来重用过去的经验数据; 4) 结合价值函数近似方法, 如 Actor-Critic 方法, 以同时学习策略和价值函数。基本思想是通过参数化策略, 并利用梯度上升法来优化策略参数, 从而最大化预期回报。核心思想是通过参数化策略, 并利用梯度上升法来优化策略参数, 从而最大化预期回报。

基于价值和基于策略的算法各有什么优缺点?

前者的优点有: **简单易用**: 通常只需要学习一个值函数, 往往收敛性也会更好。保守更新: 更新策略通常是隐式的, 通过更新价值函数来间接地改变策略, 这使得学习可能更加稳定。缺点有: **受限于离散动作**; **可能存在多个等价最优策略**: 当存在多个等效的最优策略时, 基于价值的方法可能会在它们之间不停地切换。后者的优点有: 直接优化策略: 由于这些算法直接操作在策略上, 所以它们可能更容易找到更好的策略; 适用于连续动作空间; **更高效的探索**: 通过调整策略的随机性, 基于策略的方法可能会有更高效的探索策略。缺点有: **高方差**: 策略更新可能会带来高方差, 这可能导致需要更多的样本来学习。**可能会收敛到局部最优**: 基于策略的方法可能会收敛到策略的局部最优, 而不是全局最优, 且收敛较缓慢。在实践中, 还存在结合了基于价值和基于策略方法的算法, 即 Actor-Critic 算法, 试图结合两者的优点来克服各自的缺点。选择哪种方法通常取决于具体的应用和其特点。

确定性策略与随机性策略的区别?

对于同一个状态, 确定性策略会给出一个明确的、固定的动作, 随机性策略则会为每一个可能的动作 (legal action) 提供一个概率分布。前者在训练中往往需要额外的探索策略, 后者则只需要调整动作概率。但前者相对更容易优化, 因为不需要考虑所有可能的动作, 但也容易受到噪声的影响。后者则相对更加鲁棒, 适用面更广, 因为很多的实际问题中, 我们往往无法得到一个确定的最优策略, 而只能得到一个概率分布, 尤其是在博弈场景中。

马尔可夫平稳分布需要满足什么条件?

状态连通性: 从任何一个状态可以在有限的步数内到达另一个状态; **非周期性**: 由于马尔可夫链需要收敛, 那么就一定不能是周期性的。

REINFORCE 算法会比 Q-learning 算法训练速度更快吗? 为什么?

Actor-Critic 算法

基于价值的算法如 DQN 系列算法虽然在很多任务中取得了不错的效果，但由于其只能处理确定性策略，且难以适配连续动作空间，因此在某些复杂任务中表现不佳。而纯策略梯度算法如 REINFORCE 算法虽然在一定程度上解决了这些问题，但其高方差和低采样效率的问题，往往难以在复杂环境中取得良好的效果。为了解决这些问题，研究人员提出了结合策略梯度和值函数的方法，即 Actor-Critic 算法，即不仅将策略函数进行参数化，同时也将值函数进行参数化，从而兼顾两者的优点。

纯策略梯度算法的缺点

纯策略梯度算法虽然通过直接对策略进行参数化，解决了基于价值算法难以适配连续动作空间和随机策略的问题，但也带来了新的挑战。回顾策略梯度算法通用的目标函数，如式 (1) 所示。

$$\nabla_{\theta} J(\theta) \propto \mathbb{E}_{\pi_{\theta}} [\Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)] \quad (1)$$

其中 Ψ_t 是某种形式的回报估计，在纯策略梯度算法中通常使用蒙特卡洛估计，即 $\Psi_t = G_t$ ，用来评价策略产生的轨迹或状态-动作对的好坏。

然而，由于蒙特卡洛估计本身的高方差特性，导致策略梯度估计也会具有较高的方差，从而影响训练的稳定性和收敛速度。而且，每次更新策略参数时都需要采样完整的轨迹，这进一步降低了采样效率，尤其是在复杂环境中，可能需要大量的样本才能获得可靠的梯度估计。此外，奖励稀疏的环境中，蒙特卡洛估计可能会导致梯度估计不准确，从而影响策略的改进。

Actor-Critic 原理

为了兼顾策略梯度算法的灵活性和基于价值算法的高效性，Actor-Critic 算法应运而生，即将值估计的这部分工作交给一个独立的网络（Critic），而策略部分（Actor）则专注于策略的优化。这样不仅可以利用值函数来提供更稳定的梯度估计，还能提高采样效率，从而在复杂任务中取得更好的效果。

如图 1 所示，Actor 与环境交互采样生成轨迹样本，同时 Critic 网络则利用这些样本来估计当前状态或状态-动作对的价值。更新时，Critic 网络通过时序差分方法来更新其参数，而 Actor 则利用 Critic 提供的价值估计来指导策略的更新。

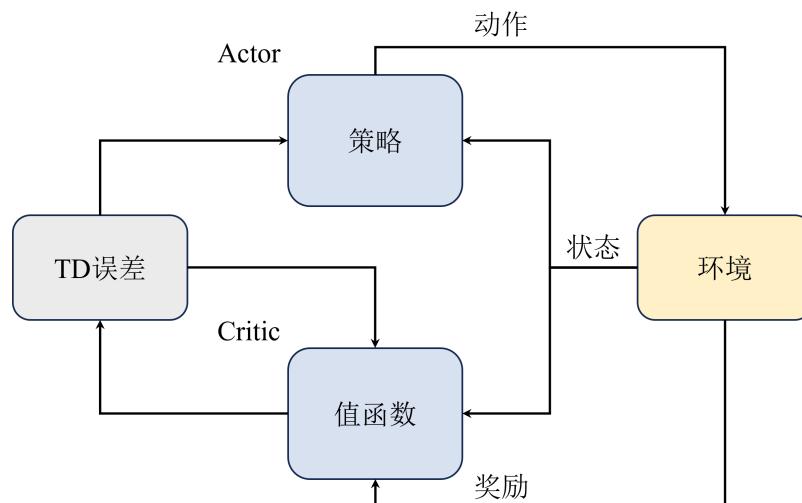


图 1: Actor-Critic 算法架构

注意，Actor-Critic 更多地是一种框架或架构，而不是具体的算法，不同算法的实现形式可能会有所不同，但都遵循 Actor-Critic 的基本思想。

例如，对于值估计部分（Critic）可以有多种形式，最基础的有两种：一种是使用状态价值函数 $V^\pi(s_t)$ 来估计当前状态的价值，另一种是使用状态-动作值函数 $Q^\pi(s_t, a_t)$ 来估计当前状态-动作对的价值。

使用状态价值函数来表示 Actor-Critic 算法的形式通常被称为 Value Actor-Critic 算法，如式 (2) 所示。

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta} [V_\omega(s_t) \nabla_\theta \log \pi_\theta(a_t | s_t)] \quad (2)$$

其中 ω 表示值函数的参数。在参数更新方面，对于策略网络（Actor）的参数 θ ，更新方式跟纯策略梯度算法类似，如式 (3) 所示。

$$\theta \leftarrow \theta + \alpha V_\omega(s_i) \nabla_\theta \log \pi_\theta(a_i | s_i) \quad (3)$$

对于值函数网络（Critic），可以先计算时序差分误差的梯度表达式，然后利用该误差来更新值函数的参数，如式 (4) 所示。

$$\nabla_\omega L(\omega) = (r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t)) \nabla_\omega V_\omega(s_t) \quad (4)$$

在式 (4) 基础上，我们可以通过梯度下降的方法来更新值函数的参数 ω ，如式 (5) 所示。

$$\begin{aligned} y_i &= r_i + \gamma V_\omega(s_{i+1}) - V_\omega(s_i) \\ \omega &\leftarrow \omega + \beta y_i \nabla_\omega V_\omega(s_i) \end{aligned} \quad (5)$$

注意，跟 DQN 系列算法类似，在这里的参数更新表达式我们使用下标 i 来表示从经验回放缓冲区中采样的样本数据，以区别于与环境交互时的时间步 t 。

使用状态-动作值函数来表示 Actor-Critic 算法的形式通常被称为 Q Actor-Critic 算法，如式 (6) 所示。

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta} [Q_\omega(s, a) \nabla_\theta \log \pi_\theta(a | s)] \quad (6)$$

同样地，Critic 网络的参数 ω 也可以通过时序差分方法来更新，其梯度表达式如式 (7) 所示。

$$\nabla_\omega L(\omega) = \mathbb{E}_{\pi_\theta} [(r_t + \gamma Q_\omega(s_{t+1}, a_{t+1}) - Q_\omega(s_t, a_t)) \nabla_\omega Q_\omega(s_t, a_t)] \quad (7)$$

A2C 算法

与纯策略梯度算法中使用蒙特卡洛估计回报相比，使用状态价值 $V^\pi(s_t)$ 或状态-动作价值函数 $Q^\pi(s_t, a_t)$ 来估计当前状态或状态-动作对的价值，虽然能在一定程度上缓解纯策略梯度算法的高方差问题，但仍然存在一些不足。

为了进一步提高梯度估计的稳定性和准确性，我们可以引入优势函数（advantage function）来改进 Actor-Critic 算法，这就是 Advantage Actor-Critic（A2C）算法的基本思想。

优势函数 $A^\pi(s_t, a_t)$ 定义如式 (8) 所示。

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \quad (8)$$

其中 $Q^\pi(s_t, a_t)$ 表示在状态 s_t 下采取动作 a_t 的预期回报，而 $V^\pi(s_t)$ 则表示在状态 s_t 下的平均预期回报。优势函数 $A^\pi(s_t, a_t)$ 衡量了在给定状态下选择特定动作相对于平均水平的优势。

为什么引入优势函数能提高梯度估计的稳定性呢？这是因为优势函数通过减去一个基线（通常选择状态价值函数 $V^\pi(s_t)$ 作为基线），从而减少了梯度估计的方差。具体来说，原先对于每一个状态-动作对只能以自己为参照物估计，现在可以用平均水平作为参照物估计了，这样就能减少方差。

