



北京交通大学

BEIJING JIAOTONG UNIVERSITY

“浅谈最优化” 课程设计 研究报告

UNDERGRADUATE CLASS DESIGN PAPER
RESEARCH REPORT

题 目: 人工智能中的最优化
姓名学号: 苏学睿 19211278
指导老师: 罗自炎 教授、修乃华 教授
学 院: 数学与统计学院
专业班级: 知行 1901

2022 年 6 月

人工智能中的最优化

摘要

最优化理论与统计分析、模拟仿真并称为数学建模的三大基本方法和技术。在应用数学知识解决实际问题时，最优化理论是非常活跃的研究发现和分支。从上世纪后半叶开始迅速发展至今。已具备非常完善的知识体系。针对不同的问题，新的理论和方法不断被提出。

作者注意到最优化理论自诞生以来，在各个领域的应用非常广泛，尤其在人工智能领域，在控制、预测、搜索、估计等任务中发挥了至关重要的作用。本文基于本学期学到的知识，站在优化的角度上，梳理了人工智能领域控制、预测、搜索三大任务情景下最优化算法的设计与使用。可以说，最优化的问题形式在这三类任务情景中均发挥了或是作为算法重要部分或是引导出更简单近似方法的重要作用。

首先，作者对最优化的知识进行了系统地梳理。解释了相关的基本概念。对一阶、二阶、带约束、无约束最优性条件进行了阐述。列举了学过的几类算法的算法框架和理论性质。然后阐述了控制、预测、搜索三类任务情景下最优化算法的设计与使用。

第一，当前在控制领域-强化学习的研究中，基本目标是在马氏决策过程的框架内，基于**贝尔曼最优方程**，通过**动态规划**、**时序差分**和**策略梯度**等优化算法求解最优值函数和最优策略。由于策略函数和值函数均为复杂的抽象函数，不可直接求解，所以强化学习的目标就是寻找合适的方式不断优化策略和价值函数使其接近最优策略和“真实”的价值函数。**所以说，强化学习的算法可以看作是在特定任务情景下的优化算法。**本文主要介绍了如何使用贝尔曼最优方程求解最优值函数和最优策略。同时对强化学习这门学科诞生三十年来（1990-2018）产生的几大算法（Q-learning、DQN、AC、TRPO、PPO 等）进行了简要介绍，指出算法中“优化”思想的体现，以此来说明优化在该领域中的重要性。

第二，（此部分为作者原创内容，若有错误之处请老师指正）本文介绍了一类对直播间行为进行预测的 HMM 模型。但随着参数和状态空间的增大，通过极大似然对模型参数进行估计的方法是行不通的。于是作者在该框架下将问题转化为优化问题，使用非线性共轭梯度法进行求解，在实际数据上取得了较好的成果。

第三，在各种建模比赛中，启发式算法使用频率很高。这些算法不同于传统的优化算法，由于其本身的智能性，该算法被广泛应用在搜索任务情景中。本文对几种启发式优化算法进行了理论介绍，并将其应用在**旅行商问题（TSP）**的求解中。传统搜索算法对于该问题大多是病态的，但是本文介绍的四种启发式算法却可以高效的求解该问题。

关键词：最优化理论 最优化算法 人工智能 凸优化 启发式算法

Optimization Theory In AI

ABSTRACT

Optimization theory, statistical analysis and simulation are the three basic methods and techniques of mathematical modeling. When applying mathematical knowledge to solve practical problems, optimization theory is a very active research discovery and branch. It has developed rapidly since the second half of the last century. It has a very perfect knowledge system. For different problems, new theories and methods have been put forward.

The author notes that since the birth of optimization theory, it has been widely used in various fields, especially in the field of artificial intelligence, and has played a vital role in control, prediction, search, estimation and other tasks. Based on the knowledge learned this semester, from the perspective of optimization, this paper combs the design and use of optimization algorithms in the three task scenarios of control, prediction and search in the field of artificial intelligence. It is not too much to say that the optimization problem form plays a role in these three types of task scenarios, either as an important part of the algorithm or as a guide to simpler approximation methods.

I systematically combs the knowledge of optimization. The related basic concepts are explained. The first-order, second-order, constrained and unconstrained optimality conditions are described. The algorithm framework and theoretical properties of several learned algorithms are listed. Then it describes the design and use of optimization algorithm under three kinds of task scenarios: control, prediction and search. The prediction algorithm based on HMM model is the original content of the author (preparing to publish papers). If there is any deficiency, please correct it.

