第3章

强化学习

CHAPTER 3

3.1　概述

强化学习一词最早于 20 世纪 60 年代开始出现在工程文献中，其中，明斯基(Marvin Minsky)在其 1961 年发表的论文 *Steps Toward Artificial Intelligence* 中提出的强化学习一词影响广泛。目前认为强化学习的出现与心理学中的动物学习和最优控制的优化理论这两个领域的发展密切相关。

心理学中的动物学习是早期人工智能算法研究的方向之一，即通过努力模仿人类大脑和动物学习的过程来建立模型和算法，其中极为重要的一种方式就是试错学习。鉴于此，随后的研究逐步深入到有监督的学习算法，尽管其中也有奖励和惩罚的概念，但实际上却是一种模式识别和感知学习的系统。此类方法最具代表性的就是 Donald Michie 提出的解决 Tic-Tac-Toe 游戏的系统。Tic-Tac-Toe 就是井字棋游戏，在 3×3 的面板上，一个人画 O，一个人画 X，当某种标记沿着一条直线连起来则该标记者获胜，后续其他学者也陆续地提出了其他形式的试错系统来解决此问题。另一方式是时间差分学习(Temporal-Difference Learning)，它的部分概念来源于动物心理学，借鉴了食物或者疼痛给动物带来的刺激。该部分文献虽然较少，但是为后续的经典强化学习模型 Q-Learning 提供了坚实的基础。

最优控制是另一种与强化学习密切相关的研究领域。最优控制是指在给定的约束条件下，寻求一个控制，使给定的系统性能指标达到极大值(或极小值)。例如，确定一个最优控制方式使空间飞行器由一个轨道转换到另一个轨道过程中燃料消耗最少。20 世纪 50 年代中期，理查德·贝尔曼(Richard Bellman)等通过扩展哈密尔顿和 Jacobi 的理论，提出了贝尔曼方程来解决最优控制问题，也被称为动态规划方程。在强化学习中，贝尔曼方程用于计算给定策略下价值函数在策略指引下所采取的轨迹对应的期望，描述了不同状态值之间的关系。1957 年，贝尔曼还提出了最优控制问题的随机离散版本，也就是著名的马尔可夫决策过程(Markov Decision Process，MDP)。1960 年，Howard 提出马尔可夫决策过程的策略迭代方法，这些都成为现代强化学习的理论基础。1972 年，Klopf 把试错学习和时序差分结合在一起。1988 年，Sutton 提出了时序差分(Temporal Difference，TD)算法，其证明时序差分学习可以和有监督学习获得同样的结果而且占用更少的内存，并且具有更快的收敛速度。1989 年，Watkins 提出了 Q-Learning 的模型，它是时间差分与最优控制的结合，也将之前不同强化学习发展的路径统一到了一起。但是这个模型对于状态过多等问题仍然无法有效解决。2013 年，DeepMind 进一步深入研究，提出了基于深度学习的强化学习模型，即 Deep Q-Network(DQN)，并在 2015 年发表了改进版本。随着强化学习的研究和应用日益开展，成为目前机器学习领域的研究热点之一，并逐步应用于人工智能、调度优化、最优控制等领域。

强化学习是一个大家族，包含很多种算法。例如，通过行为的价值选取特定行为的方法(包括使用表格学习的 Q-Learning、Sarsa)、使用神经网络学习的 DQN 和直接输出行为的 Policy gradients 等。本章主要讲解 Q-Learning 及其改进算法 DQN。

3.2 理论基础

3.2.1 基本框架

强化学习又称再励学习、评价学习或增强学习，是机器学习的范式和方法论之一，用于描述和解决智能体(Agent)在与环境的交互过程中通过学习策略以达成回报最大化或实现特定目标的问题。

这里以多臂游戏机问题作为引入案例：假设你走进了一个电玩城，面前有一排抽奖游戏机。每个游戏机都有多个拉杆，每次拉动一个拉杆都有可能获得一定的奖励。然而，你并不知道每个游戏机背后的概率分布，也就是拉动某个拉杆可能获得奖励的概率。你的目标是在有限的尝试次数内，找到一个最佳的策略，以最大化你的累积奖励。依照此场景下面进行强化学习框架的讲解。

1. 智能体

智能体就是你的化身，试图在电玩城中找到一个最佳策略，即选择哪个游戏机的哪个拉杆来最大化你的奖励。在强化学习中，智能体是在环境中进行决策的实体，它通过与环境的交互来学习如何做出决策，以最大化某种回报或奖励。

(1) 智能体能够感知环境的状态，并基于这些状态信息选择并执行动作。

(2) 在执行动作后，智能体会接收到来自环境的反馈，包括新的状态信息和奖励，也可能是惩罚。这些反馈帮助智能体评估其动作的效果，并指导其后续的学习和行为。

(3) 策略(Policy)定义了智能体在不同状态下选择动作的概率分布。智能体的目标是找到一个最优策略，以最大化长期累积奖励。

2. 环境

电玩城和游戏机构成了环境，提供了你与之交互的场景。每次你选择拉动某个拉杆，环境会给你一个奖励，即正反馈或负反馈。强化学习中的环境是智能体进行交互的外部系统或世界。环境负责接收智能体执行的动作，然后产生新的状态和奖励/惩罚作为反馈。

(1) 系统中智能主体以外的部分。

(2) 向智能主体反馈状态和奖励。

(3) 按照一定的规律发生变化。

3. 动作

在每个时间步，你可以选择拉动一个游戏机的某个拉杆，这就是你的动作。强化学习中的动作是智能体根据当前环境状态所采取的行为或决策。这些动作可以是物理上的(如机器人移动手臂、汽车转向等)，也可以是逻辑上的(如选择某个选项、执行某个命令等)。智能体执行的动作会影响环境的状态，并导致环境产生新的状态和奖励。这些新的状态和奖励会作为反馈提供给智能体，使其能够学习如何改进其策略。

4. 状态

状态是强化学习中的一个核心概念，它定义了智能体在特定时间点所处的位置或情境。在这个例子中，状态可以简单地表示为你当前所面对的游戏机以及拉杆。在强化学习中，状态是对环境当前情况的描述，它包含智能体进行决策所需的所有信息。

在强化学习迭代过程中，状态转移是指环境从一个状态变为另一个状态的过程。当智能体执行一个动作时，环境会根据其内部规则（状态转移函数）更新到新的状态。新的状态会作为反馈提供给智能体，使其能够学习如何改进其策略。智能体需要根据当前状态来选择动作，准确描述和表示状态对于智能体的学习性能非常重要。智能体通过不断与环境交互并观察状态的变化来学习如何做出最优决策。

5. 奖励

每次拉动拉杆后，你会得到一个奖励，表示你本次操作的好坏程度。在强化学习中，奖励是环境在智能体执行某个动作后给出的反馈信号，用于评估该动作在达到目标过程中的价值或效果。奖励是强化学习的核心驱动力，它指导着智能体如何学习并改进其行为策略。

在强化学习中，你需要学习一个策略，即在每个状态下选择哪个动作，以最大化累积奖励。同时，你还需要估计每个状态的价值，表示在当前策略下从这个状态开始能够获得的预期累积奖励。在游戏机的例子中，每个拉杆的概率分布即为状态的价值。图 3-1 展示了强化学习的框架。

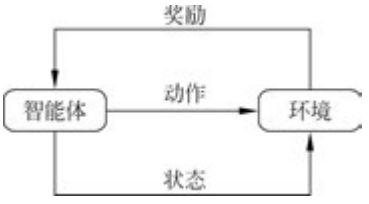

图 3-1　强化学习框架

强化学习中一个重要的权衡是探索（Exploration）与利用（Exploitation）。在游戏机问题中，探索意味着尝试不同的拉杆，以便更好地了解每个游戏机的概率分布。利用则是根据已有的信息，选择目前看起来最好的拉杆来获得更多奖励。

强化学习是一个试错过程。开始时，你可能会随机尝试不同的拉杆，逐渐积累经验并估计每个拉杆的价值。然后，你可以根据估计的价值来改进你的策略，选择估计价值最高的拉杆。这个过程就是策略改进。接着，你可以再次尝试新的拉杆，重新估计价值，进而再次改进策略，形成策略迭代。

尽管游戏机问题相对简单，但在实际中还存在许多挑战。例如，如何在探索和利用之间平衡，以便获得最优解；如何有效地估计状态的价值；以及如何应对连续状态和动作空间等问题。强化学习的目标是在面对这些挑战时，找到一种能够逐步优化策略的方法。抽奖游戏机问题提供了一个直观的例子，有助于理解强化学习的基本概念，如智能体、环境、动作、状态、奖励等。通过试错、探索和利用，强化学习可以帮助智能体逐渐找到最优策略，以最大化累积奖励。然而，强化学习仍然面临各种挑战，需要不断探索和创新，以应对现实世界中更加复杂的问题。

3.2.2　马尔可夫概念

1. 马尔可夫性质

在强化学习中，通常使用马尔可夫决策过程（Markov Decision Process，MDP）来描述具

有马尔可夫性的问题。若状态 S_t 具有马尔可夫性，当且仅当$\mathbb{P}[S_{t+1}|S_t]=\mathbb{P}[S_{t+1}|S_1,S_2,\cdots,S_t]$。当给定当前时刻的状态时，将来与历史无关，即状态是对过去的充分统计，这个时候这个问题便具有马尔可夫性，可以用马尔可夫过程来描述。MDP 是强化学习问题在数学上的理想化形式。几乎所有的强化学习问题都可以在数学上表示为马尔可夫决策过程。

2. 状态转移矩阵

对于马尔可夫状态 s 与其后继状态 s'，它们的状态转移概率定义为

$$\boldsymbol{P}_{ss'}=\mathbb{P}[S_{t+1}=s' \mid S_t=s] \tag{3-1}$$

状态转移矩阵 $\boldsymbol{P}$ 定义了马尔可夫状态 s 到其所有后继状态 s' 的转移概率：

$$\boldsymbol{P}=\begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \tag{3-2}$$

其中，P_{ab} 代表从状态 s_a 转移到后继状态 s_b，矩阵的每一行总和为 1。

3. 马尔可夫过程

马尔可夫过程是一种无记忆的随机过程。马尔可夫过程由元组$(S,\boldsymbol{P})$构成，其中，S 是有限状态的集合，而 $\boldsymbol{P}$ 是状态转移矩阵 $\boldsymbol{P}_{ss'}=\mathbb{P}[S_{t+1}=s'|S_t=s]$。马尔可夫过程基本可以分为以下三类。

(1) 时间、状态都离散的马尔可夫过程(马尔可夫链)。

(2) 时间连续、状态离散的马尔可夫过程(连续时间的马尔可夫链)。

(3) 时间、状态都连续的马尔可夫过程。

从初始状态 S_1 =“课程一”开始，我们可以从马尔可夫链中采样一些子序列，每个子序列又称 episode，即 $S_1,S_2,\cdots,S_T$。

接下来举个例子，把上课学习的过程当作一个简单的马尔可夫过程，包含马尔可夫链的两个元素 S、$\boldsymbol{P}$。图 3-2 展示了马尔可夫决策链。

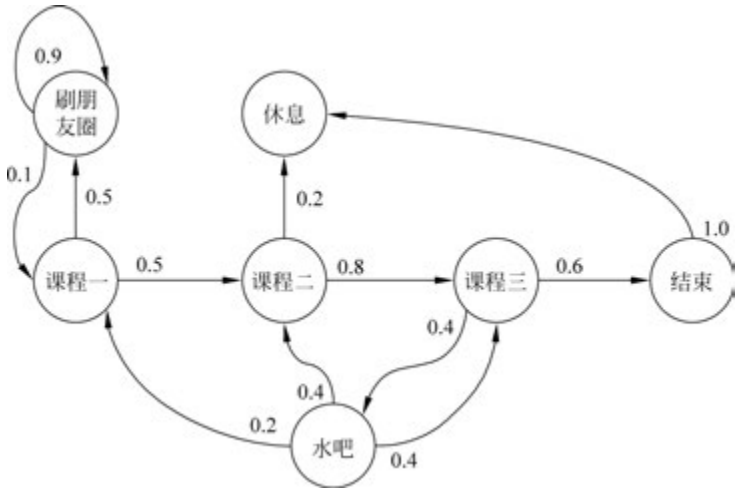

图 3-2 马尔可夫决策链

这个任务中有 7 个状态(刷朋友圈,课程一,课程二,课程三,结束,水吧,休息)。在每个状态对应着相应的转移概率,例如,在“课程一”状态下,有 0.5 的概率继续转移到“课程二”以及 0.5 的概率转移到“刷朋友圈”,这里可以注意到每个状态转移概率之和为 1。在这个例子中,状态转移矩阵见图 3-3。

$$\boldsymbol{P}=\begin{array}{c}\\ \text{课程一}\\ \text{课程二}\\ \text{课程三}\\ \text{结束}\\ \text{水吧}\\ \text{刷朋友圈}\\ \text{休息}\end{array}\begin{array}{c}\begin{array}{ccccccc}\text{课程一} & \text{课程二} & \text{课程三} & \text{结束} & \text{水吧} & \text{刷朋友圈} & \text{休息}\end{array}\\ \begin{bmatrix}0 & 0.5 & 0 & 0 & 0 & 0.5 & 0\\ 0 & 0 & 0.8 & 0 & 0 & 0 & 0.2\\ 0 & 0 & 0 & 0.6 & 0.4 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 1.0\\ 0.2 & 0.4 & 0.4 & 0 & 0 & 0 & 0\\ 0.1 & 0 & 0 & 0 & 0 & 0.9 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}\end{array}$$