举例来说，小明在上次数学考试中得了 85 分，这次考试他得了 90 分。单看这次考试的成绩，似乎小明进步了 5 分。但是如果由于上次考试全班的平均分只有 75 分，而这次考试比较容易，全班的平均分达到了 100 分，就会发现相对于全班的平均水平，小明这次考试实际上退步了 20 分（从领先 10 分到落后 10 分）。这就是优势函数的作用，通过引入基线，我们能够更准确客观地评估小明的表现。

类似地，引入优势函数后的 A2C 算法的目标函数如式 (9) 所示。

$$\nabla_{\theta} J(\theta) \propto \mathbb{E}_{\pi_{\theta}} [A^{\pi}(s_t, a_t) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)] \quad (9)$$

多进程

在讲 A3C 算法之前，我们先来了解一下多进程训练的基本思想。在实战中，会使用 `multiprocessing` 模块或者 `ray` 等分布式计算框架来实现多进程训练，如代码 1 所示。

代码 1: 使用 Ray 实现多进程计算

```
import time
import ray

# 模拟耗时任务
def slow_square(x):
    time.sleep(1)
    return x * x

# Ray 版本
@ray.remote
def ray_square(x):
    time.sleep(1)
    return x * x

def run_single(nums):
    """单进程串行执行"""
    results = []
    for x in nums:
        results.append(slow_square(x))
    return results

def run_ray(nums):
    """Ray 并行执行"""

    futures = [ray_square.remote(x) for x in nums]
    results = ray.get(futures)

    ray.shutdown()
    return results

if __name__ == "__main__":
    nums = list(range(1, 9))
    print(f"任务数量: {len(nums)}")

    # --- 单进程 ---
    t0 = time.time()
    res_single = run_single(nums)
    t1 = time.time()
    print(f"单进程耗时: {t1 - t0:.2f} 秒")
```

```

# --- Ray 并行 ---
ray.init(ignore_reinit_error=True, num_cpus=8)
t2 = time.time()
res_ray = run_ray(nums)
t3 = time.time()
print(f"Ray 并行耗时: {t3 - t2:.2f} 秒")

```

执行结果如代码 2 所示。

代码 2: 多进程计算执行结果

```

任务数量: 8
单进程耗时: 8.01 秒
Ray 并行耗时: 2.69 秒

```

从结果可以看出，使用多进程并行计算显著减少了总的计算时间。

传统的强化学习算法通常是单进程串行训练，即一个智能体与环境交互并更新其策略和价值函数参数。然而，这种方式在某些情况下可能效率较低，尤其是在环境交互较慢或样本效率较低的情况下。而多进程训练可以让多个智能体同时与环境交互，同一时间内收集更多的样本数据，从而提高训练效率和样本利用率。

A3C 算法

基于多进程训练的思想，A3C（Asynchronous Advantage Actor-Critic）算法应运而生。A3C 算法通过引入多个并行的智能体（进程），每个智能体都拥有独立的策略和价值网络，并与环境进行交互采样数据。每个智能体在与环境交互一段时间后，会将其参数异步地更新到全局网络中，从而实现多进程的协同训练。

如图 2 所示，每一个智能体（进程）都拥有一个独立的网络和环境以供交互，并且每个进程每隔一段时间都会将自己的参数同步到全局网络中，这样就能提高训练效率。

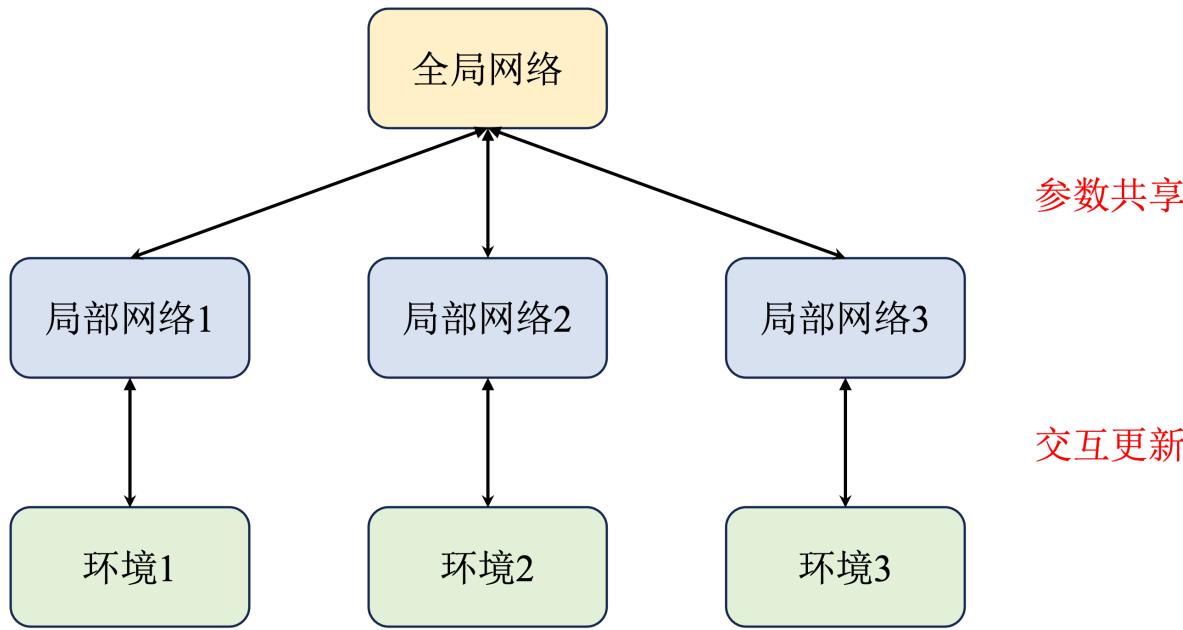


图 2: A3C 算法架构

与传统的单进程训练相比，A3C 算法除了能够提高训练效率外，还具有以下优点：

1. **增强探索能力**：多个智能体在不同的环境实例中进行探索，能够覆盖更广泛的状态空间，减少陷入局部最优的风险。

2. **稳定性提升**: 异步更新机制能够减少不同智能体之间的相互干扰, 从而提高训练的稳定性。

此外, 多进程训练的思路也可以应用到其他强化学习算法中, 包括基于价值的方法例如 DQN 系列算法, 以及其他基于策略梯度的方法, 例如后续章节要讲的 PPO 算法等, 从而进一步提升这些算法的训练效率和性能。

广义优势估计

前面讲到通过引入优化函数来评估策略相对于平均水平的优势, 如式 (10) 所示。

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \quad (10)$$

然而, 在实际中我们并不知道真实的 $Q^\pi(s_t, a_t)$ 和 $V^\pi(s_t)$, 所以需要估计。一种简单的方式是使用一阶时序差分 (TD) 误差来估计优势函数, 如式 (11) 所示。

$$\hat{A}_t^{\text{TD}} = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \quad (11)$$

这种估计方式虽然方差较小, 但是偏差较大。

也可以使用蒙特卡洛估计来估计优势函数, 虽然这种方式是无偏估计, 但是方差较大, 如式 (12) 所示。

$$\hat{A}_t^{\text{MC}} = G_t - V^\pi(s_t) = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V^\pi(s_t) \quad (12)$$

对应实现如代码 3 所示。

代码 3: 计算蒙特卡洛优势估计的示例代码

```
def monte_carlo_advantage_estimation(rewards, values, gamma):
    """计算蒙特卡洛优势估计
    rewards: 奖励序列
    values: 价值函数估计序列
    gamma: 折扣因子
    """
    T = len(rewards)
    advantages = []
    for t in range(T):
        G = 0
        for l in range(T - t):
            G += (gamma ** l) * rewards[t + l]
        advantages.append(G - values[t])
    return advantages

# 示例数据
rewards = [1, 2, 3, 4, 5]
values = [0.5, 1.5, 2.5, 3.5, 4.5]
gamma = 0.9
advantages = monte_carlo_advantage_estimation(rewards, values, gamma)
print("蒙特卡洛优势估计:", advantages)
```

在这里我们还可以使用介于单步 TD 估计和蒙特卡洛估计之间的多步 (n-step) 估计来估计优势函数, 如式 (13) 所示。

$$\hat{A}_t^{\text{TD}(n)} = \sum_{l=0}^{n-1} \gamma^l r_{t+l} + \gamma^n V^\pi(s_{t+n}) - V^\pi(s_t) \quad (13)$$

对应实现如代码 4 所示。

代码 4: 计算 n 步优势估计的示例代码

```
def n_step_advantage_estimation(rewards, values, gamma, n):
    """计算 n 步优势估计
    rewards: 奖励序列
    values: 价值函数估计序列
    gamma: 折扣因子
    n: 步数
    """
    T = len(rewards)
    advantages = []
    for t in range(T):
        G = 0
        for l in range(n):
            if t + l < T: # 确保不越界
                G += (gamma ** l) * rewards[t + l]
            else:
                break
        if t + n < T:
            G += (gamma ** n) * values[t + n]
        advantages.append(G - values[t])
    return advantages

# 示例数据
rewards = [1, 2, 3, 4, 5]
values = [0.5, 1.5, 2.5, 3.5, 4.5]
gamma = 0.9
n = 3
advantages = n_step_advantage_estimation(rewards, values, gamma, n)
print("n 步优势估计:", advantages)
```

然而 n 步估计仍然存在一个问题，即 n 是一个固定的量，在实际环境中，不同状态下的轨迹长度、噪声大小、回报结构等因素可能会有所不同，因此很难让模型在每个时刻动态地选择合适的 n 值来调整方差和偏差的权衡。

为了解决这个问题，我们可以引入一个新的参数 λ ，通过对不同步数的估计进行加权平均，从而形成一种新的估计方式，这就是广义优势估计（generalized advantage estimation, GAE）的方法，如式 (14) 所示。

$$\begin{aligned} A^{\text{GAE}(\gamma, \lambda)}(s_t, a_t) &= \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l} \\ &= \sum_{l=0}^{\infty} (\gamma \lambda)^l (r_{t+l} + \gamma V^\pi(s_{t+l+1}) - V^\pi(s_{t+l})) \end{aligned} \quad (14)$$