Firstly, in the current research on reinforcement learning in the control field, the basic goal is to solve the optimal value function and optimal strategy through **dynamic programming, time difference** and **policy gradient** within the framework of Markov decision process, based on **Bellman optimal equation**. Since both the strategy function and the value function are complex nonconvex functions, which cannot be solved directly, the framework of reinforcement learning itself is put aside. No matter what approximate method is adopted in theoretical analysis to solve the above nonconvex functions, it needs to be solved with the help of optimization algorithm when it is implemented. This paper mainly introduces how to use the bellman optimal equation to solve the optimal value function and optimal strategy, and transform them into the form of optimization problems. At the same time, it

briefly introduces several major algorithms (Q-Learning, DQN, AC, TRPO, PPO, etc.) that have been produced since the birth of reinforcement learning to 2018, and also points out the embodiment of optimization problems in algorithms, so as to illustrate the importance of optimization in this field.

Secondly, this paper introduces a kind of HMM model to predict the behavior of live broadcast room. However, with the increase of parameters and state space, the method of estimating model parameters by maximum likelihood is not feasible. So the author transforms the problem into an optimization problem under this framework, and uses the nonlinear conjugate gradient method to solve it, and has achieved good results on the actual data.

Thirdly, heuristic algorithms are frequently used in various modeling competitions. These algorithms are different from the traditional optimization algorithms. Because of their intelligence, this algorithm is widely used in search task scenarios. In this paper, several heuristic optimization algorithms are introduced theoretically and applied to the **traveling salesman problem (TSP)**. Traditional search algorithms are ill conditioned for this problem, but the four heuristic algorithms introduced in this paper can solve this problem efficiently.

Key words: optimization theory Optimization algorithm artificial intelligence Convex optimization heuristic algorithm

目录

第 1 章	最优化理论的发展历程	1
第 2 章	最优化基本概念	2
2.1	最优化问题	2
2.1.1	最优化问题定义	2
2.1.2	最优化问题分类	2
2.2	凸集	3
2.3	凸函数	3
第 3 章	最优化基本理论概述	4
3.1	一阶最优性条件	4
3.1.1	无约束优化	4
3.1.2	带约束优化 (KKT 条件)	4
3.2	二阶最优性条件	6
3.2.1	无约束优化	6
3.2.2	带约束优化	6
第 4 章	最优化部分算法分析	8
4.1	模式搜索法	8
4.2	线搜索方法	8
4.3	几类无约束优化算法框架	9
4.3.1	最速下降法	9
4.3.2	牛顿法	9
4.3.3	非线性共轭梯度法	9
4.3.4	拟牛顿法	10
4.3.5	Gauss-Newton 算法	10
4.3.6	信赖域方法	10
4.3.7	LMF 算法	10
4.4	几类带约束优化算法框架	10
4.4.1	积极集方法算法	11
4.4.2	投影梯度算法	11
4.4.3	乘子罚函数算法	11
第 5 章	强化学习与最优化	12
5.1	有限维马氏决策过程	13
5.1.1	值函数与动作值函数	14
5.1.2	最优策略和最优值函数	14
5.2	动态规划求解最优策略和最优值函数	15

5.2.1	策略评价	15
5.2.2	策略改进	16
5.2.3	策略迭代	17
5.3	Sarsa 算法	18
5.4	Q-learning 算法	19
5.5	DQN 算法	20
5.5.1	DQN 算法原理	21
5.6	Actor-Critic 算法	21
5.6.1	策略梯度理论	22
5.6.2	Actor-Critic 算法原理与优化方法	22
5.7	TRPO 算法	23
5.7.1	TRPO 算法原理	23
5.8	PPO 算法	25
5.9	小结	26
第 6 章	基于 HMM 模型的非线性共轭梯度法	27
6.1	问题背景	27
6.2	HMM 模型	27
6.3	模型建立	28
6.4	问题求解	31
第 7 章	旅行商问题求解	32
7.1	旅行商问题	32
7.2	启发式算法解决旅行商问题 (TSP)	33
7.2.1	启发式算法	33
7.2.2	实例求解	36
	参考文献	38
附录 A	附录代码	40
A.1	SARSA 算法 <i>pytorch</i> 代码	40
A.2	Q-learning 算法 <i>pytorch</i> 代码	43
A.3	DQN 算法 <i>pytorch</i> 代码	45
A.4	Actor-Critic 算法 <i>pytorch</i> 代码	50
A.5	TRPO 算法 <i>pytorch</i> 代码	54
A.6	PPO 算法 <i>pytorch</i> 代码	61
A.7	共轭梯度法解 HMM 模型 <i>python</i> 代码	66
A.8	模拟退火 (SA) 算法 <i>python</i> 代码	76
A.9	遗传 (GA) 算法 <i>python</i> 代码	79
A.10	蚁群 (ACO) 算法 <i>python</i> 代码	83
A.11	禁忌搜索 (TS) 算法 <i>python</i> 代码	87