图 3-3　状态转移矩阵

3.2.3　马尔可夫奖励

1. 马尔可夫奖励过程

马尔可夫奖励过程(Markov Reward Process,MRP)是具有价值的马尔可夫链,由元组$(S,\boldsymbol{P},R,\gamma)$构成。其中,$S$ 是有限状态的集合,$\boldsymbol{P}$ 是状态转移矩阵 $\boldsymbol{P}_{ss'}=\mathbb{P}[S_{t+1}=s'|S_t=s]$;$R_s$ 是奖励函数,$R_s=\mathbb{E}[R_{t+1}|S_t=s]$,用奖励的期望去除奖励的随机性;$\gamma$ 是折扣因子,$\gamma\in[0,1]$。马尔可夫奖励过程在马尔可夫过程的基础上添加了奖励函数和折扣因子。

下面将之前例子中的马尔可夫过程相应地变成马尔可夫奖励过程,如图 3-4 所示。

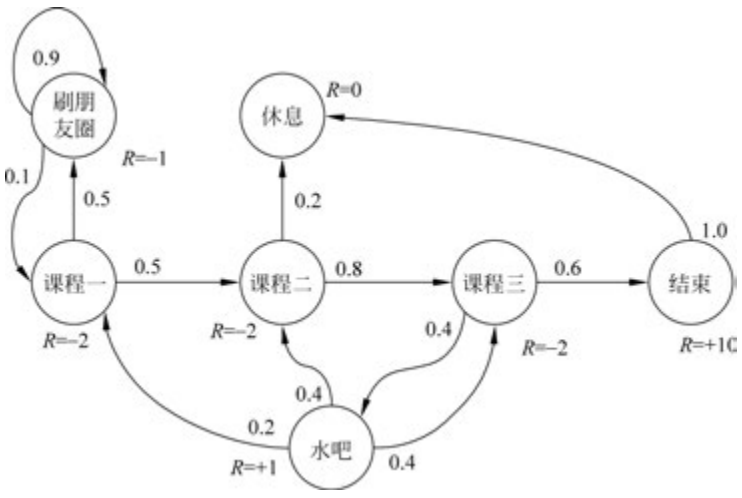

图 3-4　马尔可夫奖励过程

2. 回报

在一个马尔可夫奖励过程中,从 t 时刻的状态 S_t 开始,直到终止状态时,所有奖励的衰减之和 G_t 称为回报。

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3-3}$$

折扣因子 γ 在其中起到了很重要的作用：①避免有环的马尔可夫过程计算收益时出现无限循环；②折扣因子让即时奖励比延时奖励占的权重更大；③动物/人类行为中都表现出对即时奖励的偏好，折扣因子赋予了智能体这一特性；④可以表达对未来的不确定性。

3. 价值函数

几乎所有的强化学习算法都涉及价值函数的计算。价值函数是状态(或状态与动作二元组)的函数，用来评估当前智能体在给定状态(或给定状态与动作)下有多好。这里“有多好”的概念是用未来预期的收益来定义的，或者更准确地说，就是回报的期望值。当然，智能体期望未来能得到的收益取决于智能体所选择的动作。因此，价值函数是与特定的行为方式相关的，也就是策略。

在马尔可夫奖励过程中，一个状态的期望回报被称为这个状态的价值函数。价值函数 $v(s)$ 给出状态 s 的长期价值，价值函数输入为某个状态，输出这个状态的价值。

$$v(s) = \mathbb{E}[G_t \mid S_t = s] \tag{3-4}$$

4. 贝尔曼方程

求解价值函数就需要利用贝尔曼方程，贝尔曼方程是强化学习的基础和核心。

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t] \tag{3-5}$$

价值函数分解为 R_{t+1} 和 $\gamma v(S_{t+1})$ 两部分。

推导过程如下。

$$v(s) = \mathbb{E}[G_t \mid S_t = s] \tag{3-6}$$

$$= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s] \tag{3-7}$$

$$= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \mid S_t = s] \tag{3-8}$$

$$= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \tag{3-9}$$

$$= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \tag{3-10}$$

贝尔曼方程还可以用矩阵形式简明表示为

$$\boldsymbol{v} = \boldsymbol{R} + \gamma \boldsymbol{P}\boldsymbol{v} \tag{3-11}$$

其中，$\boldsymbol{v}$ 是一个列向量，每个状态一个条目：

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix} + \gamma \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} \tag{3-12}$$

由于贝尔曼方程是线性方程，因此可以进行直接求解，即

$$\boldsymbol{v} = (\boldsymbol{I} - \gamma \boldsymbol{P})^{-1} \boldsymbol{R} \tag{3-13}$$

3.2.4 决策过程

1. 马尔可夫决策过程

马尔可夫决策过程(MDP)是具有决策的马尔可夫奖励过程(MRP)，其所有状态都满足

马尔可夫性质。马尔可夫决策过程由元组$(S,A,\boldsymbol{P},R,\gamma)$构成。其中,$S$ 是有限状态的集合;A 是有限动作的集合;$\boldsymbol{P}$ 是状态转移矩阵$\boldsymbol{P}_{ss'}=\mathbb{P}[S_{t+1}=s'|S_t=s,A_t=a]$;$R$ 是奖励函数,$R_s^a=\mathbb{E}[R_{t+1}|S_t=s,A_t=a]$;$\gamma$ 是折扣因子,$\gamma\in[0,1]$。

2. 策略

策略 π 是一个函数,表示输入状态为 s 的情况下采取动作 a 的概率,策略完全定义了智能体的行为,在马尔可夫决策的过程中,策略仅取决于当前状态而非历史记录。表达式为$\pi(a|s)=\mathbb{P}[A_t=a|S_t=s]$。

严格地说,策略是从状态到每个动作的选择概率之间的映射。如果智能体在时刻 t 选择了策略 π,那么 $\pi(a|s)$就是当 $S_t=s$ 时 $A=a$ 的概率。$\pi(a|s)$中间的"|"表示每个 $s\in S$ 都定义了一个在 $a\in A$ 上的概率分布。强化学习方法规定了智能体的策略如何随着其经验而发生变化。

3. MDP 的价值函数

在马尔可夫决策过程中,一个状态价值函数 $v_\pi(s)$是从状态 s 出发,遵循策略 π 得到的期望回报:

$$v_\pi(s)=\mathbb{E}_\pi[G_t\mid S_t=s] \tag{3-14}$$

而一个动作价值函数 $q_\pi(s,a)$是从状态 s 开始,遵循策略 π,对当前状态 s 执行动作 a 得到的期望回报:

$$q_\pi(s,a)=\mathbb{E}_\pi[G_t\mid S_t=s,A_t=a] \tag{3-15}$$

MDP 是马尔可夫链的一种扩展,提供了一个用于对决策情景建模的数学框架,几乎所有的强化学习问题都可以建模为 MDP。

4. 贝尔曼方程

在强化学习中,贝尔曼方程扮演着核心角色,用于计算给定策略下价值函数的期望,通过状态值函数的定义和推导,展示了如何通过即时奖励和未来奖励的期望来计算当前状态的价值。此外,贝尔曼方程的求解方法包括解析法和迭代法,其中,迭代法通过不断更新状态值函数直到收敛,来逼近最优策略下的价值函数。贝尔曼方程不仅在理论上具有重要意义,也是实际应用中解决优化问题的重要工具。

事实上,前文所介绍的状态函数可以分解为即时奖励与后继状态的折扣值。

$$v_\pi(s)=\mathbb{E}_\pi[R_{t+1}+\gamma v_\pi(S_{t+1})\mid S_t=s] \tag{3-16}$$

动作函数也可以进行类似分解:

$$q_\pi(s,a)=\mathbb{E}_\pi[R_{t+1}+\gamma q_\pi(S_{t+1},A_{t+1})\mid S_t=s,A_t=a] \tag{3-17}$$

状态价值函数与动作价值函数之间具有密切的关系,即

$$v_\pi(s)=\sum_{a\in A}\pi(a\mid s)q_\pi(s,a) \tag{3-18}$$

$$q_\pi(s,a)=R_s^a+\gamma\sum_{s'\in S}P_{ss'}^a v_\pi(s') \tag{3-19}$$

经过递推可以得到贝尔曼方程:

$$v_\pi(s)=\sum_{a\in A}\pi(a\mid s)(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^a v_\pi(s')) \tag{3-20}$$

$$q_\pi(s,a)=R_s^a+\gamma\sum_{s'\in S}P_{ss'}^a\sum_{a'\in A}\pi(a'\mid s')q_\pi(s',a') \tag{3-21}$$

5. 最优价值函数

最优状态价值函数是所有策略产生的状态价值函数中，使状态 s 价值最大的函数：

$$v_*(s)=\max_\pi v_\pi(s) \tag{3-22}$$

最优动作价值函数是指所有策略产生的动作价值函数中，使状态-行为对(s,a)的价值最大的函数。

$$q_*(s,a)=\max_\pi q_\pi(s) \tag{3-23}$$

最优价值函数明确了 MDP 的最优可能表现，求解出最优价值函数后则认为 MDP 已经完成了求解，即将 MDP 的求解转换为对最优价值函数的求解。

3.3 算法流程

前文介绍了强化学习这一类算法的基本概念以及贝尔曼方程的推导，在本节仅选取强化学习算法中的经典算法 Q-Learning 来进行详细介绍。Q-Learning 算法是强化学习算法中 Value-based 类的算法，算法的主要思想是将状态和动作构建成一张 Q 表来存储 Q 值，然后根据 Q 值来选取能够获得最大收益的动作。

在贝尔曼方程中，当 P 和 R 函数已知时，可以通过迭代得到最优策略，而当 P 和 R 函数未知的时候，即为强化学习的问题设置时，可以有两种主要的建模求解最优策略方式。一类是基于模型的方法，这些方法通过在环境中尝试搜集转移概率和奖励函数的观测值，对问题进行建模，然后使用传统的 MDP 动态规划算法求解最优策略，这些方法包括 PEGASUS、RMAX、Fitted R-MAX；另一类便是与模型无关的方法，这些方法不显式地拟合 P 或 R 函数，而是直接逼近值函数或直接优化策略本身，本书重点介绍与模型无关的方法。

在模型无关的强化学习算法中，算法并不显式地对 P 和 R 进行建模，而是将其隐式地表达在其他模型构件中，通过对这些模型构件进行求解以找到最优策略，其大致可以分为基于值函数的方法和基于直接策略搜索的方法，前者通过估计值函数来得到最优策略，而后者则尝试直接对策略进行参数建模，通过优化某个方法来指导策略参数的更新。这些方法大部分都遵循“策略评估—策略改进”的迭代框架，在每次迭代中，对当前策略进行评估：使用当前策略在环境中进行采样，获取轨迹数据，就此数据计算或更新 Q 函数，然后更新下一步策略。

因此，大部分强化学习算法的更新过程本质上都是在不断地根据环境反馈来改进策略，只不过表现形式各不相同。要指出的另一点是，如果策略评估的数据是完全使用当前的策略来进行采样的，那么策略很容易陷入局部解。因此，在数据采样过程中，需要有效引入探索，使得智能体有机会执行一些和当前策略不同的动作。而对某个状态，将同时有可能采样带更优的动作或更差的动作，其带来的判别信息对于后面的策略提升是至关重要的。

由于强化学习的设置，虽然无法直接通过值的迭代来计算 Q，但可以通过蒙特卡洛采样得到累积期望奖励的一个近似，即：

$$q_\pi(s,a)=\frac{1}{m}\sum_{i=1}^{m}R(\tau_i) \tag{3-24}$$

其中，τ_i 是从(s,a)之后执行策略 π 得到的轨迹数据，$R(\tau_i)$是这条轨迹上的累积奖励，当然这里 m 越大近似越精确，但如果每一步迭代都进行 m 次轨迹的采样，并且为了估计得更为精准 m 还需要足够大，这在实际中几乎是不可行的，因此一种直接的改进是，每次采样轨迹后，对轨迹中的每一对 s 和 a 进行增量更新 $q_\pi(s,a)$。

$$q_\pi(s,a)=\frac{c(s,a)q(s,a)+R}{c(s,a)+1} \tag{3-25}$$

这里 $c(s,a)$是状态动作对(s,a)的更新次数，于是得到了经典的蒙特卡洛强化学习算法。对式(3-25)进行改写可以得到等价的更新形式。

$$q_\pi(s,a)=q_\pi(s,a)+\alpha(R-q_\pi(s,a)) \tag{3-26}$$

这里 $\alpha=\dfrac{1}{c(s,a)+1}$。实际上，可以对这个更一般的形式进行修改，$\alpha$ 可以设置为一个超参数，同时 R 通过当前奖励和下一个状态动作对的值函数之和进行近似，即 $R\approx r_t+\gamma q_\pi(s_{t+1},a_{t+1})$，这就得到了著名的时序差分方法，即

$$q_\pi(s_t,a_t)=q_\pi(s_t,a_t)+\alpha(r_t+\gamma q_\pi(s_{t+1},a_{t+1})-q_\pi(s_t,a_t)) \tag{3-27}$$

这样一来，不需要运行整个完整的轨迹就可以对模型进行更新，但同时也会引出来一个问题，那就是 a_{t+1} 该如何选取。一种选择就是使用采样轨迹中在下一个状态上实际执行的动作，这样的更新也被称为“on-policy”，对应的就是 SARSA 算法，因为其需要记录下一个状态上的实际执行动作，所以其训练数据通常为 5 元组$(s_t,a_t,r_t,s_{t+1},a_{t+1})$，这也正是其名字的由来。另外一种选择就是，使用当前策略，为下一个状态计算一个最优动作，即 $a_{t+1}=\pi(s_{t+1})$，这样的更新也被称为“off-policy”，对应的就是 Q-Learning 算法。