其中 δ_{t+l} 表示时间步 $t + l$ 时的 TD 误差，如式 (15) 所示。

$$\delta_{t+l} = r_{t+l} + \gamma V^\pi(s_{t+l+1}) - V^\pi(s_{t+l}) \quad (15)$$

并且广义优势还有一个递推形式，如式 (16) 所示。

$$A_t^{\text{GAE}(\gamma, \lambda)} = \delta_t + \gamma \lambda A_{t+1}^{\text{GAE}(\gamma, \lambda)} \quad (16)$$

注意为了方便，这里将 $A^{\text{GAE}(\gamma, \lambda)}(s_t, a_t)$ 简写为 $A_t^{\text{GAE}(\gamma, \lambda)}$ 。

当 $\lambda = 0$ 时, 根据递推公式, GAE 退化为单步 TD 误差, 如式 (17) 所示。

$$A^{\text{GAE}(\gamma, 0)}(s_t, a_t) = \delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \quad (17)$$

当 $\lambda = 1$ 时, GAE 退化为蒙特卡洛估计, 如式 (18) 所示。

$$A^{\text{GAE}(\gamma, 1)}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l} = \sum_{l=0}^{\infty} (\gamma)^l \delta_{t+l} \quad (18)$$

通过调整 λ 的值, 我们可以在方差和偏差之间进行权衡, 从而获得更稳定和准确的优势估计, 实现如代码 5 所示。

代码 5: 计算广义优势估计的示例代码

```
import torch

def compute_gae(rewards, values, dones, gamma=0.99, lam=0.95):
    """
    计算 Generalized Advantage Estimation (GAE)

    参数:
        rewards (Tensor): shape = [T], 每一步的即时奖励
        values (Tensor): shape = [T+1], 值函数 v(s_t), 最后一个是 v(s_{T+1})
        dones (Tensor): shape = [T], 是否为终止状态 (True/False)
        gamma (float): 折扣因子
        lam (float): GAE 衰减系数 λ

    返回:
        advantages (Tensor): shape = [T], GAE 优势值
        returns (Tensor): shape = [T], 对应的回报值 (adv + value)
    """
    T = len(rewards)
    advantages = torch.zeros(T, dtype=torch.float32)
    last_adv = 0.0

    for t in reversed(range(T)):
        # 如果到达终止状态, 下一步优势为0
        if dones[t]:
            next_non_terminal = 0.0
            next_value = 0.0
        else:
            next_non_terminal = 1.0
            next_value = values[t + 1]

        # TD 误差 δ_t
        delta = rewards[t] + gamma * next_value * next_non_terminal - values[t]
        # GAE 递推公式
        advantages[t] = delta + gamma * lam * next_non_terminal * last_adv
        last_adv = advantages[t]

    returns = advantages + values[:-1]
    return advantages, returns

# 示例数据
rewards = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0])
values = torch.tensor([0.5, 1.5, 2.5, 3.5, 4.5, 0.0]) # 注意 values 长度为 T+1
```

```
dones = torch.tensor([0, 0, 0, 0, 1], dtype=torch.bool)
advantages, returns = compute_gae(rewards, values, dones)
print("GAE 优势估计:", advantages)
print("对应回报值:", returns)
```

思考

相比于 REINFORCE 算法，A2C 主要的改进点在哪里，为什么能提高速度？

改进点主要有：**优势估计**：可以更好地区分好的动作和坏的动作，同时减小优化中的方差，从而提高了梯度的精确性，使得策略更新更有效率；**使用 Critic**：REINFORCE 通常只使用 Actor 网络，没有 Critic 来辅助估计动作的价值，效率更低；**并行化**：即 A3C，允许在不同的环境中并行运行多个 Agent，每个 Agent 收集数据并进行策略更新，这样训练速度也会更快。

A2C 算法是 on-policy 的吗？为什么？

A2C 在原理上是一个 on-policy 算法，首先它使用当前策略的样本数据来更新策略，然后它的优势估计也依赖于当前策略的动作价值估计，并且使用的也是策略梯度方法进行更新，因此是 on-policy 的。但它可以被扩展为支持 off-policy 学习，比如引入经验回放，但注意这可能需要更多的调整，以确保算法的稳定性和性能。

DDPG 算法

本章讲 DDPG 算法和 TD3 算法。DDPG 算法可以看作是 DQN 算法在连续动作空间上的一种扩展，但结构上仍然更像是 Actor-Critic 算法。TD3 算法则是对 DDPG 算法的一种改进，主要是为了克服 DDPG 算法在实际应用中存在的一些缺点，包括过估计偏差、训练不稳定等问题。

DPG 算法

在经典策略梯度算法中，我们学习的是一个随机性策略（stochastic policy），即通过学习一个概率分布 $\pi_\theta(a|s)$ 来选择动作 a 。对于连续动作空间，我们通常会选择高斯分布来表示这个概率分布，但是这种策略往往计算成本较高，且对动作采样的方差较大，导致训练过程不稳定。

在某些情况下，使用确定性策略（deterministic policy）可能会更加高效。确定性策略直接将状态映射到一个具体的动作，而不是一个概率分布，如式 (1) 所示。

$$a = \mu_\theta(s) \quad (1)$$

对应的策略梯度表达式如式 (2) 所示。

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_a Q(s_t, a)|_{a=\mu_\theta(s_t)} \nabla_\theta \mu_\theta(s_t) \right] \quad (2)$$

这就是确定性策略梯度（Deterministic Policy Gradient，DPG）算法的核心，其中 ρ^β 是策略的初始分布。

DDPG 算法

基于 DPG 方法，DDPG 算法在引入神经网络近似的同时，增加了一些要素，如式 (3) 所示。

$$\text{DDPG} = \text{DPG} + \text{target network} + \text{experience replay} + \text{OU noise} \quad (3)$$

其中目标网络和经验回放在 DQN 算法中已经介绍过了，这里我们主要讲讲 OU 噪声，引入噪声的目的是为了增加策略的探索性，从而提高算法的性能和稳定性，如式 (4) 所示。

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \quad (4)$$

其中 x_t 是 OU 过程在时间 t 的值，即当前的噪声值， μ 是回归到的均值， θ 是 OU 过程的回归速率， σ 是 OU 过程的扰动项， dW_t 是布朗运动（Brownian motion）或者维纳过程（Wiener process），是一个随机项，表示随机高斯噪声的微小变化。

代入到 DDPG 算法中，最终的动作选择如式 (5) 所示。

$$a_t = \mu_\theta(s_t) + x_t \quad (5)$$

OU 噪声是一种具有回归特性的随机过程，其与高斯噪声相比的优点在于：

- 探索性：** OU 噪声具有持续的、自相关的特性。相比于独立的高斯噪声，OU 噪声更加平滑，并且在训练过程中更加稳定。这种平滑特性使得 OU 噪声有助于探索更广泛的动作空间，并且更容易找到更好的策略。
- 控制幅度：** OU 噪声可以通过调整其参数来控制噪声的幅度。在 DDPG 算法中，可以通过调整 OU 噪声的方差来控制噪声的大小，从而平衡探索性和利用性。较大的方差会增加探索性，而较小的方差会增加利用性。
- 稳定性：** OU 噪声的回归特性使得噪声在训练过程中具有一定的稳定性。相比于纯粹的随机噪声，在 DDPG 算法中使用 OU 噪声可以更好地保持动作的连续性，避免剧烈的抖动，从而使得训练过程更加平滑和稳定。
- 可控性：** 由于 OU 噪声具有回归特性，它在训练过程中逐渐回归到均值，因此可以控制策略的探索性逐渐减小。这种可控性使得在训练的早期增加探索性，然后逐渐减小探索性，有助于更有效地进行训练。

最后，DDPG 算法的整体流程如图 1 所示，完整的代码实现可参考实战部分的内容。

DDPG 算法

- 1: 初始化 critic 网络 $Q(s, a | \theta^Q)$ 和 actor 网络 $\mu(s | \theta^\mu)$ 的参数 θ^Q 和 θ^μ
 - 2: 初始化对应的目标网络参数，即 $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
 - 3: 初始化经验回放 D
 - 4: **for** 回合数 = 1, M **do**
 - 5: **交互采样：**
 - 6: 选择动作 $a_t = \mu(s_t | \theta^\mu) + \mathcal{N}_t$, \mathcal{N}_t 为探索噪声
 - 7: 环境根据 a_t 反馈奖励 s_t 和下一个状态 s_{t+1}
 - 8: 存储样本 (s_t, a_t, r_t, s_{t+1}) 到经验回放 D 中
 - 9: 更新环境状态 $s_{t+1} \leftarrow s_t$
 - 10: **策略更新：**
 - 11: 从 D 中取出一个随机批量的 (s_i, a_i, r_i, s_{i+1})
 - 12: 求得 $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} | \theta^{\mu'}) | \theta^{Q'})$
 - 13: 更新 critic 参数，其损失为： $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \theta^Q))^2$
 - 14: 更新 actor 参数： $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a | \theta^Q) \Big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s | \theta^\mu) \Big|_{s_i}$
 - 15: 软更新目标网络： $\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$, $\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$
 - 16: **end for**
-

图 1: DDPG 算法流程

注意，跟 DQN 算法中使用的硬更新不同，DDPG 算法中目标网络的更新一般使用软更新（Soft Update）的方式，这样可以使得目标网络的参数更加平滑地变化，从而提高训练的稳定性和收敛性。

TD3 算法

DDPG 算法虽然在连续动作空间中表现不错，但有个很大的缺点，那就是它容易出现 Q 值的过估计问题（overestimation），这会导致策略学习的不稳定，甚至发散。为了解决这个问题，TD3 算法（Twin Delayed DDPG）提出了三种关键的改进技巧，一是双 Q 网络，二是延迟更新，三是目标策略平滑。