第 1 章 最优化理论的发展历程

^[1] 最优化理论与算法是一个重要的数学分支，它所研究的问题是讨论在众多的方案中什么样的方案最优以及怎样找出最优方案。这类问题普遍存在。例如，工程设计中怎样选择设计参数，使得设计方案既满足设计要求又能降低成本，资源分配中，怎样分配有限资源，使得分配方案既能满足各方面的基本要求，又能获得好的经济效益；生产计划安排中，选择怎样的计划方案才能提高产值和利润；原料配比问题中，怎样确定各种成分的比例，才能提高质量，降低成本；城建规划中，怎样安排工厂、机关、学校、商店、医院、住户和其他单位的合理布局，才能方便群众，有利于城市各行各业的发展；农田规划中，怎样安排各种农作物的合理布局，才能保持高产稳产，发挥地区优势；军事指挥中，怎样确定最佳作战方案，才能有效地消灭敌人，保存自己；在人类活动的各个领域，诸如此类，不胜枚举。最优化这一数学分支，正是为这些问题的解决提供理论基础和求解方法，它是一门应用广泛、实用性强的学科。

最优化是个古老的课题。长期以来，人们对最优化问题进行着探讨和研究。早在 17 世纪，英国科学家 Newton 发明微积分的时代，就已提出极值问题，后来又出现 Lagrange 乘数法。1847 年法国数学家 Cauchy 研究了函数值沿什么方向下降最快的问题，提出最速下降法。1939 年前苏联数学家坎托罗维奇提出了解决下料问题和运输问题这两种线性规划问题的求解方法。人们关于最优化问题的研究工作，随着历史的发展不断深入。但是，任何科学的进步，都受到历史条件的限制，直至 20 世纪 30 年代，最优化这个古老课题并未形成独立的有系统的学科。

20 世纪 40 年代以来，由于生产和科学研究突飞猛进地发展，特别是电子计算机广泛的应用，使最优化问题的研究不仅成为一种迫切需要，而且有了求解的有力工具。因此最优化理论和算法飞速发展起来，形成一个新的学科。至今已出现线性规划、整数规划、非线性规划、几何规划、动态规划、随机规划、网络流等许多分支。最优化理论和算法在实际应用中正在发挥越来越大的作用。

最优化理论自诞生以来，在各个领域的应用非常广泛，尤其在人工智能领域，在控制、预测、搜索、估计等任务中发挥了至关重要的作用。本文基于本学期学到的知识，站在优化的角度上，梳理了人工智能领域控制、预测、搜索三大任务情景下最优化算法的设计与使用。可以说，最优化的问题形式在这三类任务情景中均发挥了或是作为算法重要部分或是引导出更简单近似方法的作用。

第2章 最优化基本概念

2.1 最优化问题

2.1.1 最优化问题定义

[2] 数学优化问题或者说优化问题可以写成如下形式：

$$\begin{aligned} & \text{minimize} && f_0(x) \\ & \text{subject to} && f_i(x) \leq b_i, \quad i = 1, \dots, m \end{aligned}$$

这里，向量 $x = (x_1, \dots, x_n)$ 称为问题的**优化变量**，函数 $f_0 : \mathbf{R}^n \rightarrow \mathbf{R}$ 称为**目标函数**，函数 $f_i : \mathbf{R}^n \rightarrow \mathbf{R}, i = 1, \dots, m$ ，被称为（不等式）**约束函数**，常数 b_1, \dots, b_m 称为**约束上限**或者**约束边界**。如果在所有满足约束的向量中向量 x^* 对应的目标函数值最小：即对于任意满足约束 $f_1(z) \leq b_1, \dots, f_m(z) \leq b_m$ 的向量 z ，有 $f_0(z) \geq f_0(x^*)$ ，那么称 x^* 为上述优化问题的**最优解**或者**解**， $f_0(x^*)$ 为**最优值**。

2.1.2 最优化问题分类

使用不同的分类指标可以将最优化问题分为不同的类别，如表2.1所示：

表 2.1 最优化问题分类方法

划分标准	问题分类	
有无约束	无约束问题	
	等式约束问题	
	不等式约束问题	
目标函数和约束函数的线性程度	线性规划问题	
	二次规划问题	
	非线性规划问题	
目标函数和可行域的凸性	凸优化问题	
	非凸优化问题	
目标函数和约束函数的解析性质	光滑优化问题	
	非光滑优化问题	
可行域中可行点的个数	连续优化问题	
	离散优化问题	0-1 规划问题
		整数规划问题
		0-1 混合规划问题
		混合整数规划问题
模型参数的确定性	确定型规划问题	
	随机规划问题	