Q-Learning 是一种无模型的强化学习算法，智能体在与环境互动的过程中学习最优策略，无须了解环境的完整动态模型。也就是说，Q-Learning 是一种用于学习在不确定环境中做出决策的方法，是一种基于值函数的方法，其核心思想是通过迭代更新一个动作价值函数 $Q(s,a)$，该函数评估在给定状态 s 下采取特定动作 a 后的价值，以最大化累积奖励。更新过程遵循贝尔曼方程，同时利用了探索和利用的概念，以平衡对未知状态的探索和已知有利路径的利用。以下是 Q-Learning 算法的主要步骤，如图 3-5 所示。

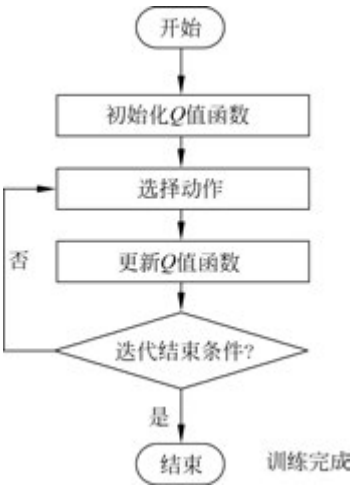

图 3-5　Q-Learning 流程图

【步骤 1】　初始化 Q 值函数：初始化一个 Q 值函数 $Q(s,a)$，其中，s 表示状态，a 表示动作。通常，可以将所有 Q 值初始化为零或者随机值。

【步骤 2】　选择动作：根据某种策略（例如 ε-greedy 策略）选择一个动作来执行。ε-greedy 策略以 ε 的概率选择一个随机动作，以 1−ε 的概率

选择当前状态下具有最高 Q 值的动作。

【步骤 3】 执行动作并观察奖励和下一个状态：执行选择的动作，并观察由环境返回的奖励以及下一个状态。

【步骤 4】 更新 Q 值函数：使用 Q-Learning 更新规则来更新 Q 值函数：$Q(s,a)\leftarrow Q(s,a)+\alpha\cdot[R(s,a)+\gamma\cdot\max_{a'}Q(s',a')-Q(s_t,a_t)]$，其中，$\alpha$ 是学习率（通常为一个小正数），$R(s,a)$是状态 s 下采取动作 a 获得的即时奖励，γ 是折扣因子（介于 0～1 之间，表示未来奖励的重要性），$\max_{a'}Q(s',a')$表示状态 s'下所有可能动作的 Q 值中的最大值。

【步骤 5】 重复步骤 2～4：重复选择动作、执行动作、观察奖励和状态、更新 Q 值函数的过程，直到达到某个停止条件，例如，达到最大迭代次数或者 Q 值函数收敛。

【步骤 6】 收敛和策略提取：当 Q 值函数收敛时，可以使用它来提取最佳策略。最佳策略通常是选择具有最高 Q 值的动作。

【步骤 7】 结束：算法终止，完成训练。

Q-Learning 是一个基本的强化学习算法，用于解决离散动作和状态的问题。对于连续动作和状态的问题，可以使用深度 Q 网络（DQN）等方法来扩展 Q-Learning。

3.4 概念详解

本书通过引入一个具体数值案例对强化学习中包含的名词概念及实际应用流程进行详细讲解。

【数值案例】 求解下述二元函数的最大值。

$$\begin{cases}\max f(x_1,x_2)=x_1^2-x_2^2\\ \text{s.t. } x_1\in\{0,1,2,3,\cdots,31\}\\ \qquad x_2\in\{0,1,2,3,\cdots,31\}\end{cases}\tag{3-28}$$

本文主要利用 Q-Learning 算法来进行问题的求解。

在用 Q-Learning 算法来解决优化问题的时候，首先需要的就是对问题进行强化学习模型建模，把求解最优化问题转换为强化学习问题，利用强化学习来解决最优化问题。

具体来说，要用 Q-Learning 对优化问题建模，需要定义以下几个要素。

（1）状态（State）：表示优化问题的当前状态，如当前的解、当前的目标函数值等。

（2）动作（Action）：表示优化问题的可行操作，如改变某个变量的值、增加或减少某个约束等。

（3）奖励（Reward）：提供了对智能体行为的即时反馈，用于评估某个动作在某一状态下的好坏，从而影响其未来的决策。

（4）策略（Policy）：表示优化问题的求解过程，即在每个状态下选择什么动作的规则。

有了这些要素，就可以使用 Q-Learning 算法来学习最优的策略，从而求解优化问题。强化学习的算法可以分为基于模型的方法和无模型的方法。基于模型的方法需要通过和环境交互来对环境进行建模，然后再利用这个模型做出动作规划或策略选择。无模型的方法不需要对环境建模，而是直接通过和环境交互来学习一个价值函数或者策略函数。本文主要利用无模型方法中的 Q-Learning 算法来求解二元函数的最大值。

Q-Learning 算法的基本术语如表 3-1 所示。

表 3-1　术语列表

术　语	表　征
状态	表示优化问题的当前状态
动作	表示优化问题的可行操作
奖励	提供了对智能体行为的即时反馈，用于评估某个动作在某一状态下的好坏，从而影响其未来的决策
策略	表示优化问题的求解过程，即在每个状态下选择什么动作的规则
Q 值	智能体在某一个状态下采取某一个动作能够获得的 Q 值的期望
Q 表	一张存储状态和动作对应 Q 值的表格，指导智能体选择动作
Q 表更新	通过迭代更新 Q 表，使得 Q 表学习到更多指导智能体的策略信息
学习率	定义了新获得的信息覆盖旧信息的程度
折扣因子	一种用来调节未来奖励的重要性的参数
探索策略	一种选择动作的策略，平衡探索与利用，避免陷入局部最优

3.4.1　定义状态和动作

1. 定义状态

在 Q-Learning 中，状态是指智能体所处的环境的特征，它可以影响智能体的决策和奖励。状态是智能体与环境交互的基础，也是智能体学习的对象。状态的集合称为状态空间。状态空间可以是离散的或者连续的，也可以是有限的或者无限的，这取决于具体的问题和环境。例如，在下棋的问题中，状态空间是有限的和离散的，每个状态就是棋盘上的棋子分布。在控制机器人的问题中，状态空间可能是无限的和连续的，每个状态就是机器人的位置和姿态。

定义状态的方法有很多，主要取决于问题的复杂度和可观测性。当状态空间、动作空间和奖励都是有限的时候，就是一个有限的马尔可夫过程。而且，状态转移和奖励函数都是确定的，这也是一个确定性环境。当状态空间和动作空间都是有限的，就可以用一个二维表格来存储每个状态-动作对应的 Q 值，这是最简单的方法，但是不适用于状态空间过大或者连续的情况。当状态空间是连续的或者高维的，可以用一些特征提取的方法来降维或者离散化，如聚类等，这样可以减少状态空间的大小，但也有可能会损失一些信息。也可以使用一些函数逼近的方法来近似 Q 函数，如线性回归、决策树等，实现状态空间的泛化，但是需要选择合适的函数形式和参数。

在本文要求解的数值案例中，x_1、x_2 的取值是离散的，而且维度较低，因此采用二维表格来存储状态空间和动作更为合适。对于这个数值案例中有两个变量的函数，每个状态可以由元组(x_1,x_2)表示，代表了二元函数的两个变量的值。Q-Learning 一般需要用于离散状态空间，在求解二元函数最大值的问题中，需要变量的值为离散的，即 x_1、x_2 从集合$\{0,1,2,3,\cdots,31\}$中取值。这样状态空间就变为有 $32\times32=1024$ 个可能的状态。

2. 定义动作

在 Q-Learning 中，动作是指每个状态下可以选择的动作，它会影响智能体的状态转移

和奖励。动作是智能体与环境交互的方式，也是智能体学习的关键。动作的选择决定了智能体能否达到最优的策略和最大的累积奖励。

动作的集合也称为动作空间。动作空间可以是离散的或者连续的，也可以是有限的或者无限的，这取决于具体的问题和环境。例如，在下棋的问题中，动作空间是有限的和离散的，每个动作就是在棋盘上放置一颗棋子。在机器人的问题中，动作空间可能是无限的和连续的，每个动作就是机器人的某个关节的角度或者速度。

在数值案例中，由于变量的取值是离散的，可以把变量的取值看成动作，那么动作也就是离散的，为了减少动作空间的大小，在这个二元函数求解最大值的问题中，可以定义以下几个操作为动作。

(1) 将 x_1 的值增加一个单位。例如，如果当前 x_1 的值是 5，那么执行这个动作后 x_1 的值变为 6。

(2) 将 x_1 的值减少一个单位。例如，如果当前 x_1 的值是 5，那么执行这个动作后 x_1 的值变为 4。

(3) 将 x_2 的值增加一个单位。例如，如果当前 x_2 的值是 5，那么执行这个动作后 x_2 的值变为 6。

(4) 将 x_2 的值减少一个单位。例如，如果当前 x_2 的值是 5，那么执行这个动作后 x_2 的值变为 4。

3.4.2 定义奖励

当状态空间和动作空间确定之后，状态转移函数也随即确定，但是奖励函数仍然是一个未知数。奖励函数定义的是状态与动作之间的数值关系，而我们要解决的问题并非一个天然存在的马尔可夫决策链，这样的数值关系并不存在。因此，一个重要的步骤就是要把我们要达到的目标(在数值案例中就是函数值最大)转换为具体的奖励函数，在学习过程中引导智能体完成设定的目标。

在强化学习中，奖励函数的设置非常关键，因为奖励函数定义了智能体与环境交互时的反馈。对于给定的二元函数问题，合理的奖励函数的设置是确保智能体可以找到有效策略的关键。奖励函数反映了智能体的目标，即最大化累积奖励。奖励函数可以是确定性的或随机的，也可以是稀疏的或密集的，这取决于具体的问题和环境。

一般来说，奖励函数的设置应该遵循以下几个原则。

(1) 一致性：奖励函数应该与真实的任务性能指标一致，即奖励函数的最大化应该导致性能指标的最大化。

(2) 简单性：奖励函数应该尽可能简单和稀疏，即只在关键的状态-动作对上给予奖励，避免过多的中间奖励或者惩罚。

(3) 安全性：奖励函数应该避免引导智能体做出危险或者不可逆的行为，即奖励函数应该考虑潜在的风险和代价。

(4) 可学习性：奖励函数应该使得智能体能够有效地从奖励信号中学习，即奖励函数应该提供足够的信息和梯度。

具体来说，一般有以下几种方法。

(1) 人工设计：这是最常见的方法，即由人类专家根据任务的需求和经验来设计奖励

函数，如设定一些固定的数值或者函数来表示奖励。这种方法的优点是直观和灵活，但是缺点是容易出现不一致性、过拟合或者不安全性的问题。

（2）逆向强化学习：这是一种从示范中学习奖励函数的方法，即通过观察一个或多个优秀的智能体或者人类的行为，来推断出它们所遵循的奖励函数，然后用这个奖励函数来指导自己的学习。这种方法的优点是可以自动地获取奖励函数，但是缺点是需要大量的示范数据，而且可能存在多个奖励函数与示范一致的问题。

（3）交互式强化学习：这是一种通过与人类交互来学习奖励函数的方法，即通过让人类提供一些反馈，如评分、指导、示范或者偏好，来不断地更新和改进奖励函数，然后用这个奖励函数来指导自己的学习。这种方法的优点是可以适应人类的需求和意图，但是缺点是需要人类的持续参与，而且人类的反馈可能存在噪声、不一致或者不完整的问题。

对于本问题，我们的目标是求二元函数的最大值。因此，奖励函数应该尽可能引导智能体探索得到更大的函数值，直接使用函数值作为奖励即可。

$$\text{reward} = f(x_1, x_2) = x_1^2 - x_2^2 \tag{3-29}$$

使用二元函数值直接作为奖励函数的优点就是非常直接和明确。智能体会被引导直接最大化这个函数，因为这是智能体可以获得的直接奖励。

为了优化奖励的过程，还需要考虑智能体的边界和非法动作，针对这些行为设置合理的惩罚。需要对超越边界的动作施加惩罚。例如，如果 x_1 已经是 0，再选择“减少 x_1”这个动作就是非法的。在这种情况下，可以给予一个大的惩罚 $-M$（M 为一个无限大的值），以确保智能体避免采取这样的动作。同样地，若 x_1 超越了 31 也是如此。

设置奖励的时候还可以考虑增加考虑前后状态变化的奖励，即可以根据 $f(x_1, x_2)$ 在采取某个动作之前和之后的差异给予奖励。例如，如果函数增加了，那么奖励可以是这个增量的正值，如果减少了则是负值。

3.4.3　策略迭代

在强化学习中的 Q 函数是指导智能体探索最优策略的重要组成部分，而在 Q-Learning 中的 Q 表就是一种用于存储每个状态-动作对应 Q 值的数据结构。Q 值是一种用来估计在给定状态下采取特定行动后可以获得的未来总回报的函数。Q 表可以用一个二维数组或者字典来实现，其中每一行或者键表示一个状态，每一列或者值表示一个动作。Q 表的大小取决于状态空间和动作空间的大小。Q-Learning 的核心思想是通过不断地与环境交互，更新和优化 Q 表，从而学习到一个最优的策略，即在每个状态下选择 Q 值最大的动作。

Q 表的作用是记录和更新智能体对每个状态-动作对的价值评估，指导智能体做出最优的决策，提高其长期的累计奖励。Q 表是 Q-Learning 算法的核心组成部分，也是一种基于表格的强化学习方法。