双 Q 网络

在 DDPG 算法中，只有一个 Critic 网络来估计动作的价值，这可能会导致 Q 值的过估计问题。为了解决这个问题，TD3 算法引入了双 Q 网络的概念，即使用两个独立的 Critic 网络来估计动作的价值。在计算目标值 y 时，取两个 Q 值中的较小值作为目标值，从而减少过估计的风险，如式 (6) 所示。

$$y = r + \gamma \min_{i=1,2} Q_{\theta'_i(s', \mu_{\phi'}(s'))} \quad (6)$$

对应的损失函数如式 (7) 所示。

$$L(\theta_i) = \mathbb{E}_{(s, a, r, s') \sim D} [(Q_{\theta_i}(s, a) - y)^2] \quad i = 1, 2 \quad (7)$$

延迟更新

延迟更新（Delayed Policy Updates）是指在 TD3 算法中，Actor 网络的更新频率要低于 Critic 网络的更新频率。具体来说，Actor 网络每隔一定数量的时间步才会更新一次，而 Critic 网络则在每个时间步都进行更新。这样做的目的是为了减少策略更新的频率，从而提高训练的稳定性。

举个例子，Critic 就好比领导，Actor 则好比员工，领导不断给员工下达目标，员工不断地去完成目标，如果领导的决策经常失误，那么员工就很容易像无头苍蝇一样不知道该完成哪些目标。

因此，一个好的解决方式就是让领导学得比员工更快，领导学得更快了之后下达目标的失误率就会更低，这样员工就能够更好地完成目标了，从而提高整个团队的效率。在实践中，Actor 的更新频率一般要比 Critic 的更新频率低一个数量级，例如 Critic 每更新 10 次，Actor 只更新 1 次。

目标策略平滑

目标策略平滑（Target Policy Smoothing）是指在计算目标值 y 时，给目标策略添加一些噪声，从而使得目标策略更加平滑，减少过估计的风险，也叫噪声正则（noise regularization）。

在延迟更新中，我们的主要思想是通过提高 Critic 的更新频率来减少值函数的估计误差，也就是“降低领导决策的失误率”。但是这样其实是治标不治本的做法，因为它只是让 Critic 带来的误差不要过分地影响到了 Actor，而没有考虑改进 Critic 本身的稳定性。

因此，我们也可以给 Critic 引入一个噪声提高其抗干扰性，这样一来就可以在一定程度上提高 Critic 的稳定性，从而进一步提高算法的稳定性和收敛性。注意，这里的噪声是在 Critic 网络上引入的，而不是在输出动作上引入的，因此它跟 DDPG 算法中的噪声是不一样的。具体来说，我们可以在计算 TD 误差的时候，给目标值 y 加上一个噪声，并且为了让噪声不至于过大，还增加了一个裁剪（clip），如式 (8) 所示。

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon) \sim \text{clip}(N(0, \sigma), -c, c) \quad (8)$$

其中 $N(0, \sigma)$ 表示均值为 0，方差为 σ 的高斯噪声， ϵ 表示噪声，clip 表示裁剪函数，即将噪声裁剪到 $[-c, c]$ 的范围内， c 是一个超参数，用于控制噪声的大小。

思考

DDPG 算法是 off-policy 算法吗？为什么？

跟 DQN 一样，DDPG 算法，主要结合了经验回放、目标网络和确定性策略，是典型的 off-policy 算法。

软更新相比于硬更新的好处是什么？为什么不是所有的算法都用软更新？

好处：**平滑目标更新**：软更新通过逐渐调整目标网络的参数，使其向主网络的参数靠近，而不是直接复制主网络的参数。这样做可以降低目标的变化幅度，减少了训练中的不稳定性；**降低方差；避免振荡**：软更新可以减少目标网络和主网络之间的振荡，这有助于更稳定地收敛到良好的策略。缺点：**速度**：软更新会使目标网络变得更加缓慢；**探索和稳定性权衡**：一些算法可能更倾向于使用硬更新，因为它们可能需要更频繁地探索新的策略，而不依赖于过去的经验。硬更新允许在每次更新时完全用新策略替代旧策略；**算法需求**：某些算法可能对硬更新更敏感，而且硬更新可能是这些算法的关键组成部分。综上，软更新和硬更新都有其用途，选择哪种方式取决于具体的问题和算法要求。

相比于 DDPG 算法，TD3 算法做了哪些改进？请简要归纳。

双Q网络：TD3 使用了两个独立的 Q 网络，分别用于估计动作的价值。这两个 Q 网络有不同的参数，这有助于减少估计误差，并提高了训练的稳定性；**目标策略噪声**：与 DDPG 不同，TD3 将噪声添加到目标策略，而不是主策略。这有助于减小动作值的过估计误差；**目标策略平滑化**：TD3 使用目标策略平滑化技术，通过对目标策略的参数进行软更新来减小目标策略的变化幅度。这有助于提高稳定性和训练的收敛性。**延迟策略更新**：TD3 引入了延迟策

略更新，意味着每隔一定数量的时间步才更新主策略网络。这可以减小策略更新的频率，有助于减少过度优化的风险，提高稳定性。

TD3 算法中 Critic 的更新频率一般要比 Actor 是更快还是更慢？为什么？

答：Critic 网络的更新频率要比 Actor 网络更快，即延迟策略更新。延迟策略更新的目的是减小策略更新的频率，以避免过度优化和提高训练的稳定性。因为 Critic 网络的更新频率更高，它可以更快地适应环境的变化，提供更准确的动作价值估计，从而帮助 Actor 网络生成更好的策略。

PPO 算法

本章我们开始讲解强化学习中比较重要的 PPO 算法，它在相关应用中有着非常重要的地位，是一个里程碑式的算法。不同于 DDPG 算法，PPO 算法是一类典型的 Actor-Critic 算法，既适用于连续动作空间，也适用于离散动作空间。

PPO 算法是一种基于策略梯度的强化学习算法，由 OpenAI 的研究人员 Schulman 等人在 2017 年提出。PPO 算法的主要思想是通过在策略梯度的优化过程中引入一个重要性权重来限制策略更新的幅度，从而提高算法的稳定性和收敛性。PPO 算法的优点在于简单、易于实现、易于调参，应用十分广泛，正可谓“遇事不决 PPO”。

PPO 的前身是 TRPO 算法，旨在克服 TRPO 算法中的一些计算上的困难和训练上的不稳定性。TRPO 是一种基于策略梯度的算法，它通过定义策略更新的信赖域来保证每次更新的策略不会太远离当前的策略，以避免过大的更新引起性能下降。然而，TRPO 算法需要解决一个复杂的约束优化问题，计算上较为繁琐。本书主要出于实践考虑，这种太复杂且几乎已经被淘汰的 TRPO 算法就不再赘述了，需要深入研究或者工作面试的读者可以自行查阅相关资料。接下来将详细讲解 PPO 算法的原理和实现，希望能够帮助读者更好地理解和掌握这个算法。

重要性采样

在展开 PPO 算法之前，我们先铺垫一个概念，即重要性采样（importance sampling）。重要性采样是一种估计随机变量的期望或者概率分布的统计方法。它的原理也很简单，假设有一个函数 $f(x)$ ，需要从分布 $p(x)$ 中采样来计算其期望值，但是在某些情况下我们可能很难从 $p(x)$ 中采样，这个时候我们可以从另一个比较容易采样的分布 $q(x)$ 中采样，来间接地达到从 $p(x)$ 中采样的效果。这个过程的数学表达式如式 (1) 所示。

$$E_{p(x)}[f(x)] = \int_a^b f(x) \frac{p(x)}{q(x)} q(x) dx = E_{q(x)} \left[f(x) \frac{p(x)}{q(x)} \right] \quad (1)$$

对于离散分布的情况，可以表达为式 (2)。

$$E_{p(x)}[f(x)] = \frac{1}{N} \sum f(x_i) \frac{p(x_i)}{q(x_i)} \quad (2)$$

这样一来原问题就变成了只需要从 $q(x)$ 中采样，然后计算两个分布之间的比例 $\frac{p(x)}{q(x)}$ 即可，这个比例称之为**重要性权重**。换句话说，每次从 $q(x)$ 中采样的时候，都需要乘上对应的重要性权重来修正采样的偏差，即两个分布之间的差异。当然这里可能会有一个问题，就是当 $p(x)$ 不为 0 的时候， $q(x)$ 也不能为 0，但是他们可以同时为 0，这样 $\frac{p(x)}{q(x)}$ 依然有定义，具体的原理由于并不是很重要，因此就不展开讲解了。

通常来讲，我们把这个 $p(x)$ 叫做目标分布， $q(x)$ 叫做提议分布（Proposal Distribution），那么重要性采样对于提议分布有什么要求呢？其实理论上 $q(x)$ 可以是任何比较好采样的分布，比如高斯分布等等，但在实际训练的过程中，聪明的读者也不难想到我们还是希望 $q(x)$ 尽可能 $p(x)$ ，即重要性权重尽可能接近于 1。我们可以从方差的角度来具体展开讲讲为什么需要重要性权重尽可能等于 1，回忆一下方差公式，如式 (3) 所示。

$$Var_{x \sim p}[f(x)] = E_{x \sim p} [f(x)^2] - (E_{x \sim p}[f(x)])^2 \quad (3)$$

结合重要性采样公式，我们可以得到式 (4)。

$$\begin{aligned} Var_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right] &= E_{x \sim q} \left[\left(f(x) \frac{p(x)}{q(x)} \right)^2 \right] - \left(E_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right] \right)^2 \\ &= E_{x \sim p} \left[f(x)^2 \frac{p(x)}{q(x)} \right] - (E_{x \sim p}[f(x)])^2 \end{aligned} \quad (4)$$

不难看出，当 $q(x)$ 越接近 $p(x)$ 的时候，方差就越小，也就是说重要性权重越接近于 1 的时候，反之越大。

其实重要性采样也是蒙特卡洛估计的一部分，只不过它是一种比较特殊的蒙特卡洛估计，允许我们在复杂问题中利用已知的简单分布进行采样，从而避免了直接采样困难分布的问题，同时通过适当的权重调整，可以使得蒙特卡洛估计更接近真实结果。