虽然以上优化问题均在相应领域有很广泛地应用，但作为初学者，我们主要学习目标函数和约束函数均连续可微的确定型规划问题，即非线性最优化问题。

2.2 凸集

在最优化问题的理论分析中, 常用到凸集和凸函数的概念, 所以下面我们简单地给出相关的定义。

集合 C 被称为凸集, 如果 C 中任意两点间的线段仍然在 C 中, 即对于任意 $x_1, x_2 \in C$ 和满足 $0 \leq \theta \leq 1$ 的 θ 都有

$$\theta x_1 + (1 - \theta)x_2 \in C.$$

粗略地, 如果集合中的每一点都可以被其他点沿着它们之间一条无阻碍的路径看见, 那么这个集合就是凸集。所谓无阻碍, 是指整条路径都在集合中。由于仿射集包含穿过集合中任意不同两点的整条直线, 任意不同两点间的线段自然也在集合中。因而仿射集是凸集。图 2.2 显示了 \mathbf{R}^2 空间中一些简单的凸和非凸集合。

我们称点 $\theta_1 x_1 + \cdots + \theta_k x_k$ 为点 x_1, \cdots, x_k 的一个凸组合, 其中 $\theta_1 + \cdots + \theta_k = 1$ 并且 $\theta_i \geq 0, i = 1, \cdots, k$ 。与仿射集合类似, 一个集合是凸集等价于集合包含其中所有点的凸组合。点的凸组合可以看做它们的混合或加权平均, θ_i 代表混合时 x_i 所占的份数。

我们称集合 C 中所有点的凸组合的集合为其凸包, 记为 $\text{conv } C$:

$$\text{conv } C = \{\theta_1 x_1 + \cdots + \theta_k x_k \mid x_i \in C, \theta_i \geq 0, i = 1, \cdots, k, \theta_1 + \cdots + \theta_k = 1\}.$$

顾名思义, 凸包 $\text{conv } C$ 总是凸的。它是包含 C 的最小的凸集。也就是说, 如果 B 是包含 C 的凸集, 那么 $\text{conv } C \subseteq B$ 。

与凸集相关的概念还有仿射集合, 凸锥, 半正定锥 (S_+^n , 在微分几何中, 三维欧氏空间下, S^2 对应二维流形), 正定锥 (S_{++}^n), 超平面, Euclid 球, 范数球, 范数锥, 多面体, 单纯形等集合的定义。随着学习的深入将再进一步的补充。

2.3 凸函数

函数 $f: \mathbf{R}^n \rightarrow \mathbf{R}$ 是凸的, 如果 $\text{dom } f$ 是凸集, 且对于任意 $x, y \in \text{dom } f$ 和任意 $0 \leq \theta \leq 1$, 有

$$f(\theta x + (1 - \theta)y) \leq \theta f(x) + (1 - \theta)f(y)$$

从几何意义上看, 上述不等式意味着点 $(x, f(x))$ 和 $(y, f(y))$ 之间的线段, 即从 x 到 y 的弦, 在函数 f 的图像上方。称函数 f 是严格凸的, 如果上式中的不等式当 $x \neq y$ 以及 $0 < \theta < 1$ 时严格成立。称函数 f 是凹的, 如果函数 $-f$ 是凸的; 称函数 f 是严格凹的, 如果 $-f$ 严格凸。

对于仿射函数, 上述不等式总成立。因此所有仿射函数 (包括线性函数) 是既凸且凹的。反之, 若某个函数是既凸又凹的, 则其是仿射函数。

函数是凸的, 当且仅当其在与其定义域相交的任何直线上都是凸的。换言之, 函数 f 是凸的, 当且仅当对于任意 $x \in \text{dom } f$ 和任意向量 v , 函数 $g(t) = f(x + tv)$ 是凸的 (其定义域为 $\{t \mid x + tv \in \text{dom } f\}$)。这个性质非常有用, 因为它容许我们通过将函数限制在直线上来判断其是否是凸函数。

第3章 最优化基本理论概述

3.1 一阶最优性条件

3.1.1 无约束优化

[3] 下面是对于无约束凸优化问题的一阶最优性条件, 对于其他有约束优化问题, 或者将其转化为无约束问题, 或者使用无约束问题近似求解 (参考自课本与西瓜书)。

定理 1 对无约束优化问题 (1.6.1), 若目标函数 f 是连续可微的凸函数, 则 x^* 为 (全局) 最优解的充分必要条件是 $\nabla f(x^*) = \mathbf{0}$ 。

证明若 x^* 是无约束优化问题 (1.6.1) 的 (全局) 最优解, 显然有 $\nabla f(x^*) = \mathbf{0}$ 。反过来, 若 x^* 是稳定点, 利用凸函数的性质, 对任意的 $x \in \mathbb{R}^n$,

$$f(x) - f(x^*) \geq \langle \nabla f(x^*), x - x^* \rangle = 0$$

故 x^* 是局部最优解, 又由于凸函数的性质: “凸函数是单峰函数, 局部最优即为全局最优”, x^* 又是全局最优解。

3.1.2 带约束优化 (KKT 条件)