1. 初始化 Q 表

Q 表的设置应该与具体问题相适应，在本书的数值案例中，该表的大小应该为 $32 \times 32 \times 4$（x_1 有 32 个可能的值，x_2 也有 32 个可能的值，有 4 个可能的动作），即 Q 值表有 1024×4 个位置值，见表 3-2。整个算法都在一直不断更新 Q 表的值，然后再根据新的值来判断在某个状态应该采取怎样的动作。

表 3-2 Q 表初始化示例

状　态	动　作			
	x_1+1	x_1-1	x_2+1	x_2-1
$(x_1=0,x_2=0)$	0	0	0	0
$(x_1=0,x_2=1)$	0	0	0	0
…	…	…	…	…
$(x_1=31,x_2=30)$	0	0	0	0
$(x_1=31,x_2=31)$	0	0	0	0

2. 设置 Q-Learning 参数

在强化学习 Q-Learning 算法中，还需要进一步设置探索的参数。

Q-Learning 中探索参数设置的作用是调节智能体在探索和利用之间的平衡。探索是指智能体尝试一些未知或不确定的动作，以获取更多的环境信息和奖励信号。利用是指智能体根据已有的信息，选择最优或最有利的动作，以获取最大的即时奖励。探索和利用是强化学习中的一个基本的权衡问题，如果只进行探索，智能体可能会错过一些高奖励的机会；如果只进行利用，智能体可能会陷入局部最优，无法发现更好的策略。

主要的探索参数设置有以下几个。

(1) 学习率 α 定义了新获得的信息覆盖旧信息的程度，取值范围为 0～1。Q-Learning 中学习率是一种用来控制 Q 值更新幅度的参数，通常用 α 表示。Q 值是一种用来估计在给定状态下采取特定行动后可以获得的未来总回报的函数。Q-Learning 的核心思想是通过不断地与环境交互，更新和优化 Q 表，从而学习到一个最优的策略，即在每个状态下选择 Q 值最大的动作。学习率的作用是调节 Q 值的更新速度和稳定性。学习率的值在[0,1]，表示在每次更新时，新的信息占旧的信息的比重。学习率的值越大，表示新的信息越重要，Q 值更新越快，但也越不稳定。学习率的值越小，表示旧的信息越重要，Q 值更新越慢，但也越稳定。学习率的设置需要根据具体的问题和环境来调节，一般来说，学习率的初始值应该较大，以便智能体能够快速地适应环境，然后随着学习的进行，学习率的值应该逐渐减小，以便智能体能够收敛到最优的 Q 值。不同取值的具体效果如下。

$\alpha=0$ 时：智能体不会学习任何新的信息。

$\alpha=1$ 时：智能体只考虑最新的信息。

$0<\alpha<1$ 时：智能体既考虑过去的知识，也考虑新的奖励。

(2) 折扣因子 γ 是一种用来调节未来奖励的重要性的参数。Q-Learning 的目标是最大化累积奖励，即从当前状态开始，智能体能够获得的所有未来奖励的总和。折扣因子决定了智能体对未来奖励的折现，即越远的未来奖励会被乘以一个越小的系数，从而降低其对当前决策的影响。折扣因子的值在[0,1]，表示在每次更新时，下一个状态的最大 Q 值占当前奖励的比重。折扣因子的值越大，表示智能体越重视未来的回报，越倾向于探索更多的状态，以寻找更高的长期回报，但也可能会错过一些高即时回报的机会；折扣因子的值越小，表示智能体越重视即时的回报，越倾向于利用已知的状态，以获取更高的短期回报，但也可能会陷入局部最优，无法发现更好的策略。折扣因子的设置需要根据具体的问题和环境来调节。

一般来说，折扣因子的初始值应该较大，以便智能体能够广泛地探索环境，然后随着学习的进行，折扣因子的值应该逐渐减小，以便智能体能够收敛到最优的策略。若是问题的目标需要长期地探索才能取得，那么折扣因子的取值应该大一些；反之，如果关心即时奖励更多，则折扣因子的取值应该小一些。不同取值的具体效果如下。

$\gamma=0$ 时：智能体只考虑当前奖励，完全忽略未来奖励。

$\gamma=1$ 时：智能体会评估每一步的长期奖励。

$0<\gamma<1$ 时：智能体对未来的奖励加上适当的权重。

(3) 此外，还需要设置探索策略相关参数。在早期的探索学习中，探索环境通常很重要，而随着 Q-Learning 算法学习到的经验增加，利用已知信息的重要性也会增加。而 ε-greedy 是一种常用的平衡两者的策略，在本案例中就可以使用这一策略。以 ε 的概率来选择一个随机动作，以 $1-\varepsilon$ 的概率来选择具有最高 Q 值的动作。

3. Q 表更新

Q-Learning 算法最关键的就是通过不断地迭代更新 Q 表来获取最优策略。

继续基于数值案例展开说明，现在已经初始化了一个有 1024 个状态空间和 4 个动作的 Q 表，这个 Q 表会在每一个训练迭代的过程中时刻记录着外界环境的收益，智能体探索外界环境，并接收外界环境的奖励，直到收益最大值收敛。训练迭代的目的就是要强化智能体的“大脑”(Q 表)，训练迭代的次数越多，则 Q 表被训练优化得更好，当一个 Q 表已经被训练得很完善的时候，智能体就可以轻松找到奖励最大的状态了。每一次的迭代探索过程称为幕(Episode)，具体来说，就是智能体与环境之间的一系列交互，通常以智能体采取一个终止动作(如达到一个目标状态或时间限制)结束。一个幕通常包括多个状态和动作的序列，每个状态都对应一个即时奖励和一个新的状态。通过多次幕的交互，智能体可以学习到如何在环境中采取最优行动以最大化累积奖励。幕是强化学习中的一个基本单位，用于指导智能体的学习过程。

接下来，进一步解读 Q-Learning 算法迭代收敛求解最大值的过程。算法流程伪代码如下。

Q-Learning(离轨策略下的时序差分控制)算法，用于预测 $\pi\approx\pi_*$：

对所有 $s\in S^+$，$a\in A(s)$，任意初始化 $Q(s,a)$，其中，Q(终止状态，.)$=0$

对每一幕：

　　初始化 S

　　对幕中的每一步循环：

　　　　使用从 Q 中得到的策略(例如 ε-greedy)，在 S 处选择 A

　　　　执行 A，观察到 R，S'

　　　　$Q(S,A)\leftarrow Q(S,A)+\alpha[R+\gamma\max_a Q(S',a)-Q(S,A)]$

　　　　$S\leftarrow S'$

　　直到 S 是终止状态

选择循环开始的前两个幕进行推演举例。

在 Q-Learning 中，R 表是一个奖励矩阵，它记录了从一个状态到另一个状态的转移所获得的奖励，也就是环境对智能体的反馈。R 表是 Q-Learning 算法的输入之一，它可以是预先

定义的，也可以是动态学习的。R 表的作用是提供 Q 表更新的基础，也就是贝尔曼方程中的奖励项。R 表反映了环境的特性，如哪些状态是目标状态，哪些状态是障碍状态，哪些状态是中间状态等。R 表可以帮助智能体找到最优的策略，也就是最大化累积奖励的动作序列。

继续基于数值案例构建一个 R 表，里面的数值表示各个行动的奖励值，可以使用目标函数值来计算，得到的 R 表如表 3-3 所示。

表 3-3 R 表

状　态	动　作			
	x_1+1	x_1-1	x_2+1	x_2-1
$(x_1=0,x_2=0)$	1	$-M$	-1	$-M$
$(x_1=0,x_2=1)$	0	$-M$	-4	0
…	…	…	…	…
$(x_1=31,x_2=30)$	$-M$	0	0	120
$(x_1=31,x_2=31)$	$-M$	-61	$-M$	61

首先，第一幕从初始化的 0 矩阵 Q 表开始，如表 3-4 所示。这个时候没有采取任何动作，因此，表格中所有元素均为 0，需要通过后续的动作不断更新 Q 表。

表 3-4 Q 表 1

状　态	动　作			
	$a_1(x_1+1)$	$a_2(x_1-1)$	$a_3(x_2+1)$	$a_4(x_2-1)$
$(x_1=0,x_2=0)$	0	0	0	0
$(x_1=0,x_2=1)$	0	0	0	0
…	…	…	…	…
$(x_1=31,x_2=30)$	0	0	0	0
$(x_1=31,x_2=31)$	0	0	0	0

初始状态为 S_1，观察矩阵第 1 行，包含两个可行的值，即状态 $S_1(x_1=0,x_2=0)$ 下的 $a_1(x_1+1)$ 和 $a_3(x_2+1)$ 动作，随机选取 $a_3(x_2+1)$，来到下一个状态 $S_2(x_1=0,x_2=1)$。

当智能体到达新的状态 S_2 之后，对应的可能行为变为 $a_1(x_1+1)$、$a_3(x_2+1)$ 和 $a_4(x_2-1)$。根据 Q 值更新公式可以得到(取 $\alpha=0.8,\gamma=0.9$)：

$$Q(s_1,a_3)=Q(s_1,a_3)+\alpha\times[R(s_1,a_3)+\gamma\times\max(Q(s_2,a_1),Q(s_2,a_3),Q(s_2,a_4))-Q(s_1,a_3)]=0+0.8\times(-1+0.9\times\max(0,-4,0)-0)=-0.8$$

达到新状态 S_2 后，智能体的“大脑”中的 Q 表更新为如表 3-5 所示。

表 3-5 Q 表 2

状　态	动　作			
	$a_1(x_1+1)$	$a_2(x_1-1)$	$a_3(x_2+1)$	$a_4(x_2-1)$
$(x_1=0,x_2=0)$	0	0	-0.8	0
$(x_1=0,x_2=1)$	0	0	0	0
…	…	…	…	…
$(x_1=31,x_2=30)$	0	0	0	0
$(x_1=31,x_2=31)$	0	0	0	0

接下来，进行下一幕继续更新 Q 表，那么在智能体到达了状态 S_2 之后，由于状态 S_2 的 Q 值都是一样的，需要在 $a_1(x_1+1)$、$a_3(x_2+1)$和 $a_4(x_2-1)$三个动作中随机选取一个动作。假设，选取动作 $a_3(x_2+1)$，则将到达状态 $S_3(x_1=0,x_2=2)$。在抵达状态 S_3 之后可以选取动作 $a_1(x_1+1)$、$a_3(x_2+1)$和 $a_4(x_2-1)$，三个动作的 R 值分别为 -3、-9 和 -1。根据 Q 值更新公式可以得到(假设 $\alpha=0.8,\gamma=0.9$)：

$$Q(s_2,a_3)=Q(s_2,a_3)+\alpha\times[R(s_2,a_3)+\gamma\times\max(Q(s_3,a_1),Q(s_3,a_3),Q(s_3,a_4))-Q(s_2,a_3)]=0+0.8\times(-4+0.9\times\max(-3,-9,-1)-0)=-3.92$$

则现在状态 S_3 变成了当前状态。至此，智能体的“大脑”中的 Q 表更新为如表 3-6 所示。

表 3-6　Q 表 3

状　态	动　作			
	$a_1(x_1+1)$	$a_2(x_1-1)$	$a_3(x_2+1)$	$a_4(x_2-1)$
$(x_1=0,x_2=0)$	0	0	-0.8	0
$(x_1=0,x_2=1)$	0	0	-3.92	0
…	…	…	…	…
$(x_1=31,x_2=30)$	0	0	0	0
$(x_1=31,x_2=31)$	0	0	0	0

大脑在获取下一步环境的实际情况之后再进行学习，学习函数对 Q 表更新的重要参数之一就是获取下一步环境的实际情况。具体来说，在进行学习过程时，Q-Learning 的大脑对象会根据所处的当前环境对各个动作的预测得分和下一步环境的实际情况(最大得分)对当前环境的 Q 表进行更新。

接下来需要更多的迭代，不断更新 Q 表，当 Q 表中的值逐渐多了起来之后，动作的选择不再是完全随机的了。例如，表 3-6 的状态 S_1 中，由于 -0.8 小于别的值，所以 -0.8 这个动作被选择的概率就会降低，而选择其他 Q 值大的动作。在这里 ε-greedy 策略还可以进一步发挥作用，随着 Q 表中的信息不断增加，智能体将以 ε 的概率来选择一个随机动作，以 $1-\varepsilon$ 的概率来选择具有最高 Q 值的动作。这样可以让智能体具有一定的探索精神从而避免陷入局部最优。

假设每一回合需要进行 200 幕的试探，Q 表就会更新 200 次，而智能体经过多个回合的训练，便会得到一个成熟的且收敛的 Q 表，可以指导智能体快速搜索到二元函数的最大值。需要注意的是，对于连续的函数值和连续的输入空间，直接应用上述 Q-Learning 方法可能并不高效。在这种情况下，可能需要使用函数逼近技术(如神经网络)来近似 Q 函数，或者考虑使用其他针对连续动作和状态空间的强化学习算法。

3.5　算法改进

Q-Learning 是强化学习领域的一种基础算法，可以通过多种方式进行改进，以获得更好的性能、更快的收敛速度或更好的适应性。这里介绍一些提高 Q-Learning 效率的策略。

(1) 学习率调度：随着时间的推移调整学习率。一开始使用较高的学习率可以帮助快速收敛，但逐渐降低它可以提供更稳定的更新。

(2) 优先经验回放：并不是所有的经验在学习中都是平等的，某些经验可能含有罕见但重要的信息。优先回放根据其 TD 错误的大小更频繁地采样重要的经验。

(3) 双重 Q-Learning：为了处理 Q 值的过高估计偏见，可以使用两个 Q 网络，从一个 Q 网络中选择动作但用另一个评估它们。

(4) 自适应学习率：使用像 RMSProp 或 Adam 这样的优化算法，根据最近的梯度大小调整学习率，尤其是当使用神经网络时，可以提高 Q-Learning 的收敛速度。