PPO 算法

既然重要性采样本质上是一种在某些情况下更优的蒙特卡洛估计，再结合前面 Actor-Critic 章节中我们讲到策略梯度算法的高方差主要来源于 Actor 的策略梯度采样估计，读者应该不难猜出 PPO 算法具体是优化在什么地方了。没错，PPO 算法的核心思想就是通过重要性采样来优化原来的策略梯度估计，其目标函数表示如式 (5) 所示。

$$\begin{aligned} J^{\text{TRPO}}(\theta) &= \mathbb{E} \left[r(\theta) \hat{A}_{\theta_{\text{old}}} (s, a) \right] \\ r(\theta) &= \frac{\pi_\theta(a | s)}{\pi_{\theta_{\text{old}}}(a | s)} \end{aligned} \quad (5)$$

这个损失就是置信区间的一部分，一般称作 TRPO 损失。这里旧策略分布 $\pi_{\theta_{\text{old}}}(a | s)$ 就是重要性权重部分的目标分布 $p(x)$ ，目标分布是很难采样的，所以在计算重要性权重的时候这部分通常用上一次与环境交互采样中的概率分布来近似。相应地， $\pi_\theta(a | s)$ 则是提议分布，即通过当前网络输出的 `probs` 形成的类别分布 Categorical 分布（离散动作）或者 Gaussian 分布（连续动作）。

读者们可能对这个写法感到陌生，似乎少了 Actor-Critic 算法中的 `logit_p`，但其实这个公式等价于式 (6)。

$$J^{\text{TRPO}}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \quad (6)$$

换句话说，本质上 PPO 算法就是在 Actor-Critic 算法的基础上增加了重要性采样的约束而已，从而确保每次的策略梯度估计都不会过分偏离当前的策略，也就是减少了策略梯度估计的方差，从而提高算法的稳定性和收敛性。

前面我们提到过，重要性权重最好尽可能地等于 1，而在训练过程中这个权重它是不会自动地约束到 1 附近的，因此我们需要在损失函数中加入一个约束项或者说正则项，保证重要性权重不会偏离 1 太远。具体的约束方法有很多种，比如 KL 散度、JS 散度等等，但通常我们会使用两种约束方法，一种是 clip 约束，另一种是 KL 散度。clip 约束定义如式 (7) 所示。

$$J_{\text{clip}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right] \quad (7)$$

其中 ϵ 是一个较小的超参，一般取 0.1 左右。这个 clip 约束的意思就是始终将重要性权重 $r(\theta)$ 裁剪在 1 的邻域范围内，实现起来非常简单。

另一种 KL 约束定义如式 (8) 所示。

$$J^{KL}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t - \beta \text{KL} [\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_\theta(\cdot | s_t)] \right] \quad (8)$$

KL 约束一般也叫 KL-penalty，它的意思是在 TRPO 损失的基础上，加上一个 KL 散度的惩罚项，这个惩罚项的系数 β 一般取 0.01 左右。这个惩罚项的作用也是保证每次更新的策略分布都不会偏离上一次的策略分布太远，从而保证重要性权重不会偏离 1 太远。在实践中，我们一般用 clip 约束，因为它更简单，计算成本较低，而且效果也更好。

到这里，我们就基本讲完了 PPO 算法的核心内容，其实在熟练掌握 Actor-Critic 算法的基础上，去学习这一类的其他算法是不难的，读者只需要注意每个算法在 Actor-Critic 框架上做了哪些改进，取得了什么效果即可。

一个常见的误区

在之前的章节中，我们讲过 on-policy 和 off-policy 算法，前者使用当前策略生成样本，并基于这些样本来更新该策略，后者则可以使用过去的策略采集样本来更新当前的策略。on-policy 算法的数据利用效率较低，因为每次策略更新后，旧的样本或经验可能就不再适用，通常需要重新采样。而 off-policy 算法由于可以利用历史经验，一般使用经验回放来存储和重复利用之前的经验，数据利用效率则较高，因为同一批数据可以被用于多次更新。但由于经验的再利用，可能会引入一定的偏见，但这也有助于稳定学习。但在需要即时学习和适应的环境中，on-policy 算法可能更为适合，因为它们直接在当前策略下操作。

那么 PPO 算法究竟是 on-policy 还是 off-policy 的呢？有读者可能会因为 PPO 算法在更新时重要性采样的部分中利用了旧的 Actor 采样的样本，就觉得 PPO 算法会是 off-policy 的。实际上虽然这批样本是从旧的策略中采样得到的，但我们并没有直接使用这些样本去更新我们的策略，而是使用重要性采样先将数据分布不同导致的误差进行了修正，即是两者样本分布之间的差异尽可能地缩小。换句话说，就可以理解为重要性采样之后的样本虽然是由旧策略采样得到的，但可以近似为从更新后的策略中得到的，即我们要优化的 Actor 和采样的 Actor 是同一个，因此 PPO 算法是 on-policy 的。

思考

为什么 DQN 和 DDPG 算法不使用重要性采样技巧呢？

DQN 和 DDPG 是 off-policy 算法，它们通常不需要重要性采样来处理不同策略下的采样数据。相反，它们使用目标网络和优势估计等技巧来提高训练的稳定性和性能。

PPO 算法原理上是 on-policy 的，但它可以是 off-policy 的吗，或者说可以用经验回放来提高训练速度吗？为什么？（提示：是可以的，但条件比较严格）

答：跟 A2C 一样，可以将经验回放与 PPO 结合，创建一个 PPO with Experience Replay (PPO-ER) 算法。在 PPO-ER 中，智能体使用经验回放缓冲区中的数据来训练策略网络，这样可以提高训练效率和稳定性。这种方法通常需要调整 PPO 的损失函数和采样策略，以适应 off-policy 训练的要求，需要谨慎调整。

PPO 算法更新过程中在将轨迹样本切分个多个小批量的时候，可以将这些样本顺序打乱吗？为什么？

将轨迹样本切分成多个小批量时，通常是可以将这些样本顺序打乱的，这个过程通常称为样本随机化（sample shuffling），这样做的好处有降低样本相关性、减小过拟合风险以及增加训练多样性（更全面地提高探索空间）。

为什么说重要性采样是一种特殊的蒙特卡洛采样？

原因有：**估计期望值**：蒙特卡洛方法的核心目标之一是估计一个随机变量的期望值。蒙特卡洛采样通过从分布中生成大量的样本，并求取这些样本的平均值来估计期望值。重要性采样也是通过从一个分布中生成样本，但不是均匀地生成样本，而是按照另一个分布的权重生成样本，然后使用这些带权重的样本来估计期望值。**改进采样效率**：重要性采样的主要目的是改进采样效率。当我们有一个难以从中采样的分布时，可以使用重要性采样来重新调整样本的权重，以使估计更准确。这类似于在蒙特卡洛采样中调整样本大小以提高估计的精确性。**权重分布**：在重要性采样中，我们引入了一个额外的权重分布，用于指导采样过程。这个权重分布决定了每个样本的相对贡献，以确保估计是无偏的。在蒙特卡洛采样中，权重通常是均匀分布，而在重要性采样中，权重由分布的比率（要估计的分布和采样分布之间的比例）决定。

Gymnasium 环境介绍

[Gymnasium](#) (曾用名为 OpenAI Gym) 是由 OpenAI 提供的一个标准强化学习环境库，它为研究人员和开发者提供了一套统一的接口，用于创建和比较各种强化学习算法。Gymnasium 支持多种类型的环境，包括经典控制任务、离散和连续动作空间的任务、以及复杂的模拟环境。

如图 1 所示，对于每个环境，Gymnasium 简要介绍了其状态或观测（Observation Space）、动作空间（Action Space）以及奖励机制（Reward Mechanism）。这些信息对于理解环境的动态和设计合适的强化学习算法至关重要。

Cart Pole



This environment is part of the Classic Control environments which contains general information about the environment.

Action Space	<code>Discrete(2)</code>
Observation Space	<code>Box([-4.8 -inf -0.41887903 -inf], [4.8 inf 0.41887903 inf], (4,), float32)</code>
import	<code>gymnasium.make("CartPole-v1")</code>

Description

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson in ["Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem"](#). A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

Action Space

The action is a `ndarray` with shape `(1,)` which can take values `{0, 1}` indicating the direction of the fixed force the cart is pushed with.

- 0: Push cart to the left
- 1: Push cart to the right

图 1 Gymnasium 环境说明

在奖励说明中，通常会提到奖励临界值（Reward Threshold），这是一个预设的奖励水平，表示智能体在该环境中达到“成功”所需的最低奖励。例如，在某些环境中，达到特定的奖励临界值以上意味着智能体很有可能学会了完成任务的基本策略，具体还需可视化验证。

Gymnasium 环境接口

在 Gymnasium 中，每个环境都遵循一个标准的接口，包括以下几个关键方法：

- `reset()`：初始化环境并返回初始观测。
- `step(action)`：执行动作并返回下一个观测、奖励、是否结束标志和额外信息。
- `render()`：可视化当前环境状态。
- `close()`：关闭环境并释放资源。

通过这些方法，用户可以轻松地与环境交互，收集数据，如代码 1 所示。

代码 1 Gymnasium 环境交互示例

```
import gymnasium as gym
# 创建环境
env = gym.make("CartPole-v1")
obs, info = env.reset() # 重置环境，获得初始观测或状态
for _ in range(100):
    # env.render() # 显示画面
    action = env.action_space.sample() # 随机采样一个动作
    obs, reward, done, truncated, info = env.step(action) # 与环境交互
    if done or truncated: # 如果回合结束，重置环境
        obs, info = env.reset()
env.close()
```

其中，`env.step(action)` 返回的 `done` 和 `truncated` 标志用于指示当前回合是否结束。`done` 通常表示智能体达到了终止状态，而 `truncated` 则表示由于时间限制等原因导致的回合结束，通常 `truncated` 会比 `done` 更早触发。

A2C 算法实战

定义模型

通常来讲，Critic 的输入是状态，输出则是一个维度的价值，而 Actor 输入的也是状态，但输出的是概率分布，因此我们可以定义两个网络，如代码 1 所示。

代码 1: 实现 Actor 和 Critic

```
class Critic(nn.Module):
    def __init__(self, state_dim):
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        value = self.fc3(x)
        return value

class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, action_dim)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        logits_p = F.softmax(self.fc3(x), dim=1)
        return logits_p
```