在机器学习模型支持向量机 (Support Vector Machine) 和最大熵模型 (Maximum Entropy Model) 中, 需要用到 KKT 条件来作为理论支持, Karush-Kuhn-Tucker(KKT) 条件是非线性规划 (nonlinear programming) 最佳解的必要条件。KKT 条件将 Lagrange 乘数法 (Lagrange multipliers) 所处理涉及等式的约束优化问题推广至不等式。在实际应用上, KKT 条件 (方程组) 一般不存在代数解, 但将问题简化后我们可以认为符合 KKT 条件。我们在解释 KKT 条件前先引入拉格朗日乘子法。

拉格朗日乘子法 (Lagrange multipliers) 是一种寻找多元函数在一组约束下的极值的方法。通过引入拉格朗日乘子, 可将有 d 个变量与 k 个约束条件的最优化问题转化为具有 $d+k$ 个变量的无约束优化问题求解。

先考虑一个等式约束的优化问题. 假定 \mathbf{x} 为 d 维向量, 欲寻找 \mathbf{x} 的某个取值 \mathbf{x}^* , 使目标函数 $f(\mathbf{x})$ 最小且同时满足 $g(\mathbf{x}) = 0$ 的约束. 从几何角度看, 该问题的目标是在由方程 $g(\mathbf{x}) = 0$ 确定的 $d-1$ 维曲面上寻找能使目标函数 $f(\mathbf{x})$ 最小化的点. 此时不难得到如下结论:

- 对于约束曲面上的任意点 \mathbf{x} , 该点的梯度 $\nabla g(\mathbf{x})$ 正交于约束曲面;
- 在最优点 \mathbf{x}^* , 目标函数在该点的梯度 $\nabla f(\mathbf{x}^*)$ 正交于约束曲面.

由此可知, 在最优点 \mathbf{x}^* , 如附图 B.1 所示, 梯度 $\nabla g(\mathbf{x})$ 和 $\nabla f(\mathbf{x})$ 的方向必相同或相反, 即存在 $\lambda \neq 0$ 使得

$$\nabla f(\mathbf{x}^*) + \lambda \nabla g(\mathbf{x}^*) = \mathbf{0}$$

λ 称为拉格朗日乘子. 定义拉格朗日函数

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda g(\mathbf{x})$$

不难发现, 将其对 \mathbf{x} 的偏导数 $\nabla_{\mathbf{x}} L(\mathbf{x}, \lambda)$ 置零即得第一个式子, 同时, 将其对 λ 的偏导数 $\nabla_{\lambda} L(\mathbf{x}, \lambda)$ 置零即得约束条件 $g(\mathbf{x}) = 0$. 于是, 原约束优化问题可转化为对拉格朗日函数 $L(\mathbf{x}, \lambda)$ 的无约束优化问题。

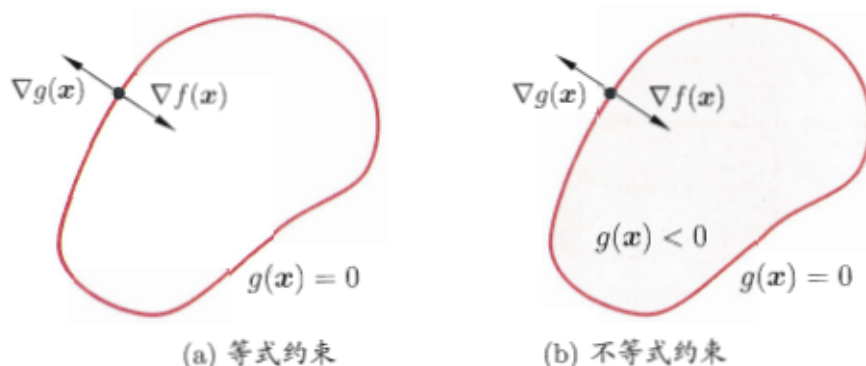


图 3.1 KKT 条件

现在考虑不等式约束 $g(\mathbf{x}) \leq 0$, 如图3.1所示, 此时最优点 \mathbf{x}^* 或在 $g(\mathbf{x}) < 0$ 的区域中, 或在边界 $g(\mathbf{x}) = 0$ 上. 对于 $g(\mathbf{x}) < 0$ 的情形, 约束 $g(\mathbf{x}) \leq 0$ 不起作用, 可直接通过条件: $\nabla f(\mathbf{x}) = 0$ 来获得最优点; 这等价于将 λ 置零然后对 $\nabla_{\mathbf{x}} L(\mathbf{x}, \lambda)$ 置零得到最优点. $g(\mathbf{x}) = 0$ 的情形类似于上面等式约束的分析, 但需注意的是, 此时 $\nabla f(\mathbf{x}^*)$ 的方向必与 $\nabla g(\mathbf{x}^*)$ 相反, 即存在常数 $\lambda > 0$ 使得 $\nabla f(\mathbf{x}^*) + \lambda \nabla g(\mathbf{x}^*) = 0$. 整合这两种情形, 必满足 $\lambda g(\mathbf{x}) = 0$. 因此, 在约束 $g(\mathbf{x}) \leq 0$ 下最小化 $f(\mathbf{x})$, 可转化为在如下约束下最小化拉格朗日函数:

$$\begin{cases} g(\mathbf{x}) \leq 0 \\ \lambda \geq 0 \\ \mu_j g_j(\mathbf{x}) = 0 \end{cases}$$

上式称为 Karush-Kuhn-Tucker (简称 KKT) 条件.