Q-Learning 算法最成功的改进无疑就是 DeepMind 团队在 2013 年提出、2015 年改进的 Deep Q Network (DQN)。DQN 是深度强化学习(Deep Reinforcement Learning，DRL)的开山之作，其核心思想是使用神经网络来近似 Q 函数，从而解决具有高维输入空间的决策问题。

事实上，Q-Learning 算法的迭代更新都是针对离散状态和动作的，实现通常是将值函数用状态空间到动作空间的一张表来表达。然而，在大规模状态/动作问题空间(包括连续状态/动作空间问题)中，值表形式的值函数所需要的存储空间远远超过了现代计算机的硬件条件，使得这些经典的算法不再适用。这也是强化学习中著名的“维度灾难”问题。值函数估计是解决维度灾难问题的主要手段之一，其主要思想是将状态值函数或动作值函数进行参数化，将值函数空间转换为参数空间，达到泛化的目的。同样以 Q 函数为例，将其参数化建模为 $\hat{Q}(s,a|\theta)$。实际上，在重新审视时序差分公式时，不难发现其本质上就是下式的一步随机梯度下降：

$$J=(y_t-Q(s_t,a_t))^2 \tag{3-30}$$

其中，$y_t=r_t+\gamma q_\pi(s_{t+1},a_{t+1})$。

因此，当参数化 Q 的表示时，可以同样通过最小化：

$$J(\theta)=(y_t-Q(s_t,a_t\mid\theta))^2 \tag{3-31}$$

其中，参数 θ 的更新公式如下。

$$\theta=\theta+\alpha(r_t+\gamma Q(s_{t+1},a_{t+1})-Q(s_t,a_t))\nabla_\theta\hat{Q}(s,a\mid\theta) \tag{3-32}$$

当 $\hat{Q}(s,a|\theta)$是一个神经网络模型时，就得到了 DQN 算法更新公式。

DQN 对 Q-Learning 的改进过程中，除了引入神经网络来估计 Q 值外，另一个主要贡献之一是引入了经验回放(Experience Replay)机制。在这种机制下，智能体会将每个时间步的经验(状态、动作、奖励、下一个状态)存储在一个回放缓冲区中，即记忆库(Memory)。在训练过程中，DQN 从回放缓冲区中随机采样一批经验，然后用这些经验来更新神经网络的权重。这种方法打破了数据间的相关性，提高了数据的有效利用，并有助于稳定学习过程。DQN 的另一个关键创新是使用了目标网络(Target Network)。在 DQN 中，维护了两个结构相同但参数不同的神经网络：一个 Q 值网络用于预测 Q 值，另一个目标网络用于生成目标 Q 值。目标网络的参数定期从 Q 值网络复制过来，但更新频率较低。这种方法可以减少目标值的变化，防止过拟合，从而提高训练稳定性和收敛速度。

DQN 算法的基本术语如表 3-7 所示。

表 3-7　DQN 算法的基本术语

术　语	表　征
Q 值网络	作为 Q 函数来表示优化问题的当前状态
目标网络	目标网络是一个固定的网络，用于计算目标值，通过减少目标值的变化来提高训练的稳定性
经验回放	将智能体的经验存储在一个经验池中，并从中随机抽样进行训练，以减少样本之间的相关性
记忆库	用于存储经验回放的缓冲区
损失函数	损失函数是一种用来衡量神经网络预测值和目标值之间差异的函数，它可以指导神经网络的参数更新，使得预测值更接近目标值
网络更新频率	延迟更新的目标网络用来计算 Q 的目标值，避免 Q 网络误差的“自激效应”，并借此来提高训练稳定性

3.6　工程案例

3.6.1　可重构制造系统调度问题

面向具有多品种、小批量特点的产品订单，在一个基于可重构机床的制造系统中如何合理地组织现有的制造资源，进行快速高效的产品生产是制造企业需要解决的关键问题。

具体来说，该制造系统由多台可重构机床组成。每台可重构机床具有多种构型，每种构型对应一种产品特征（工序）的加工。对于一台可重构机床来说，不同构型之间的转换需要一定的设置时间和成本。此外，产品订单由多种类型的产品组成，每种产品的加工数量也各不相同。一种产品包含多个特征（工序），特征的加工存在固定的先后顺序。具体的问题假设如下。

（1）针对每台可重构机床切换不同构型时的设置时间及成本是固定相同的。

（2）可重构机床加工期间针对某一产品特征加工时间固定不考虑其运行故障。

（3）一台可重构机床的一种构型至多对应一种产品的一个特征的加工，即不考虑一种可重构机床的构型可以生产一种产品的多个特征的加工的情况。

（4）每种产品类型的同一产品特征必须在一台可重构机床上连续生产，不允许间断及生产任务拆分。

（5）在零时刻所有可重构机床均可用，所有产品均可以被加工。

3.6.2　可重构制造系统调度数学模型

首先，针对问题中涉及的数学符号进行如下表述。

索引

$[i,j]$　产品类型索引（$[i,j=(1,2,\cdots,P)]$）。

k,l　产品 i 的特征（工序）索引（$[k,l=(1,2,\cdots,F_i)]$）。

m　可重构机床索引（$[m=(1,2,\cdots,M)]$）。

c 可重构机床 m 的构型索引($[c=(1,2,\cdots,C_m)]$)。

常量

F_i 产品 i 的特征总数量。

C_m 可重构机床 m 的构型总数量。

N_i 产品 i 的加工数量。

$\mathrm{Pt}_{i,k}^{m}$ 产品 i 的第 k 个特征在可重构机床 m 上的加工时间。

$\mathrm{Pc}_{i,k}^{m}$ 产品 i 的第 k 个特征在可重构机床 m 上的加工成本。

Rt_m 可重构机床 m 的构型切换时间。

Rc_m 可重构机床 m 的构型切换成本。

$B_{i,k}^{m,c}$ 产品 i 的第 k 个特征能采用可重构机床 m 的第 c 个构型加工则为 1,否则为 0。

$W_{i,k,j,l}^{m}$ 如果在可重构机床 m 上产品 i 的第 k 个特征的加工构型与产品 j 的第 l 个特征的加工构型不同则为 1,否则为 0。

L 足够大的数。

决策变量及辅助变量

$\mathrm{ts}_{i,k}$ 产品 i 的第 k 个特征的生产开始时间。

$\mathrm{tc}_{i,k}$ 产品 i 的第 k 个特征的生产完成时间。

fc_i 产品 i 的最终生产完成时间。

$x_{i,k}^{m}$ 如果产品 i 的第 k 个特征选择可重构机床 m 进行加工则为 1,否则为 0。

$s_{i,k,j,l}^{m}$ 如果在可重构机床 m 上产品 i 的第 k 个特征加工完成后紧接着加工产品 j 的第 l 个特征则为 1,否则为 0。

基于上述符号构建了如下基于可重构机床的车间调度数学模型,其目标函数及约束条件具体描述如下。

Makespan 即最小化最大完工时间,用于表征产品生产的总体制造跨度。其中,fc_i 是产品 i 的最终完工时间,即产品 i 的最后一个特征 F_i 的生产完成时间。

$$\mathrm{Makespan}=\min(\max \mathrm{fc}_i) \quad \forall i \in \{1,2,\cdots,P\} \tag{3-33}$$

s. t.

约束 1:产品特征加工时间约束。

$$\mathrm{ts}_{i,k}+N_i\sum_{m=1}^{M}(\mathrm{Pt}_{i,k}^{m}x_{i,k}^{m})=\mathrm{tc}_{i,k} \quad \forall i \in \{1,2,\cdots,P\};\quad k \in \{1,2,\cdots,F_i\} \tag{3-34}$$

约束 2:产品前后特征的加工时间约束。

$$\mathrm{tc}_{i,k} \leqslant \mathrm{ts}_{i,k+1} \quad \forall i \in \{1,2,\cdots,P\};\quad k \in \{1,2,\cdots,F_i-1\} \tag{3-35}$$

约束 3:产品完工时间约束产品特征的完成时间小于产品的完工时间。

$$\mathrm{tc}_{i,k} \leqslant \mathrm{fc}_i \quad \forall i \in \{1,2,\cdots,P\};\quad k \in \{1,2,\cdots,F_i\} \tag{3-36}$$

约束 4:可重构机床上的时间占用约束。

$$\mathrm{tc}_{i,k}+\mathrm{Rt}_m W_{i,k,j,l}^{m}s_{i,k,j,l}^{m} \leqslant \mathrm{ts}_{j,l}+L(1-s_{i,k,j,l}^{m}) \quad \forall i,j \in \{1,2,\cdots,P\};$$
$$k \in \{1,2,\cdots,F_i\};\quad l \in \{1,2,\cdots,F_j\};\quad m \in \{1,2,\cdots,M\} \tag{3-37}$$

约束 5:表征机床构型切换矩阵。

$$W_{i,k,j,l}^{m}=\sum_{c=1}^{C_m}(B_{i,k}^{m,c}B_{j,l}^{m,c})\quad \forall i,j\in\{1,2,\cdots,P\};\quad k\in\{1,2,\cdots,F_i\};$$

$$l\in\{1,2,\cdots,F_l\};\quad m\in\{1,2,\cdots,M\} \tag{3-38}$$

约束 6：一台可重构机床的一种构型至多对应一种产品的一个特征的加工。

$$\sum_{c=1}^{C_m}B_{i,k}^{m,c}\leqslant 1\quad \forall i\in\{1,2,\cdots,P\};\quad k\in\{1,2,\cdots,F_i\};\quad m\in\{1,2,\cdots,M\} \tag{3-39}$$

约束 7：对于任意的一个产品特征必须被某一个可重构机床进行加工。

$$\sum_{m=1}^{M}x_{i,k}^{m}=1\quad \forall i\in\{1,2,\cdots,P\};\quad k\in\{1,2,\cdots,F_i\} \tag{3-40}$$

约束 8：在一台可重构机床上一个特征加工至多只有一个紧前特征加工，至多仅有一个紧后加工。

$$\sum_{i=1}^{P}\sum_{k=1}^{F_i}s_{i,k,j,l}^{m}\leqslant x_{j,l}^{m}\quad \forall j\in\{1,2,\cdots,P\};\quad l\in\{1,2,\cdots,F_j\};\quad m\in\{1,2,\cdots,M\} \tag{3-41}$$

$$\sum_{j=1}^{P}\sum_{l=1}^{F_j}s_{i,k,j,l}^{m}\leqslant x_{i,k}^{m}\quad \forall i\in\{1,2,\cdots,P\};\quad k\in\{1,2,\cdots,F_i\};\quad m\in\{1,2,\cdots,M\} \tag{3-42}$$

约束 9：定义变量的取值范围。

$$\mathrm{ts}_{i,k},\mathrm{tc}_{i,k},\mathrm{fc}_{i}\geqslant 0 \tag{3-43}$$

$$x_{i,k}^{m},s_{i,k,j,l}^{m}\in\{0,1\} \tag{3-44}$$

3.6.3　强化学习算法应用分析

使用强化学习算法求解可重构制造系统调度问题，需要重点解决以下三个问题。

(1) 如何有效地表示可重构制造系统的状态和动作空间，使得强化学习算法能够从中学习和探索？

(2) 如何设计合适的奖励函数，使得强化学习算法能够优化可重构制造系统的目标？

(3) 如何选择合适的强化学习算法，使得强化学习能够在复杂的可重构机床制造环境中稳定地收敛和泛化？

3.6.4　可重构制造系统状态表达和动作空间

1. 可重构制造系统状态表达

若要使用强化学习算法求解可重构制造系统的调度问题，首先需要将其系统的状态和动作空间合理地表达出来。参考前文章节的求解，基于可重构制造系统调度的问题特征，同样需要进行以下两个子决策问题的求解。

(1) 产品特征的加工顺序子决策问题。

在满足同一产品前后加工特征的顺序约束的基础上，需要考虑在可供选择的可重构机床上先生产哪一产品特征，以及后生产哪一产品特征。不同的产品特征生产顺序会影响最

终调度方案的整体完工时间。

(2) 加工产品特征的可重构机床配置子决策问题。

在满足同一时刻一台可重构机床至多只能用于一项产品特征加工的约束基础上,需要考虑针对不同产品的不同特征选择哪一可重构机床进行加工。不同的可重构机床配置方案会影响可重构机床的加工进程(例如,涉及方案是否需要进行构型切换等),进而影响最终调度方案的整体完工时间。

本案例在某一时刻的状态由完成加工特征数量、所有工件加工特征完成情况、加工设备当前构型情况、工件所在加工设备位置构成,以一个四元组对状态空间进行表示。

$$S=<N,F,C,P> \tag{3-45}$$

其中,N 表示完成加工特征数量,当完成加工特征数量与总计工件加工特征数量相等时,完成加工任务。F 是所有工件加工特征完成情况,定义 F_i 为工件 i 加工特征完成情况 $F_i=(f_{(i,1)},f_{(i,2)},\cdots,f_{(i,L_i)})$,$L_i$ 为工件 i 的加工特征数量,定义 $f_{(i,j)}$ 为 i 工件 j 加工特征的完成情况,当对应工件加工特征未完成加工时,$f_{(i,j)}=0$;当对应工件加工特征完成加工时,$f_{(i,j)}=1$,当前状态 s 时工件加工特征完成情况为 $F=(F_1,F_2,F_3,\cdots,F_n)$,$n$ 为需要加工的工件总数量。C 表示当前可重构制造系统中加工设备的构型情况,定义 c_k 为当前状态下加工设备 k 的构型情况,$c_k\in(1,2,3,\cdots,l_k)$,l_k 为加工设备 k 所能转换的构型数量。P 表示工件当前所处的位置,即当前工件所在加工设备的编号,定义 p_i 为工件 i 当前所在加工设备的设备编号,即工件 i 当前所在位置为 $p_i\in(1,2,3,\cdots,m)$,m 为可重构制造系统中加工设备的总数量。