这里由于是离散的动作空间，根据在策略梯度章节中设计的策略函数，我们使用了 softmax 函数来输出概率分布。另外，实践上来看，由于 Actor 和 Critic 的输入是一样的，因此我们可以将两个网络合并成一个网络，以便于加速训练。这有点类似于 Duelling DQN 算法中的做法，如代码 2 所示。

代码 2: 实现合并的 Actor 和 Critic

```

class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim):
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.action_layer = nn.Linear(256, action_dim)
        self.value_layer = nn.Linear(256, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        logits_p = F.softmax(self.action_layer(x), dim=1)
        value = self.value_layer(x)
        return logits_p, value

```

注意当我们使用分开的网络时，我们需要在训练时分别更新两个网络的参数，即需要两个优化，而使用合并的网络时则只需要更新一个网络的参数即可。

动作采样

与 DQN 算法不同等确定性策略不同，A2C 的动作输出不再是 Q 值最大对应的动作，而是从概率分布中采样动作，这意味着即使是很小的概率，也有可能被采样到，这样就能保证探索性，如代码 3 所示。

代码 3: 采样动作

```

from torch.distributions import Categorical
class Agent:
    def __init__(self):
        self.model = ActorCritic(state_dim, action_dim)
    def sample_action(self, state):
        '''动作采样函数
        '''
        state = torch.tensor(state, device=self.device, dtype=torch.float32)
        logits_p, value = self.model(state)
        dist = Categorical(logits_p)
        action = dist.sample()
        return action

```

注意这里直接利用了 PyTorch 中的 `Categorical` 分布函数，这样就能直接从概率分布中采样动作了。

策略更新

我们首先需要计算出优势函数，一般先计算出回报，然后减去网络输出的值即可，如代码 4 所示。

代码清 4: 计算优势函数

```

class Agent:
    def _compute_returns(self, rewards, dones):
        returns = []
        discounted_sum = 0

```

```

for reward, done in zip(reversed(rewards), reversed(dones)):
    if done:
        discounted_sum = 0
        discounted_sum = reward + (self.gamma * discounted_sum)
        returns.insert(0, discounted_sum)

# 归一化
returns = torch.tensor(returns, device=self.device,
dtype=torch.float32).unsqueeze(dim=1)
returns = (returns - returns.mean()) / (returns.std() + 1e-5) # 1e-5 to avoid
division by zero
return returns

def compute_advantage(self):
    '''计算优势函数
    '''

    logits_p, states, rewards, dones = self.memory.sample()
    returns = self._compute_returns(rewards, dones)
    states = torch.tensor(states, device=self.device, dtype=torch.float32)
    logits_p, values = self.model(states)
    advantages = returns - values
    return advantages

```

这里我们使用了一个技巧，即将回报归一化，这样可以让优势函数的值域在 $[-1, 1]$ 之间，这样可以让优势函数更稳定，从而减少方差。计算优势之后就可以分别计算 Actor 和 Critic 的损失函数了，如代码 5 所示。

代码 5: 计算损失函数

```

class Agent:
    def compute_loss(self):
        '''计算损失函数
        '''

        logits_p, states, rewards, dones = self.memory.sample()
        returns = self._compute_returns(rewards, dones)
        states = torch.tensor(states, device=self.device, dtype=torch.float32)
        logits_p, values = self.model(states)
        advantages = returns - values
        dist = Categorical(logits_p)
        log_probs = dist.log_prob(actions)
        # 注意这里策略损失反向传播时不需要优化优势函数，因此需要将其 detach 掉
        actor_loss = -(log_probs * advantages.detach()).mean()
        critic_loss = advantages.pow(2).mean()
        return actor_loss, critic_loss

```

到这里，我们就实现了 A2C 算法的所有核心代码，完整代码请读者参考本书的代码仓库。最后展示一下训练的效果，如图 1 所示。

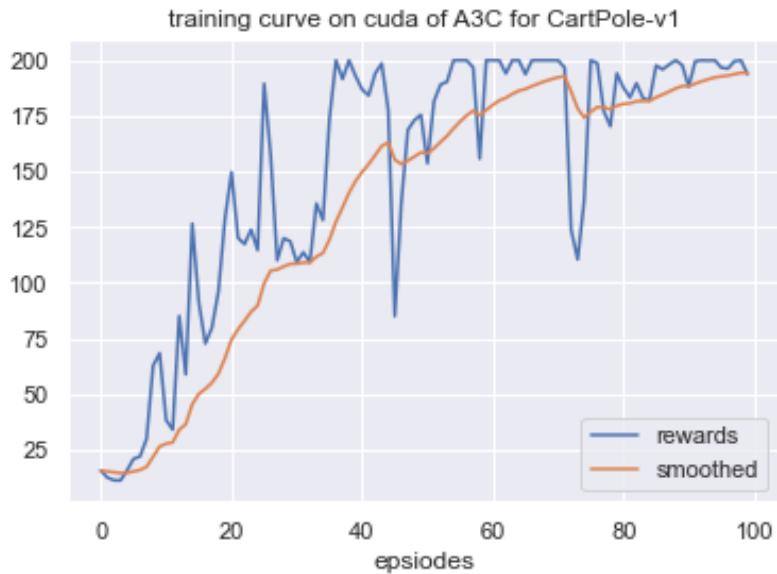


图 1: CartPole 环境 A2C 算法训练曲线

DQN 算法实战

请读者再次注意，本书中所有的实战仅提供核心内容的代码以及说明，完整的代码请读者参考本书对应的 GitHub 仓库。并且正如所有代码实战那样，读者须养成先写出伪代码再编程的习惯，这样更有助于提高对算法的理解。

算法流程

如图 1 所示，DQN 算法的完整流程包括初始化网络和经验回放池、与环境交互采样、存储样本到经验回放池、从经验回放池中随机采样小批量样本、计算目标 Q 值和损失函数、更新网络参数以及定期更新目标网络参数等步骤。

DQN 算法

```
1: 初始化当前网络参数  $\theta$  和目标网络参数  $\bar{\theta} \leftarrow \theta$ 
2: 初始化经验回放  $D$ 
3: for 回合数  $m = 1, 2, \dots, M$  do
4:   重置环境，获得初始状态  $s_0$ 
5:   for 时步  $t = 1, 2, \dots, T$  do
6:     交互采样：
7:       根据  $\epsilon$ -greedy 策略采样动作  $a_t$ 
8:       环境根据  $a_t$  反馈奖励  $r_t$  和下一个状态  $s_{t+1}$ 
9:       存储样本  $(s_t, a_t, r_t, s_{t+1})$  到经验回放  $D$  中
10:      更新状态  $s_{t+1} \leftarrow s_t$ 
11:      策略更新：
12:        从  $D$  中随机采样一个批量的样本
13:        计算  $Q$  的期望值，即  $y_i = r_t + \gamma \max_{a_{i+1}} Q_{\bar{\theta}}(s_{i+1}, a)$ 
14:        计算损失  $L(\theta) = (y_i - Q_{\theta}(s_i, a_i))^2$  并梯度下降更新参数  $\theta$ 
15:        每  $C$  步复制参数到目标网络  $\bar{\theta} \leftarrow \theta$ 
16:    end for
17: end for
```

图 1: DQN 算法流程

定义模型

首先是定义模型，就是定义两个神经网路，即当前网络和目标网络，由于这两个网络结构相同，这里我们只用一个 Python 类来定义，如代码 1 所示。

代码 1: 定义一个全连接网络

```

class MLP(nn.Module): # 所有网络必须继承 nn.Module 类, 这是 PyTorch 的特性
    def __init__(self, input_dim, output_dim, hidden_dim=128):
        super(MLP, self).__init__()
        # 定义网络的层, 这里都是线性层
        self.fc1 = nn.Linear(input_dim, hidden_dim) # 输入层
        self.fc2 = nn.Linear(hidden_dim, hidden_dim) # 隐藏层
        self.fc3 = nn.Linear(hidden_dim, output_dim) # 输出层

    def forward(self, x):
        # 各层对应的激活函数
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x) # 输出层不需要激活函数

```

这里我们定义了一个三层的全连接网络，输入维度就是状态数，输出维度就是动作数，中间的隐藏层采用最常用的ReLU 激活函数。这里我们用 PyTorch 的 Module 类来定义网络，这是 PyTorch 的特性，所有网络都必须继承这个类。在 PyTorch 中，我们只需要定义网络的前向传播，即 forward 函数，反向传播的过程 PyTorch 会自动完成，这也是 PyTorch 的特性。注意，由于我们在本次实战中要解决的问题并不复杂，因此定义的网络模型也比较简单，读者可以根据自己的需求定义更复杂的网络结构，例如增加网络的层数和隐藏层的维度等。

经验回放

经验回放的功能比较简单，主要实现缓存样本和取出样本等两个功能，如代码 2 所示。

代码 2: 定义经验回放

```

class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity # 经验回放的容量
        self.buffer = [] # 用列表存放样本
        self.position = 0 # 样本下标, 便于覆盖旧样本

    def push(self, state, action, reward, next_state, done):
        ''' 缓存样本
        ...
        if len(self.buffer) < self.capacity: # 如果样本数小于容量
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        ''' 取出样本, 即采样
        ...
        batch = random.sample(self.buffer, batch_size) # 随机采出小批量转移
        state, action, reward, next_state, done = zip(*batch) # 解压成状态, 动作等
        return state, action, reward, next_state, done

    def __len__(self):
        ''' 返回当前样本数
        ...
        return len(self.buffer)

```

当然，经验回放的实现方式其实有很多，这里只是一个参考。在 JoyRL 中，我们也提供了一个使用 Python 队列实现的经验回放，读者可以参考相关源码。

定义智能体

智能体即策略的载体，因此有的时候也会称为策略。智能体的主要功能就是根据当前状态输出动作和更新策略，分别跟伪代码中的交互采样和模型更新过程相对应。我们会把所有的模块比如网络模型等都封装到智能体中，这样更符合伪代码的逻辑。而在 JoyRL 线上代码中，会有更泛用的代码架构，感兴趣的读者可以参考相关源码。