上述做法可推广到多个约束. 考虑具有 m 个等式约束和 n 个不等式约束, 且可行域 $\mathbb{D} \subset \mathbb{R}^d$ 非空的优化问题

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & h_i(\mathbf{x}) = 0 \quad (i = 1, \dots, m) \\ & g_j(\mathbf{x}) \leq 0 \quad (j = 1, \dots, n) \end{aligned}$$

引入拉格朗日乘子 $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_m)^T$ 和 $\mu = (\mu_1, \mu_2, \dots, \mu_n)^T$, 相应的拉格朗日函数为

$$L(\mathbf{x}, \lambda, \mu) = f(\mathbf{x}) + \sum_{i=1}^m \lambda_i h_i(\mathbf{x}) + \sum_{j=1}^n \mu_j g_j(\mathbf{x})$$

由不等式约束引入的 KKT 条件 ($j = 1, 2, \dots, n$) 为

$$\begin{cases} g_j(\mathbf{x}) \leq 0 \\ \mu_j \geq 0 \\ \mu_j g_j(\mathbf{x}) = 0 \end{cases}$$

3.2 二阶最优性条件

目标函数梯度为 0 的点被称为问题的稳定点，但是稳定点不一定是最优解，还可能是最大值解。所以要确认一个解是不是最优解，还需要考察目标函数在该点的二阶最优性条件。

3.2.1 无约束优化

定理 2.1 (二阶必要条件) 设 $x^* \in \mathbb{R}^n$ 是无约束优化问题 (1.6.1) 的局部最优解, 且 $f(x)$ 在 x^* 点附近二阶连续可微. 则 $\nabla f(x^*) = 0$, $\nabla^2 f(x^*)$ 半正定。

证明由定理 1.6.1 知 $\nabla f(x^*) = 0$. 若 $\nabla^2 f(x^*)$ 非半正定, 则存在单位向量 $d \in \mathbb{R}^n$, 使 $d^T \nabla^2 f(x^*) d < 0$. 对 $\alpha > 0$ 充分小, 利用 Taylor 展式,

$$f(x^* + \alpha d) = f(x^*) + \alpha \nabla f(x^*)^T d + \frac{1}{2} \alpha^2 d^T \nabla^2 f(x^*) d + o(\alpha^2) < f(x^*),$$

这与 x^* 是局部最优解矛盾. 结论得证。

上述结论给出的最优性条件不是充分的. 如单元函数 $f(x) = x^3$ 在 $x = 0$ 点同时满足一阶和二阶最优性必要条件, 但它并不是该函数的局部最优值点。

定理 2.2 (二阶充分条件) 设 $x^* \in \mathbb{R}^n$ 满足 $\nabla f(x^*) = 0$ 且 $\nabla^2 f(x^*)$ 正定, 则 x^* 是无约束优化问题 (1.6.1) 的严格局部最优解。

证明对任意充分靠近 x^* 的 $x \in \mathbb{R}^n$, 存在单位向量 $d \in \mathbb{R}^n$ 及充分小的 $\alpha > 0$ 使 $x = x^* + \alpha d$. 由 Taylor 展式及 $\nabla^2 f(x^*)$ 的正定性得

$$f(x) = f(x^*) + \frac{1}{2} \alpha^2 d^T \nabla^2 f(x^*) d + o(\alpha^2) > f(x^*)$$

从而, x^* 是 (1.6.1) 的严格局部最优解。

同样, 上述结论给出的二阶充分性条件也不是必要的. 如零点单元函数 $f(x) = x^4$ 的严格局部最优解, 但上述二阶充分性条件在该点并不成立. 由此推断, 无约束优化问题不存在充分必要的最优性条件。

无约束优化问题的二阶最优性条件是借助目标函数在局部最优值点邻域上的凸性来刻画. 一般的优化方法只能保证迭代点列的极限点满足一阶必要条件, 而不能保证满足二阶充分条件. 换句话讲, 这些方法只能得到稳定点, 只有在特殊情况下, 才能保证问题的稳定点为其全局最优解。

3.2.2 带约束优化

约束优化问题的最优值点未必满足 KKT 条件, 而 K-T 点也未必是最优值点. 那么, 在什么情况下, 约束优化问题的 K-T 点为最优值点呢?