2. 可重构制造系统动作空间

动作可以定义为 4 个部分:选择需要加工的工件,选择需要加工的工件加工特征,选择进行加工操作的机床,选择进行加工操作的机床的构型。基于此,动作空间可表示为 $A=((1,1,1,1),(1,1,1,2),(i,j,k,l),\cdots,(n,L_n,m,l_m))$,对于动作$(i,j,k,l)$,表示选择机床 k 的构型 l 对工件 i 的加工特征 j 进行加工。

3.6.5 可重构制造系统的环境设置和奖励表达

以最小化最大完工时间为目标,设置可重构制造系统集成式工艺-重构-调度问题的奖励函数如下所示。

$$R(s_t,a_t)=\begin{cases}\mathrm{VC}+(\mathrm{LT}_{i-1}-\mathrm{LT}_i), & (s,a)\in\text{情况 1}\\ \text{penalty value 1}, & (s,a)\in\text{情况 2}\end{cases} \tag{3-46}$$

式中,情况 1 表示在选择某一动作后,机床 k 的构型 l 对工件 i 的加工特征 j 进行加工正常进行时,将 $\mathrm{VC}+(\mathrm{LT}_{i-1}-\mathrm{LT}_i)$作为奖励,其中,VC 为一个价值常数,$\mathrm{LT}_i$ 为进行本次动作执行完成后,当前可重构制造系统中已经安排的加工特征中最晚完成加工的时间。LT_0 为初始时可重构制造系统中已经安排的加工特征中最晚完成加工的时间,由于此时没有对任意工件的任意特征进行加工安排,此时 $\mathrm{LT}_0=0$。

情况 2 表示所选动作不满足约束要求,例如,动作中的机床构型无法对对应工件的加工特征进行加工时;工件的加工特征需要在完成其他加工特征之后进行加工等情况,将获得

一个负值惩罚。

3.6.6　强化学习算法迭代

由于工程案例相对复杂，综合 DQN 算法的诸多优点，因此使用 DQN 算法来进行求解。DQN 算法流程如图 3-6 所示。

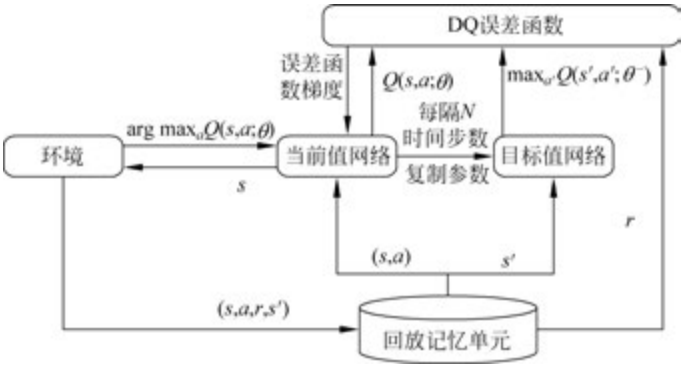

图 3-6　DQN 算法流程

1. 构建 DQN 神经网络

需要构建一个神经网络，用于估计 Q 值函数和计算目标的 Q 值。采用 PyTorch 框架进行神经网络的构建，用于强化学习 DQN 算法中策略函数的构建。

本书所构建的神经网络定义了三个全连接层，每个全连接层都设置一个权重矩阵，用于将输入向量映射到输出向量。权重矩阵的维度由输入和输出的大小决定，权重矩阵的初始值是从均值为 0，标准差为 0.1 的正态分布中随机采样的。进一步，选择激活函数，这里采用 ReLU 激活函数。ReLU 激活函数是一种常用的非线性激活函数，可以极大地提高神经网络的性能和效率，有效解决梯度消失的问题，实现网络的稀疏性，提高网络的非线性拟合能力。这样，就完成了 DQN 神经网络的构建。

2. 初始化 DQN 神经网络

在构建了神经网络的结构后，需要进一步初始化两个神经网络，即 Q 值网络和目标网络。两者的结构相同，但是参数不同，随机初始化两个神经网络的参数。同时还需要定义一系列的超参数，如学习率、折扣因子、探索率、记忆容量等，以及优化器、损失函数、记忆库等。

3. 动作选择函数和状态存储池的构建

构建一个动作选择函数，根据当前的状态，使用 ε-greedy 策略选择一个动作。ε-greedy 策略是一种在探索和利用之间平衡的策略，它以一定的小概率随机选择一个动作，否则选择当前状态下最大奖励值对应的动作。这样可以避免陷入局部最优，同时也可以利用已有的知识。本书所构建的 ε-greedy 策略的初始值设为 0.1，表示有 10％的概率随机选择动作，90％的概率选择最大奖励值对应的动作。

另外，还需要构建一个状态存储池，将每一步的状态、动作、奖励和下一个状态存储到记

忆库中，用于后续的学习。记忆库是一个循环队列，当达到最大容量时，会覆盖最早的记忆。

4. 设置学习函数

在设置了记忆库后，需要根据记忆库的数据更新网络的参数，使其逼近最优的动作值函数。设置一个 Q_NETWORK_ITERATION 超参数，表示每隔多少步更新一次目标网络的参数。如果当前的学习步数是 Q_NETWORK_ITERATION 的整数倍，就将评估网络的参数复制给目标网络，使其同步。这样可以保持目标网络的稳定性，避免目标动作值和当前动作值相互影响，导致不收敛的问题。然后将学习步数加 1，记录训练的进度。

设置一个超参数 BATCH_SIZE，从记忆库中随机抽取一批数据，表示每次训练的样本数量。将抽取的数据分别提取出状态、动作、奖励和下一个状态，转换为 PyTorch 的张量格式，用于后续的计算。

使用目标网络计算下一个状态下的动作值，得到一个 BATCH_SIZE × NUM_ACTIONS 的矩阵，表示每个状态下每个动作的动作值。然后使用 detach 函数，将该矩阵从计算图中分离，使其不参与梯度的计算，保持其不变。然后使用 max 函数，从矩阵中提取出每一行的最大值，得到一个 BATCH_SIZE×1 的向量，表示每个状态下最大的动作值，称为 q_next。

然后就可以计算目标函数值，根据公式 q_target＝batch_reward＋GAMMA×q_next，其中，batch_reward 是每个状态的奖励，GAMMA 是折扣因子，表示未来奖励的重要程度。目标动作值表示采取当前动作后的长期回报的估计。使用损失函数，计算评估动作值和目标动作值之间的均方误差，作为评估网络性能的指标。损失函数的定义是 $L(\omega)=\mathbb{E}[(R+\gamma\cdot\max_{a'}Q(s',a';\omega^{-})-Q(s,a;\omega))^2]$，其中，$\omega$ 是评估网络的参数，ω^{-} 是目标网络的参数，R 是奖励，γ 是折扣因子，Q 是动作值函数，s 是状态，a 是动作，s' 是下一个状态，a' 是下一个动作。损失函数的含义是，用目标动作值减去当前动作值的平方，作为评估当前动作值好坏的指标。如果当前的状态是终止状态，就将损失值添加到 self.loss_list 中，用于后续的分析和可视化。

最后使用优化器更新评估网络的参数，使其逼近最优的动作价值函数。优化器的方法是使用梯度下降法，即沿着损失函数的负梯度方向，以一定的学习率，更新评估网络的参数。具体步骤是：先将优化器的梯度清零，然后使用 backward 函数，计算损失函数对评估网络参数的梯度，最后使用 step 函数，根据梯度和学习率更新评估网络的参数。

5. 算法迭代主函数

前文已经构建 DQN 神经网络和动作、奖励计算函数，下面需要将各个函数和模块串联起来，在自定义的环境中训练一个智能体，使其可以通过迭代学习到最优的生产线排产调度策略。

首先，需要设定总的训练轮数(episodes)，还需要设定用于记录每一轮累积奖励的列表(reward_list)用于后续的分析和可视化。

开始一轮训练后，首先需要重置环境状态，得到初始环境状态，即一个两层的全为 0 的数列编码，策略为空列表。然后就可以使用循环来遍历每一步的行动，在每一步的行动中执行以下几个步骤。

(1) 将状态转换为神经网络输入格式,在本书的案例中是一个一维的张量。

(2) 使用前文设定的动作函数选择一个智能体的动作,并将其添加到策略的列表中去。

(3) 调用前文设定的环境,根据当前的状态和动作,得到下一个状态,奖励以及判断该轮训练是否结束。

(4) 将下一个状态也转换为神经网络的输入格式,即一个一维的张量。

(5) 使用前文设定的动作存储函数,将当前的状态、动作、奖励和下一个状态存储到记忆库中,用于后续的学习。

(6) 将奖励累加至累积奖励中,并将奖励添加至每一步的价值列表中去。

(7) 如果记忆库的数量达到一定的阈值,就使用前文设置的学习函数,从记忆库中抽取一批数据,更新评估网络的参数,使其逼近最优的动作值函数。如果当前状态是终止状态,则结束此轮训练。

(8) 将当前状态更新为下一个状态,继续循环。

在这一轮的训练结束后,将累积的奖励添加到 reward_list 中用于后续的分析和可视化。训练轮数没有结束,则重新开始新的训练轮数。

在结束了所有的训练轮数后便可以根据记录输出最好的结果和训练的迭代图。

3.7　基于 Python 的代码实现

3.7.1　构建神经网络

1. 导入依赖包,定义超参数

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import lizishengcheng

#hyper-parameters
#设置学习率、折扣率、初始探索率等超参数
BATCH_SIZE = 512
LR = 0.01
GAMMA = 0.9
EPISILO = 1
MEMORY_CAPACITY = 2000
Q_NETWORK_ITERATION = 100

#获取动作空间、状态空间大小
NUM_ACTIONS = len(lizishengcheng.Actions)
NUM_STATES = 1 + sum([i for i in lizishengcheng.Op_num]) + lizishengcheng.M_num + len
(lizishengcheng.Job_kind)
ENV_A_SHAPE = 0
```

2. 设置神经网络结构

```
    #神经网络参数设置
class Net(nn.Module):
    """docstring for Net"""
    def __init__(self):
        #设置神经网络层数及每层大小
        super(Net, self).__init__()
        self.fc1 = nn.Linear(NUM_STATES, 256)
        self.fc1.weight.data.normal_(0, 0.1)
        self.out = nn.Linear(256, NUM_ACTIONS)
        self.out.weight.data.normal_(0, 0.1)

    #基于所给定状态,通过神经网络计算 Q 值进行动作选择
    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        action_prob = self.out(x)
        return action_prob
```

3.7.2 构建 DQN 智能体

```
#创建 DQN 智能体
class DQN():
    """docstring for DQN"""

    def __init__(self):
        #设置 DQN 智能体,包括创建神经网络、创建样本池等
        super(DQN, self).__init__()
        self.eval_net, self.target_net = Net(), Net()
        self.learn_step_counter = 0
        self.memory_counter = 0
        self.memory = np.zeros((MEMORY_CAPACITY, NUM_STATES * 2 + 2))
        self.optimizer = torch.optim.Adam(self.eval_net.parameters(), lr=LR)
        self.loss_func = nn.MSELoss()
        self.loss = None
        self.EPISILO = EPISILO
    #基于状态进行动作选择
    def choose_action(self, state):
        state = torch.unsqueeze(torch.FloatTensor(
state), 0)
        #选择 Q 值最优对应动作
        if np.random.randn() >= self.EPISILO:      #贪婪策略
            action_value = self.eval_net.forward(state)
            action = torch.max(action_value, 1)[1].data.numpy()
            action = action[0] if ENV_A_SHAPE == 0 else action.reshape(ENV_A_SHAPE)
        #选择随机动作
        else:#随机策略
            action = np.random.randint(0, NUM_ACTIONS)
            action = action if ENV_A_SHAPE == 0 else action.reshape(ENV_A_SHAPE)
        return action
```

```
    #将经验存储至样本池中
    def store_transition(self, state, action, reward, next_state):
        transition = np.hstack((state, [action, reward], next_state))
        index = self.memory_counter % MEMORY_CAPACITY
        self.memory[index, :] = transition
        self.memory_counter += 1

    #更新神经网络
    def learn(self, done):

        #参数更新
        #以 Q_NETWORK_ITERATION 为周期更新目标网络
        if self.learn_step_counter % Q_NETWORK_ITERATION == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())
        self.learn_step_counter += 1

        #从样本池中获取 BATCH_SIZE 数量的样本进行学习
        sample_index = np.random.choice(MEMORY_CAPACITY, BATCH_SIZE)
        batch_memory = self.memory[sample_index, :]
        batch_state = torch.FloatTensor(batch_memory[:, :NUM_STATES])
        batch_action = torch.LongTensor(batch_memory[:, NUM_STATES:NUM_STATES +
1].astype(int))
        batch_reward = torch.FloatTensor(batch_memory[:, NUM_STATES + 1:NUM_
STATES + 2])
        batch_next_state = torch.FloatTensor(batch_memory[:, -NUM_STATES:])