如代码 3 所示，两个网络就是前面所定义的全连接网络，输入为状态维度，输出则是动作维度。这里我们还定义了一个优化器，用来更新网络参数。在 DQN 算法中采样动作和预测动作跟 Q-learning 是一样的，其中采样动作使用的是 ϵ - greedy 策略，便于在训练过程中探索，而测试只需要检验模型的性能，因此不需要探索，只需要单纯的进行 argmax 预测即可，即选择最大值对应的动作。

代码 3: 定义智能体

```
class Agent:
    def __init__(self):
        # 定义当前网络
        self.policy_net = MLP(state_dim,action_dim).to(device)
        # 定义目标网络
        self.target_net = MLP(state_dim,action_dim).to(device)
        # 将当前网络参数复制到目标网络中
        self.target_net.load_state_dict(self.policy_net.state_dict())
        # 定义优化器
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=learning_rate)
        # 经验回放
        self.memory = ReplayBuffer(buffer_size)
        self.sample_count = 0 # 记录采样步数
    def sample_action(self,state):
        ''' 采样动作，主要用于训练
        '''
        self.sample_count += 1
        # epsilon 随着采样步数衰减
        self.epsilon = self.epsilon_end + (self.epsilon_start - self.epsilon_end) *
math.exp(-1. * self.sample_count / self.epsilon_decay)
        if random.random() > self.epsilon:
            with torch.no_grad(): # 不使用梯度
                state = torch.tensor(np.array(state), device=self.device,
dtype=torch.float32).unsqueeze(dim=0)
                q_values = self.policy_net(state)
                action = q_values.max(1)[1].item() # choose action corresponding to the
maximum q value
        else:
            action = random.randrange(self.action_dim)
    def predict_action(self,state):
        ''' 预测动作，主要用于测试
        '''
        with torch.no_grad():
            state = torch.tensor(np.array(state), device=self.device,
dtype=torch.float32).unsqueeze(dim=0)
            q_values = self.policy_net(state)
```

```

    action = q_values.max(1)[1].item() # choose action corresponding to the maximum
    q_value
    return action
def update(self):
    pass

```

DQN 算法更新本质上跟 Q-learning 区别不大，但由于读者可能第一次接触深度学习的实现方式，这里单独拎出来分析 DQN 算法的更新方式，如代码 4 所示。

代码 4: DQN 算法更新

```

<div align=center>

</div>
<div align=center>图 1: CartPole 环境 A2C 算法训练曲线</div>

```

首先由于我们是小批量随机梯度下降，所以当经验回放不满足批大小时选择不更新，这实际上是工程性问题。然后在更新时我们取出样本，并转换成 Torch 的张量，便于我们用 GPU 计算。接着计算 Q 值的估计值和实际值，并得到损失函数。在得到损失函数并更新参数时，我们在代码上会有一个固定的写法，即梯度清零，反向传播和更新优化器的过程，跟在深度学习中的写法是一样的，最后我们需要定期更新一下目标网络，即每隔 C 步复制参数到目标网络。

定义环境

由于我们在 Q-learning 算法中已经讲过怎么定义训练和测试过程了，所有强化学习算法的训练过程基本上都是通用的，因此我们在这里及之后的章节中不再赘述。但由于我们在 DQN 算法中使用了跟 Q-learning 算法中不一样的环境，但都是 OpenAI Gym 平台的，所以我们简单介绍一下该环境。环境名称叫做 Cart Pole^①，中文译为推车杆游戏。如图 2 所示，我们的目标是持续左右推动保持倒立的杆一直不倒。

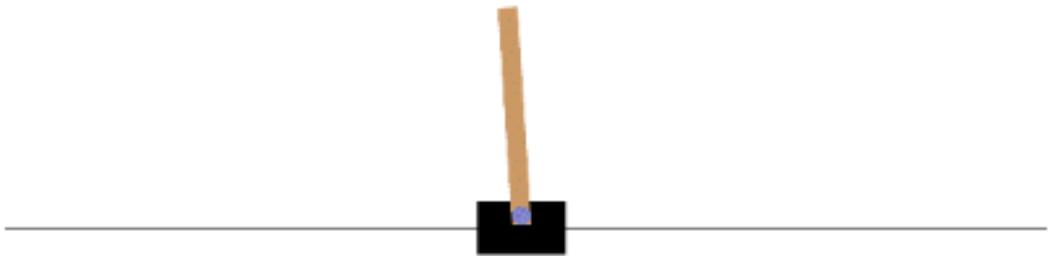


图 2: Cart-Pole 游戏

^① 官网环境介绍：https://gymnasium.farama.org/environments/classic_control/cart_pole/

环境的状态数是 4, 动作数是 2。有读者可能会奇怪，这不是比 Q-learning 算法中的 cliffwalking-v0 环境（状态数是 48, 动作数是 2）更简单吗，应该直接用 Q-learning 算法就能解决？实际上是不能的，因为 Cart Pole 的状态包括推车的位置（范围是 -4.8 到 4.8）、速度（范围是负无穷大到正无穷大）、杆的角度（范围是 -24 度到 24 度）和角速度（范围是负无穷大到正无穷大），这几个状态都是连续的值，也就是前面所说的连续状态空间，因此用 Q-learning 算法是很难解出来的。

环境的奖励设置是每个时步下能维持杆不倒就给一个 +1 的奖励，因此理论上在最优策略下这个环境是没有终止状态的，因为最优策略下可以一直保持杆不倒。回忆前面讲到基于 TD 的算法都必须要求环境有一个终止状态，所以在这里我们可以设置一个环境的最大步数，比如我们认为如果能在两百个时步以内坚持杆不倒就近似说明学到了一个不错的策略。

设置参数

定义好智能体和环境之后就可以开始设置参数了，如代码 5 所示。

代码 5: 参数设置

```
self.epsilon_start = 0.95 # epsilon 起始值  
self.epsilon_end = 0.01 # epsilon 终止值  
self.epsilon_decay = 500 # epsilon 衰减率  
self.gamma = 0.95 # 折扣因子  
self.lr = 0.0001 # 学习率  
self.buffer_size = 100000 # 经验回放容量  
self.batch_size = 64 # 批大小  
self.target_update = 4 # 目标网络更新频率
```

与 Q-learning 算法相比，除了 ϵ 、折扣因子以及学习率之外多了三个超参数，即经验回放的容量、批大小和目标网络更新频率。注意这里学习率在更复杂的环境中一般会设置得比较小，经验回放的容量是一个比较经验性的参数，根据实际情况适当调大即可，不需要额外花太多时间调。批大小也比较固定，一般都在 64, 128, 256, 512 中间取值，目标网络更新频率会影响智能体学得快慢，但一般不会导致学不出来。总之，DQN 算法相对来说是深度强化学习的一个稳定且基础的算法，只要适当调整学习率都能让智能体学出一定的策略。