考虑如下约束优化问题

$$\begin{aligned} \min \quad & f(x) = x_1^2 + x_2^2 \\ \text{s.t.} \quad & x_1^2 + x_2 - 1 \geq 0. \end{aligned}$$

容易验证 $x^* = (0, 1)^T$ 是其 K-T 点, 并且 $\nabla^2 f(x)$ 在 x^* 点正定. 但 x^* 并非该问题的局部最优值点. 事实上, 约束区域的边界点 $(x_1, 1 - x_1^2)^T$ 对应的目标函数值为 $1 - x_1^2 + x_1^4$. 只要 $x_1 \in (0, 1)$, 就有 $f(x) < f(x^*)$.

该例子说明, 尽管目标函数在 K-T 点 x^* 的 Hesse 阵正定, 但 x^* 未必是该问题的局部最优解, 原因很简单, 因为 x^* 不是目标函数的稳定点. 与无约束优化问题的二阶最优性条件不同, 约束优化问题的二阶最优性条件是建立在 Lagrange 函数关于 x 的 Hesse 阵在最优值点的临界锥上的半正定性上.

定理 2.3(二阶必要条件) 设 x^* 是约束优化问题

$$\min \{f(x) \mid c_i(x) = 0, i \in \mathcal{E}; c_i(x) \geq 0, i \in \mathcal{I}\}$$

的局部最优解, (x^*, λ^*) 为其 K-T 对. 若约束函数全是线性的或者 $\nabla c_i(x^*), i \in \mathcal{A}(x^*)$ 线性无关, 则对于满足 $s^T \nabla c_i(x^*) = 0, i \in \mathcal{A}(x^*)$ 的任意 $s \in \mathbb{R}^n$ 有

$$s^T \nabla_{xx} L(x^*, \lambda^*) s \geq 0.$$

定理 2.4(二阶充分条件) 设 (x^*, λ^*) 为约束优化问题的 K-T 对. 若对任意满足

$$\begin{cases} s^T \nabla c_i(x^*) = 0, & \text{若 } i \in \mathcal{E}, \\ s^T \nabla c_i(x^*) \geq 0, & \text{若 } \lambda_i^* = 0, i \in \mathcal{I}(x^*), \\ s^T \nabla c_i(x^*) = 0, & \text{若 } \lambda_i^* > 0, i \in \mathcal{I}(x^*) \end{cases}$$

的非零向量 $s \in \mathbb{R}^n$ 都有 $s^T \nabla_{xx} L(x^*, \lambda^*) s > 0$, 则 x^* 是约束优化问题 (7.2.2) 的严格局部最优解. 进一步, 存在 $\gamma > 0$ 和 $\delta > 0$ 使对任意的 $x \in N(x^*, \delta) \cap \Omega$, 成立

$$f(x) \geq f(x^*) + \gamma \|x - x^*\|^2.$$

优化方法大多基于上述条件判断收敛性, 所以不管在理论分析上还是实际使用中上述方法都发挥很大的作用。

第 4 章 最优化部分算法分析

下面我们列举了部分最优化的算法并将其分类，如表4.1。然后针对表格中的无约束最优化算法分别进行分析。（启发式算法将在拓展分析部分结合实例说明）

表 4.1 优化算法

一级分类	二级分类	算法
无约束优化	模式搜索法	坐标轮换法 单纯形法
	线搜索方法	最速下降法 牛顿法 共轭梯度法 拟牛顿法
	启发式算法	随机算法 禁忌算法 模拟退火算法 粒子群算法 遗传算法...
带约束优化		梯度投影方法 内点罚函数方法 外点罚函数方法 乘子罚函数方法

4.1 模式搜索法

在运筹学中，单纯形法是解决线性规划问题的一个有效的算法。线性规划就是在一组线性约束条件下，求解目标函数最优解的问题，是一类经典的最优化算法。具体理论分析参见王周宏老师的运筹学基础^[4]

4.2 线搜索方法

^[3] 在一学期的学习中，我发现步长规则在各类算法中都有使用，所以步长规则的选取是很重要的一个方面。以线搜索方法为例，设 d_k 为目标函数在 x_k 点的下降方向。为求目标函数的最小值，一个自然的想法是沿该方向寻求一个新点使目标函数有最大程度的下降，也就是取步长

$$\alpha_k = \arg \min_{\alpha \geq 0} f(x_k + \alpha d_k).$$

这样的 α_k 称为精确步长, 又称最优步长。虽然在实际中这种求解最优步长的算法在某些例子中表现不好, 可能因为其最优步长的求解计算量过大, 但是最优步长在理论上有很好的性质。根据最优性条件, 该步长满足如下正交性条件:

$$\mathbf{d}_k^T \nabla f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) = 0.$$