        #根据样本计算损失函数
        #q_eval
        q_eval = self.eval_net(batch_state).gather(1, batch_action)
        q_next = self.target_net(batch_next_state).detach()
        q_target = batch_reward + GAMMA * q_next.max(1)[0].view(BATCH_SIZE, 1)
        loss = self.loss_func(q_eval, q_target)
        #基于损失函数进行优化更新
        if done:
            self.loss = loss.item()      #新增
            #print("损失函数为")
            #print(self.loss)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

3.7.3　智能体环境构建

1. 导入依赖包,设置环境变量

```
import gym
import lizishengcheng
import xlrd
from workpiece import workpiece
from machine import machine


class Rms(gym.Env):
```

```
    def __init__(self):
        #状态空间

        self.action_space = lizishengcheng.Actions                    #动作空间
        self.n_action = len(self.action_space)                        #动作空间大小
        #self.n_feature = 12                                          #状态特征
        self.state = None

        self.readfile = xlrd.open_workbook(filename=r"E:\job\textbook\case0510(1)\case0510\
案例.xls")
        self.Job_kind = lizishengcheng.Job_kind   #工件种类,里面的元素是对应种类的数量
        #各个种类工件的总工序数量
        self.totalOpNum = sum(
            [(lizishengcheng.Op_num[i]) * (lizishengcheng.Job_kind[i]) for i in range(len
(lizishengcheng.Job_kind))])
        self.M_num = lizishengcheng.M_num                             #机器数
        self.M_kind = lizishengcheng.M_kind                           #机器种类
        self.C_num = lizishengcheng.C_num                             #构型集
        self.unlockOps = lizishengcheng.unlockOps                     #加工特征后解锁可加工特征表
        self.MT = lizishengcheng.Movetimes                            #运输时间
        self.TT = lizishengcheng.transfomations                       #重构时间
        self.Processing_time = lizishengcheng.Processing_time         #加工时间
        self.Jobs = []                                                #工件集
        self.Machines = []                                            #设备集
        self.Jobs_position = []                                       #工件所在设备位置
        self.opAvailable = []                                         #可加工的工序
        self.C_now = []                                               #设备当前构型
        self.Op_fin = 0                                               #已完成工序数量
        self.RActions = []                                            #已采取动作集合
        self.makespan = 0                                             #目标函数值
```

2. 初始化环境状态

```
#初始化
def reset(self):
    self.RActions = []

    #工件集初始化
    self.Jobs = []
    for i in range(len(self.Job_kind)):
        F = workpiece(i, self.Job_kind[i])                            #创建工件类
        self.Jobs.append(F)
    #机器集初始化
    self.Machines = []
    for i in range(self.M_num):
        F = machine(self.M_kind[i], self.C_num[i])                    #创建机器类
        self.Machines.append(F)
    #状态初始化
    self.Jobs_position = [self.M_num for j in range(len(self.Job_kind))]   #工件回到初始位置
    self.opAvailable = []                                             #工序可加工情况初始化
    opAvaSheet = self.readfile.sheet_by_name("工序初始可加工情况表")
    opAvaRow = opAvaSheet.nrows
    for i in range(len(self.Job_kind)):
```

```
            iOpAvailable = []
            for j in range(opAvaRow):
                if opAvaSheet.cell_value(j, 1) == i + 1:
                    iOpAvailable.append(int(opAvaSheet.cell_value(j, 5)))
            self.opAvailable.append(iOpAvailable)
        self.Op_fin = 0                                    #完成工序数初始化
        self.C_now = [0 for i in range(self.M_num)]        #机床当前构型初始化

        #构建状态
        self.state = []
        self.state.append(self.Op_fin)                     #state[0]0 完成工序数
        self.state.append(self.opAvailable)                #state[1]1 各个工序完成情况
        self.state.append(self.C_now)                      #state[2]2 机床当前构型情况表
        self.state.append(self.Jobs_position)              #state[3]3 工件所在机床位置表
        return self.state
```

3. 动作转移策略

```
    #基于动作的状态转移
def step(self, action: int):
    #获取动作
    action = self.action_space[action]
    Job, Operation, Machine, config = action[0], action[1], action[2], action[3] #根据动作获取
#工件种类、工序、设备、构型

    #设置奖励相关参数、惩罚值、虚拟利润值
    penalty_value = -1
    #positive_value = 1
    profit = 8000
    #默认处于未完成状态
    done = 0

    #self.state[1][Job][Operation] == 1 表示所选择的工件的加工特征可加工
    if self.state[1][Job][Operation] == 1:#选择的加工特征,在当前状态下可加工(前工序已加
#工或无前工序约束)

        #记录设备第一次加工时的构型作为初始构型
        if not self.Machines[Machine].End:
            self.state[2][Machine] = config                #机器重构为所选择的构型
        #计算执行该动作前的已安排工序的最大完工时间
        mp_i = 0
        EndTime = []
        for i in range(len(self.Jobs)):
            if self.Jobs[i].End:
                EndTime.append(max(self.Jobs[i].End))
        if EndTime:
            mp_i = max(EndTime)
        #获取当前动作对应所需的工件运输时间及设备重构时间
        mt = self.MT[self.state[3][Job]][Machine]
        tt = self.TT[Machine][self.state[2][Machine]][config]

        try:
            last_ot = max(self.Jobs[Job].End)      #计算选择工件到后续加工完的时间
        except:
```

```
            last_ot = 0
        try:
            last_mt = max(self.Machines[Machine].End) #计算选择机器的上一个工序的完工时间
        except:
            last_mt = 0

        #加工开始时间计算
        if last_ot >= last_mt:                          #机器先完工,从工件完成加工时开始执行动作
            if mt >= tt:                                #重构先完成,则移动到达后开始加工
                Start_time = last_ot + mt
            else:                                       #移动先完成,则重构完成后开始加工
                if last_ot + mt >= last_mt + tt:
                    Start_time = last_ot + mt
                else:
                    Start_time = last_ot + tt
        else:  #工件先完工,仍然从工件完成加工时开始执行动作
            if (last_ot + mt) > (last_mt + tt):     #重构先完成,则移动到达后开始加工
                Start_time = last_ot + mt
            else:                                       #移动先完成,则重构完成后开始加工
                Start_time = last_mt + tt
        PT = (self.Processing_time[Job][Operation][Machine][config]) * (self.Job_kind[Job])
  #工序加工时间获取
        end_time = Start_time + PT                      #结束时间计算: 开始时间+加工时间
        self.Machines[Machine]._add(Start_time, end_time, Job,
                                    PT)         #给机器添加开始时间、结束时间,对应加工
#的工件还有加工时间(重构的状态)
        self.Jobs[Job]._add(Start_time, end_time, Machine, PT)#给工件添加开始时间、结束时
#间,对应机器还有加工时间

        self.state[1][Job][Operation] = 2               #设置所选择工序状态为已加工
        if (len(self.unlockOps[Job][Operation]) != 1) or (self.unlockOps[Job][Operation][0]
!= -1):
            #判断该工序完成加工后的可加工工序情况
            for i in range(len(self.unlockOps[Job][Operation])):
                if self.state[1][Job][self.unlockOps[Job][Operation][i]] != 2:#如果哪一道工
#序没有被加工
                    self.state[1][Job][self.unlockOps[Job][Operation][i]] = 1   #解锁该道
#工序的加工权限
        self.state[3][Job] = Machine                #工件移动到所加工的机床
        self.state[2][Machine] = config             #机器重构为所选择的构型
        self.state[0] += self.Job_kind[Job]         #更新已经加工完工序数量
        #计算执行该动作后的已安排工序的最大完工时间
        EndTime = []
        for i in range(len(self.Jobs)):
            if self.Jobs[i].End:
                EndTime.append(max(self.Jobs[i].End))
        mp_j = max(EndTime)
        #计算执行正确动作对应的奖励值
        reward = profit + (mp_i - mp_j)
        #记录该动作对应的数据信息
        RAction = [action, PT, Start_time, mt, tt]
        self.RActions.append(RAction)
        #判断是否所有加工特征均已经完成加工
        if self.state[0] == self.totalOpNum:
```

```
            #获取最终目标函数值,并将完成判断因子 done 的数值改为 1,表示完成
            self.makespan = mp_j
            done = 1

    else:
        #选择动作错误不可执行,例如,选择的加工特征不可加工,或者已经完成加工,选错了,给
#一个惩罚
        reward = penalty_value

    return self.state, reward, done, self.RActions
```

3.7.4　智能体主函数构建

1. 导入依赖包

```
import Rms_env
from DQN import DQN
import matplotlib.pyplot as plt
import cost
import DataToExcel as Dte
import numpy as np

#样本容量、Q 网络迭代更新超参数设置
MEMORY_CAPACITY = 2000
Q_NETWORK_ITERATION = 100
#创建环境实例
env = Rms_env.Rms()
```

2. 循环迭代

```
def main():

    #记录迭代数据
    row = 0
    #想写入哪个表格后面就跟哪个表格
    excel_name = 'excel/' + 'excel1.xls'
    #sheet 名称
    sheet_name = '迭代情况'                          #自适应的是整个制造系统
    #表头
    title = ['迭代次数', '奖励', '损失函数', "实际工时"]
    #新建表格
    Dte.excel_int(excel_name, sheet_name)
    #写入表头
    Dte.excel_write_title(excel_name, title)

    #创建 DQN 智能体
    dqn = DQN()
    #设置迭代次数、初始最大奖励
    episodes = 500
    max_reward = 3000
    print("Collecting Experience....")
    reward_list = []
```

```
bestActions = []

#迭代优化
for i in range(episodes):
    #状态初始化、累计奖励初始化
    state = env.reset()
    ep_reward = 0
    policy = []
    each_step_value = []
    env.configuration = []

    while True:
        #选择动作
        state_D = cost.state_AD_change(state)
        action = dqn.choose_action(state_D)
        policy.append(action)

        #基于所选择的动作获取奖励、新状态等
        next_state, reward, done, Actions = env.step(action)

        next_state_D = cost.state_AD_change(next_state)

        #存储状态转移的经验值样本库
        dqn.store_transition(state_D, action, reward, next_state_D)
        #记录状态转移对应获取的奖励
        ep_reward += reward
        each_step_value.append(reward)

        #在样本数量超过样本容量时更新智能体
        if dqn.memory_counter >= MEMORY_CAPACITY:
            dqn.learn(done)
            if done:
                #记录数据
                Dte.excel_write_array(excel_name, str(row), 0, row)
                Dte.excel_write_array(excel_name, str(ep_reward), 1, row)
                Dte.excel_write_array(excel_name, str(dqn.loss), 2, row)
                Dte.excel_write_array(excel_name, str(env.makespan), 3, row)
                row += 1
                break
        if done:
            #记录数据
            Dte.excel_write_array(excel_name, str(row), 0, row)
            Dte.excel_write_array(excel_name, str(ep_reward), 1, row)
            Dte.excel_write_array(excel_name, str(dqn.loss), 2, row)
            Dte.excel_write_array(excel_name, str(env.makespan), 3, row)
            row += 1
            break
        #状态转移至新的状态
        state = next_state
    #记录目标函数数据
    reward_list.append(env.makespan)

    #累积奖励大于最大奖励时更新记录的最大累积奖励
    if ep_reward > max_reward:
```

```
                bestActions = Actions
                max_reward = ep_reward
                print("最大回报为: ", max_reward)
                print("最大时间为" + str(env.makespan))
            dqn.EPISILO = max(0.01, dqn.EPISILO * 0.995)

    print(bestActions)
    plt.plot(np.arange(len(reward_list)), reward_list)
    plt.ylabel('Cost')
    plt.xlabel('training steps')
    plt.show(block=True)

if __name__ == '__main__':
    main()
```

3.7.5　其他相关函数

1. 工件相关函数

```
#工件类
class workpiece:
    def __init__(self, Kind, num):
        self.Kind = Kind                #零件的种类
        self.Num = num                  #零件的数量
        self.Start = []                 #工件加工特征加工的开始时间
        self.End = []                   #工件加工特征加工的结束时间
        self.T = []                     #工件加工特征加工的加工时间
        self.assign_for = []            #零件的加工所分配的机床

    #在安排对应机床后添加对应的加工信息
    def _add(self, S, E, obs, t):
        self.Start.append(S)
        self.End.append(E)
        self.T.append(t)
        self.assign_for.append(obs)
```

2. 格式转换函数

```
import numpy
#状态格式转换,方便智能体识别状态
def state_AD_change(state):
    Op_fin = [state[0]]
    opAvailable = []
    for i in range(len(state[1])):
        opAvailable += state[1][i]
    C_now = state[2]
    Jobs_position = state[3]
    state_D = Op_fin + opAvailable + C_now + Jobs_position
    state_D = numpy.array(state_D)
    return state_D
```

3. 设备相关函数

```
#设备类
```

```
class machine:
    def __init__(self, Kind, Config):
        self.Kind = Kind                #机器的种类
        self.Config = Config            #机器的构型
        self.Start = []                 #进行加工的各个任务的开始时间
        self.End = []                   #进行加工的各个任务的结束时间
        self.T = []                     #进行加工的各个任务的加工时间
        self.assign_for = []            #机床所进行加工对应的工件

    #在完成安排后添加对应的任务参数
    def _add(self, S, E, obs, t):
        self.Start.append(S)
        self.End.append(E)
        self.T.append(t)
        self.assign_for.append(obs)
```