最后展示一下我们的训练曲线和测试曲线，分别如图 3 和 4 所示。

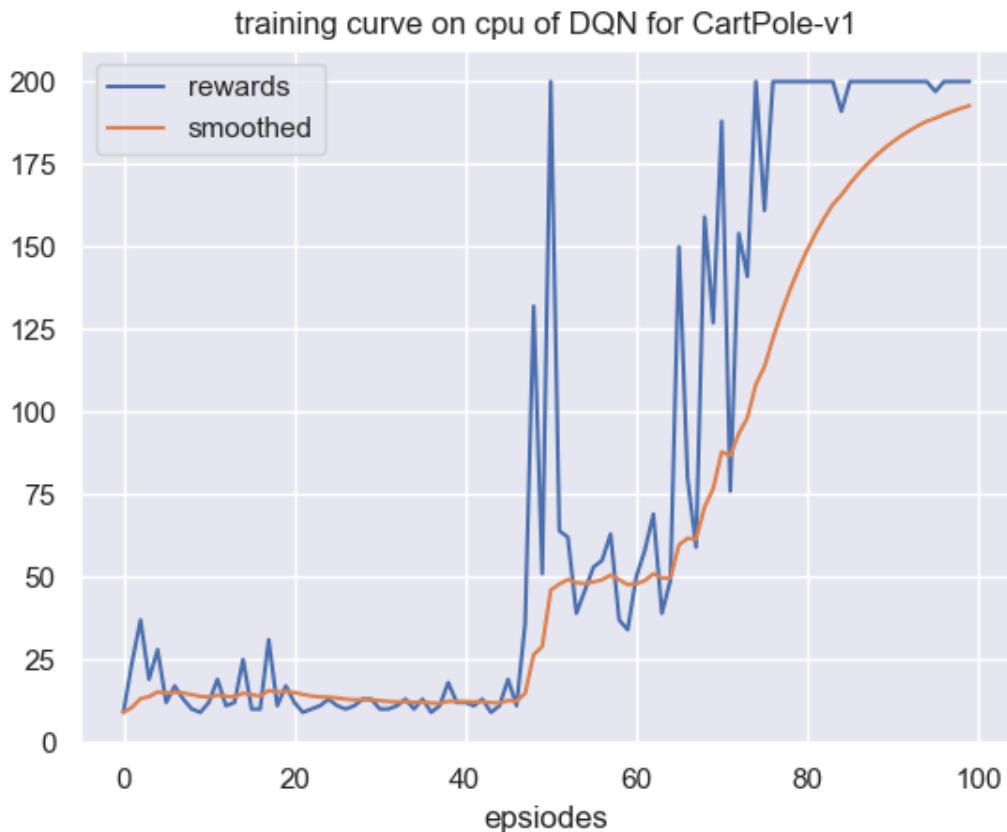


图 3: CartPole-v1 环境 DQN 算法训练曲线

testing curve on cpu of DQN for CartPole-v1

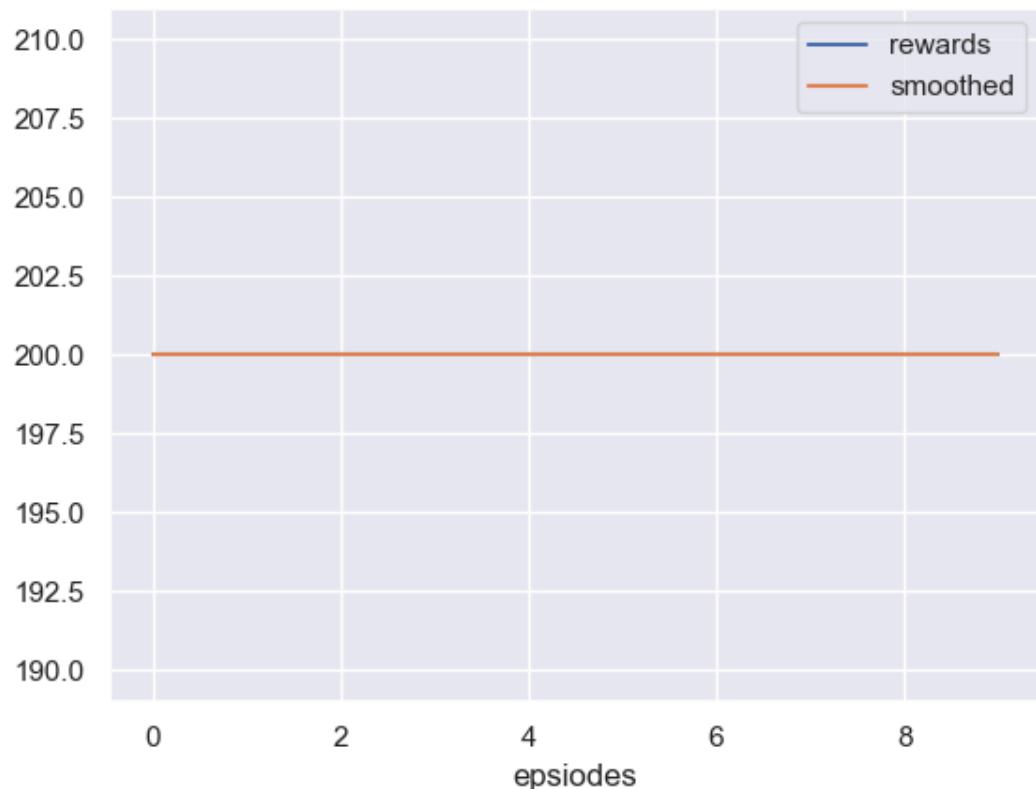


图 4: CartPole-v1 环境 DQN 算法测试曲线

其中我们该环境每回合的最大步数是 200，对应的最大奖励也为 200，从图中可以看出，智能体确实学到了一个最优的策略，即达到收敛。

DDPG 算法实战

同之前章节一样，本书在实战中将演示一些核心的代码，完整的代码请参考 JoyRL 代码仓库。

算法流程

如图 1 所示，DDPG 算法的训练方式其实更像 DQN 算法。注意在第 15 步中 DDPG 算法将当前网络参数复制到目标网络的方式是软更新，即每次一点点地将参数复制到目标网络中，与之对应的是 DQN 算法中的硬更新。软更新的好处是更加平滑缓慢，可以避免因权重更新过于迅速而导致的震荡，同时降低训练发散的风险。

DDPG 算法

- 1: 初始化 critic 网络 $Q(s, a | \theta^Q)$ 和 actor 网络 $\mu(s | \theta^\mu)$ 的参数 θ^Q 和 θ^μ
- 2: 初始化对应的目标网络参数，即 $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
- 3: 初始化经验回放 D
- 4: **for** 回合数 $= 1, M$ **do**
- 5: **交互采样：**
- 6: 选择动作 $a_t = \mu(s_t | \theta^\mu) + \mathcal{N}_t$, \mathcal{N}_t 为探索噪声
- 7: 环境根据 a_t 反馈奖励 s_t 和下一个状态 s_{t+1}
- 8: 存储样本 (s_t, a_t, r_t, s_{t+1}) 到经验回放 D 中
- 9: 更新环境状态 $s_{t+1} \leftarrow s_t$
- 10: **策略更新：**
- 11: 从 D 中取出一个随机批量的 (s_i, a_i, r_i, s_{i+1})
- 12: 求得 $y_i = r_i + \gamma Q' \left(s_{i+1}, \mu' \left(s_{i+1} | \theta^{\mu'} \right) | \theta^{Q'} \right)$
- 13: 更新 critic 参数，其损失为： $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \theta^Q))^2$
- 14: 更新 actor 参数： $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a | \theta^Q) \Big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s | \theta^\mu) \Big|_{s_i}$
- 15: 软更新目标网络： $\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$, $\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$
- 16: **end for**

图 1: DDPG 算法流程

定义模型

如代码 1 所示，DDPG 算法的模型结构跟 Actor-Critic 算法几乎是一样的，只是由于 DDPG 算法的 Critic 是 Q 函数，因此也需要将动作作为输入。除了模型之外，目标网络和经验回放的定义方式跟 DQN 算法一样，这里不做展开。

代码 1: 实现 DDPG 算法的 \$text{Actor}\$ 和 \$text{Critic}\$

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim = 256, init_w=3e-3):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(state_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, action_dim)
```

```

        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = torch.tanh(self.linear3(x)) # 输入0到1之间的值
        return x

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=256, init_w=3e-3):
        super(Critic, self).__init__()

        self.linear1 = nn.Linear(state_dim + action_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, 1)
        # 随机初始化为较小的值
        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

    def forward(self, state, action):
        # 按维数1拼接
        x = torch.cat([state, action], 1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)
        return x

```

动作采样

由于 DDPG 算法输出的是确定性策略，因此不需要像其他策略梯度算法那样，通过借助高斯分布来采样动作的概率分布，直接输出 Actor 的值即可，如代码 2 所示。

代码 2: DDPG 算法的动作采样

```

class Agent:
    def __init__(self):
        pass
    def sample_action(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(self.device)
        action = self.actor(state)
        return action.detach().cpu().numpy()[0, 0]

```

策略更新

如代码 3 所示，DDPG 算法的策略更新则更像 Actor-Critic 算法。

代码 3: DDPG 算法的策略更新

```

class Agent:
    def __init__(self):

```

```
pass
def update(self):
    # 从经验回放中中随机采样一个批量的样本
    state, action, reward, next_state, done = self.memory.sample(self.batch_size)
    actor_loss = self.critic(state, self.actor(state))
    actor_loss = - actor_loss.mean()

    next_action = self.target_actor(next_state)
    target_value = self.target_critic(next_state, next_action.detach())
    expected_value = reward + (1.0 - done) * self.gamma * target_value
    expected_value = torch.clamp(expected_value, -np.inf, np.inf)

    actual_value = self.critic(state, action)
    critic_loss = nn.MSELoss()(actual_value, expected_value.detach())

    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    # 各自目标网络的参数软更新
    for target_param, param in zip(self.target_critic.parameters(),
self.critic.parameters()):
        target_param.data.copy_(
            target_param.data * (1.0 - self.tau) +
            param.data * self.tau
        )
    for target_param, param in zip(self.target_actor.parameters(),
self.actor.parameters()):
        target_param.data.copy_(
            target_param.data * (1.0 - self.tau) +
            param.data * self.tau
        )
```

核心代码到这里全部实现了，我们展示一下训练效果，如图 2 所示。

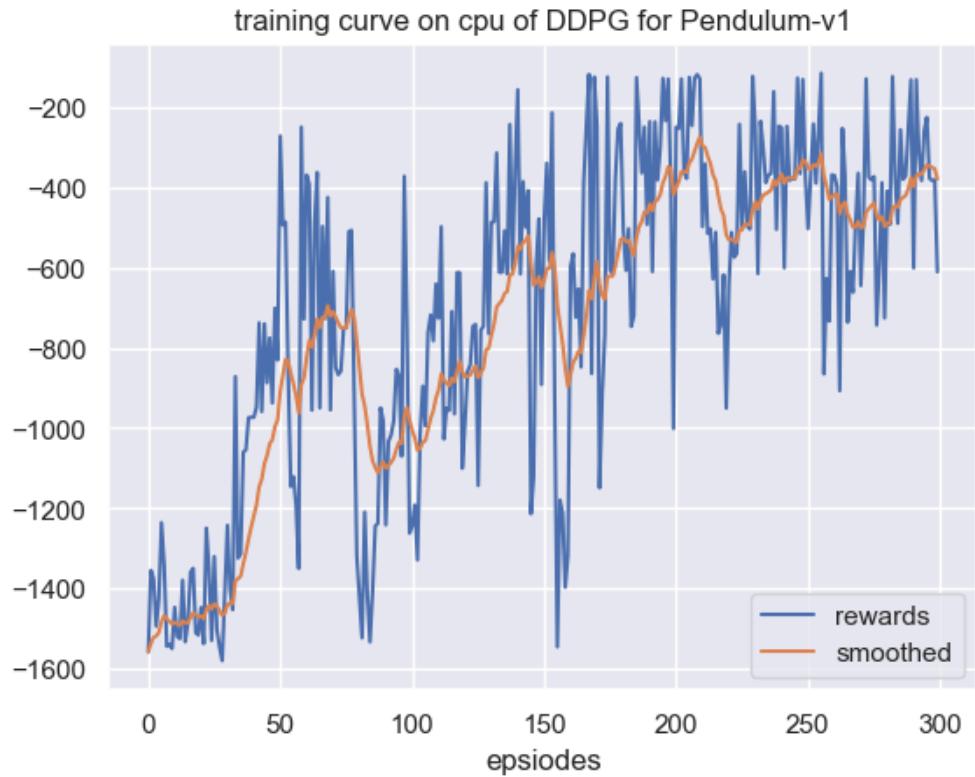


图 2: Pendulum 环境 DDPG 算法训练曲线

这里我们使用了一个具有连续动作空间的环境 Pendulum，如图 3 所示。在该环境中，钟摆以随机位置开始，我们的目标是将其向上摆动，使其保持直立。

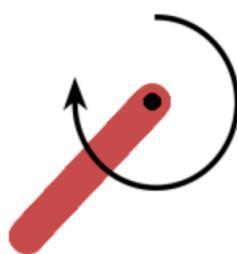


图 3: Pendulum 环境演示