4. 案例表格相关函数

```
#coding=UTF-8
import xlrd
import xlwt
from xlutils.copy import copy

#新建表格
def excel_int(path, sheet_name):
    workbook = xlwt.Workbook()                          #新建一个工作簿
    workbook.add_sheet(sheet_name)                      #在工作簿中新建一个表格
    workbook.save(path)                                 #保存工作簿
    print("新建表格成功,表格名称为:", path)

#写入表头
def excel_write_title(path, titels):
    workbook = xlrd.open_workbook(path)                 #打开工作簿
    new_workbook = copy(workbook)                       #将 xlrd 对象复制转换为 xlwt 对象
    new_worksheet = new_workbook.get_sheet(0)           #获取转换后工作簿中的第一个表格
    for j in range(0, len(titels)):
        new_worksheet.write(0, j, str(titels[j]))       #在表格中写入数据(对应的行)
    new_workbook.save(path)                             #保存工作簿

#向表格按列写入一维数组(列表)
def excel_write_array(path, value, column, row):
    workbook = xlrd.open_workbook(path)                 #打开工作簿
    new_workbook = copy(workbook)                       #将 xlrd 对象复制转换为 xlwt 对象
    new_worksheet = new_workbook.get_sheet(0)           #获取转换后工作簿中的第一个表格
    new_worksheet.write(row+1, column, value)
    new_workbook.save(path)                             #保存工作簿
```

5. 表格读取函数

```
import xlrd

#根据 Excel 表格生成案例
def Generatorexample():
    '''
    Processing time
    M_num: Machine Number,
    Op_num: Operation Number,
```

```
    J_num:Job NUMBER
    '''
    readfile = xlrd.open_workbook(filename=r"E:\job\textbook\case0510(1)\case0510\案例.
xls")

    jobSheet = readfile.sheet_by_name("工件表")
    jobRow = jobSheet.nrows
    Job_kind = []   #初始工件种类,序号代表种类,里面的元素是对应种类的数量
    Job_num = int(jobSheet.cell_value(0, 1))          #初始工件数量
    Op_num = []                                       #为每种工件生成工序数量
    for i in range(jobRow - 2):   #生成一张表记录每种零件的类型和工序数量
        Job_kind.append(int(jobSheet.cell_value(i + 2, 1)))
        Op_num.append(int(jobSheet.cell_value(i + 2, 2)))

    #产生机床,[[,],[],[],[],[]]第一层是分机床的不同类型,第二层是分机床的构型种类和数量
    machSheet = readfile.sheet_by_name("机床表")
    machRow = machSheet.nrows
    M_kind = []                                       #初始机床种类
    M_num = int(machSheet.cell_value(0, 1))           #初始机床总数量
    C_num = []                                        #每台机床构型数量
    C_now = []                                        #机床当前状态表
    for i in range(machRow - 2):   #生成一张表记录每个机床的类型和构型数量
        M_kind.append(int(machSheet.cell_value(i + 2, 1)))
        C_num.append(int(machSheet.cell_value(i + 2, 2)))
        C_now.append(int(machSheet.cell_value(i + 2, 3)))
    M_cnum = sum(C_num)                               #所有机床总构型数
    #记录加工时间 [工件种类[工序[机床种类[机床构型(加工时间),机床构型(不能加工就是-1)],机
#床种类,,,,,],工序,,,,],工件种类,,,,,]
    Processing_time = []
    C_fin = []
    BActions = []

    proSheet = readfile.sheet_by_name("加工时间表")
    for i in range(len(Job_kind)):                    #遍历每种工件
        Job_i = []
        C_ifin = []
        for j in range(Op_num[i]):                    #遍历当前种类工件的工序
            Op_i = []
            C_ifin.append(0)                          #没有完成的工艺记为 0
            for k in range(M_num):                    #遍历所有机器
                Actions_i_j_k = [(i, j, k, l) for l in range(C_num[k])]   #动作空间
                if (k == 0):
                    jichuangshu = 0
                else:
                    jichuangshu = sum(C_num[0:k])
                M_i = [int(proSheet.cell_value(((j * M_cnum) + jichuangshu + (l + 1) +
2), (((i + 1) * 2) + 1))) for l
                       in range(C_num[k])]            #为所有构型添加一个-1
                BActions.extend(Actions_i_j_k)
                Op_i.append(M_i)                      #将各个机床构型加工时间添加至工序表中
            Job_i.append(Op_i)                        #将工序情况添加至工件表中
        C_fin.append(C_ifin)
        Processing_time.append(Job_i)                 #将工件情况添加至总表中
    O_num = 0
    for i in range(len(Job_kind)):                    #得到总工序数量
        O_num += Job_kind[i] * Op_num[i]
    #生成可执行动作的动作空间
    Actions = []
    for i in range(len(BActions)):
```

```
            if Processing_time[BActions[i][0]][BActions[i][1]][BActions[i][2]][BActions[i][3]]
!= -1:
                Actions.append(BActions[i])
        StrActions = []
        for i in range(len(Actions)):
            StrActions.append(str(Actions[i]))

        Movetimes = []                                  #机床间工件移动的时间
        moveTimeSheet = readfile.sheet_by_name("机床间距表")
        for i in range(M_num + 1):
            Move_Fmi = []
            for j in range(M_num):
                Move_Fmi.append(int(moveTimeSheet.cell_value(i + 2, j + 2)))
            Movetimes.append(Move_Fmi)
        transfomations = []                             #构型变化表
        transfomationSheet = readfile.sheet_by_name("构型变化表")
        for i in range(M_num):
            TF_Mi = []
            for j in range(C_num[i]):
                TF_Mi_jth = []
                for k in range(C_num[i]):
                    TF_Mi_jth.append(int(transfomationSheet.cell_value(j + 1, (sum(C_num[0:i])
+ i + k + 1))))
                TF_Mi.append(TF_Mi_jth)
            transfomations.append(TF_Mi)

        opAvailables = []                               #工序可加工情况表
        unlockOps = []                                  #加工后解锁后续加工工序表
        opAvaSheet = readfile.sheet_by_name("工序初始可加工情况表")
        opAvaRow = opAvaSheet.nrows
        for i in range(len(Job_kind)):
            iOpAvailables = []
            iunlockOps = []
            for j in range(opAvaRow):
                if opAvaSheet.cell_value(j, 1) == i + 1:
                    iOpAvailables.append(int(opAvaSheet.cell_value(j, 5)))
                    ijunlockOps = []
                    for k in range(int(opAvaSheet.cell_value(j, 6))):
                        ijunlockOps.append(int(opAvaSheet.cell_value(j, 7 + k)))
                    iunlockOps.append(ijunlockOps)
            opAvailables.append(iOpAvailables)
            unlockOps.append(iunlockOps)

        return M_kind, Job_kind, Processing_time, O_num, Op_num, M_num, Job_num, C_num, C_
fin, C_now, Actions, StrActions, Movetimes, transfomations, unlockOps, opAvailables
        #机器种类  加工时间表  总工序数量  工序编号和对应工序数量映射  工序编号和对应工
#序数量映射

#这里可以用调用表格代替
M_kind, Job_kind, Processing_time, O_num, Op_num, M_num, Job_num, C_num, C_fin, C_now,
Actions, StrActions, Movetimes, transfomations, unlockOps, opAvailables = Generatorexample()
print(Processing_time)
```

3.7.6　实例验证

输入数据格式如表 3-8～表 3-11 所示。

表 3-8　生产订单信息

产 品 类 型	产 品 数 量	特 征 数 量
1A	200	3
2B	500	3
3C	100	4
4D	300	2

表 3-9　可重构机床信息

可重构机床	构　型　数	转 换 时 间	转 换 成 本
1A	3	800	350
2B	5	600	300
3C	3	1000	500

表 3-10　产品特征对应机床构型表

产 品 特 征	可重构机床 1	可重构机床 2	可重构机床 3
$F(1,1)$	1	—	2
$F(1,2)$	—	4	1
$F(1,3)$	2	1	3
$F(2,1)$	2	3	—
$F(2,2)$	3	—	1
$F(2,3)$	1	1	—
$F(3,1)$	—	5	2
$F(3,2)$	2	—	1
$F(3,3)$	1	4	3
$F(3,4)$	3	3	—
$F(4,1)$	1	4	3
$F(4,2)$	3	2	2

表 3-11　产品特征加工时间表

产 品 特 征	可重构机床 1	可重构机床 2	可重构机床 3
$F(1,1)$	5	—	8
$F(1,2)$	—	10	12
$F(1,3)$	6	7	9
$F(2,1)$	13	19	—
$F(2,2)$	10	—	5
$F(2,3)$	8	12	—
$F(3,1)$	—	4	9
$F(3,2)$	6	—	1
$F(3,3)$	3	7	10
$F(3,4)$	9	3	—

续表

产 品 特 征	可重构机床 1	可重构机床 2	可重构机床 3
$F(4,1)$	5	9	4
$F(4,2)$	8	3	6

实验结果展示如图 3-7 和图 3-8 所示，其搜索到的最优 Makespan＝13000s，运行时间为 0.96546s。

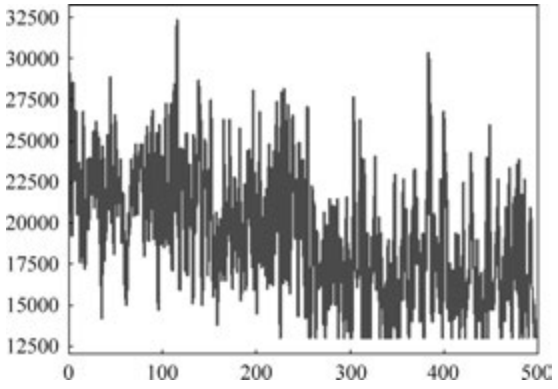

图 3-7 DQN 算法迭代优化图

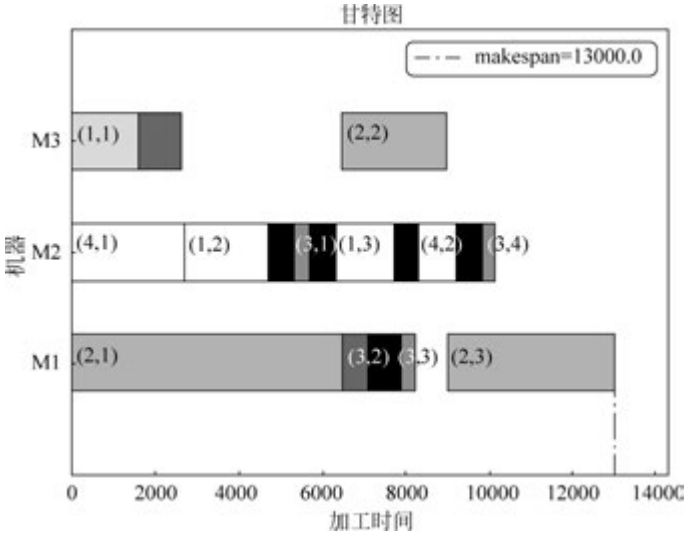

图 3-8 最优个体方案的甘特图

小结

强化学习的出现与心理学中的动物学习和最优控制优化理论两个领域密切相关。强化学习是目前优化领域的研究热点之一，其最终目的是在交互过程中获取最优策略。强化学习用于描述和解决智能体在与环境的交互过程中通过学习策略以达成回报最大化或实现特定目标的问题。本章首先介绍了强化学习算法的基础理论，包括状态-动作-奖励-策略以及

马尔可夫决策过程等。然后，介绍了 Q-Learning 算法的主要执行流程，包括初始化 Q 值函数、选择动作、执行动作并观察奖励和下一个状态、更新 Q 值函数、收敛和策略提取、终止条件判定等步骤。并对强化学习中包含的名词概念进行解释，通过引入一个具体数值案例对 Q-Learning 算法进行举例讲解。随后，给出了 Q-Learning 算法的两种改进思路，包括学习率调度改进和引入神经网络来估计 Q 值形成深度强化学习(DRL)。最后，基于一个可重构制造系统调度工程问题，详细解析了 DQN 算法的应用过程，并采用 Python 语言进行编程实现 DQN 算法的整个流程完成实例验证。


在线测试

习题

一、填空题

1. 一个标准的神经网络包括_____、_____和_____。

2. Q-Learning 是一种_____的强化学习算法，是一种基于_____的方法，其核心思想是通过迭代更新一个_____。

3. 学习率 α 定义了新获得的信息覆盖旧信息的程度。在_____时，智能体不会学习任何新的信息；在_____时，智能体只考虑最新的信息；在_____时，智能体既考虑过去的知识也考虑新的奖励。

4. _____决定了智能体对未来奖励的折现，一般来说，其初始值应该_____，以便智能体能够广泛地探索环境；随着学习的进行，其值应该逐渐_____，以便智能体能够收敛到最优的策略。若问题的目标需要长期的探索才能取得，则其取值应该_____。

5. DQN 算法对 Q-Learning 算法的改进过程中，除了引入_____来估计 Q 值外，另一个主要贡献之一是引入了_____机制，在这种机制下，智能体会将每个时间步的经验存储在一个回放缓冲区中。

二、判断题

1. 目标网络是一个固定的网络，用于计算目标值，通过增大目标值的变化来提高训练的稳定性。(　　)

2. ε-greedy 是一种常用的策略来平衡探索与利用。(　　)

3. 在模型无关的强化学习算法中，算法并不显式地对 P 和 R 进行建模，而是将其隐式地表达在其他模型构件中，通过对这些模型构件进行求解以找到最优策略。(　　)

4. 无模型的强化学习方法通过在环境中尝试搜集转移概率和奖励函数的观测值，对问题进行建模，然后使用传统的 MDP 动态规划算法求解最优策略。(　　)

5. 如果策略评估的数据是完全使用当前的策略来进行采样的，那么策略很容易陷入局部解。(　　)

三、应用题

1. 简述 Q-Learning 算法的步骤流程(7 大步骤)，并画出对应流程框图。

2. 简述在设计奖励函数时应该遵循的几个原则。