该步长规则称为最优步长规则, 又称精确线搜索步长规则。该步长规则下的线搜索算法模型为

线搜索方法, 即 $V \approx v_\pi$	
步 1	取初始点 $\mathbf{x}_0 \in \mathbb{R}^n$ 和参数 $\varepsilon \geq 0$. 令 $k = 0$
步 2	若 $\ \mathbf{g}_k\ \leq \varepsilon$, 算法终止; 否则, 进入下一步
步 3	计算下降方向 \mathbf{d}_k , 使 $\mathbf{d}_k^T \mathbf{g}_k < 0$
步 4	计算步长 $\alpha_k = \arg \min \{f(\mathbf{x}_k + \alpha \mathbf{d}_k) \mid \alpha \geq 0\}$
步 5	令 $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$, $k = k + 1$, 转步 2

在实际中, 可以不取最优步长进行优化, 对此也有很多种解法, 称为非精确线搜索方法。典型的方法有 Armijo、Wolfe、Strong Wolfe 步长规则。最速下降法、牛顿法、共轭梯度法、拟牛顿法等算法都会用到这些步长规则。

4.3 几类无约束优化算法框架

接下来介绍几类学到的无约束优化算法^[3]。

4.3.1 最速下降法

框架如下:

- 步 1. 取初始点 $\mathbf{x}_0 \in \mathbb{R}^n$ 和参数 $\varepsilon \geq 0$. 令 $k = 0$.
- 步 2. 若 $\|\mathbf{g}_k\| \leq \varepsilon$, 算法终止; 否则, 进入下一步.
- 步 3. 计算下降方向 \mathbf{d}_k , 使 $\mathbf{d}_k^T \mathbf{g}_k < 0$.
- 步 4. 计算步长 $\alpha_k = \arg \min \{f(\mathbf{x}_k + \alpha \mathbf{d}_k) \mid \alpha \geq 0\}$.
- 步 5. 令 $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$, $k = k + 1$, 转步 2.

4.3.2 牛顿法

框架如下:

- 步 1. 取初始点 \mathbf{x}_0 及精确参数 $\varepsilon \geq 0$, 令 $k = 0$.
- 步 2. 若 $\|\mathbf{g}_k\| \leq \varepsilon$, 算法终止, 否则进入下一步.
- 步 3. 令 $\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{G}_k^{-1} \mathbf{g}_k$, $k = k + 1$, 转步 2.

4.3.3 非线性共轭梯度法

框架如下:

- 步 1. 取初始点 \mathbf{x}_0 及终止参数 $\varepsilon \geq 0$, $\mathbf{d}_0 = -\mathbf{g}_0$. 令 $k = 0$.

步 2. 若 $\|g_k\| \leq \varepsilon$, 算法终止, 否则进入下一步.

步 3. 按照一定的步长规则 (FR, PR, CW, Dixon 规则) 选取步长 α_{k1}^1

步 4. 计算 $x_{k+1} = x_k + \alpha_k d_k, g_{k+1} = \nabla f(x_{k+1})$,

4.3.4 拟牛顿法

框架如下:

步 1. 取 $x_0 \in \mathbb{R}^n$, n 阶阵 H_0 和参数 $\varepsilon \geq 0$. 令 $k = 0$.

步 2. 计算 $d_k = -H_k g_k$. 令 $x_{k+1} = x_k + \alpha_k d_k$, 其中, 步长 α_k 由线搜索产生. 若 $\|g_{k+1}\| \leq \varepsilon$, 算法停止; 否则, 转下一步.

步 3. 按照一定的校正公式求 H_{k+1} , $k = k + 1$, 转步 2.

4.3.5 Gauss-Newton 算法

框架如下:

步 1. 取初始点 x_0 及精度参数 $\varepsilon \geq 0$, 令 $k = 0$.

步 2. 若 $\|g_k\| \leq \varepsilon$, 算法终止, 否则进入下一步.

步 3. 计算 Gauss-Newton 方向 $d_k = -(J_k^T J_k)^{-1} J_k^T r_k$

步 4. 利用线搜索步长规则选取步长 α_k .

步 5. 令 $x_{k+1} = x_k + \alpha_k d_k, k = k + 1$, 转步 2.

4.3.6 信赖域方法

这个方法的思想很重要, 强化学习最著名的算法 PPO 就是借助这个方法建立起来的, 也是因为这个方法, 强化学习真正进入大众的视野, 被人们广泛使用. 信赖域方法基本框架如下:

步 1. 取最大信赖域半径 $\hat{\Delta}$, 初始点 x_0 , 参数 $\Delta_0 \in (0, \hat{\Delta}]$, $\eta \in [0, \frac{1}{4})$, $\varepsilon \geq 0$, 令 $k = 0$.

步 2. 若 $\|g_k\| \leq \varepsilon$, 算法终止, 否则进入下一步.

步 3. 求解信赖域子问题 (2.3.1) 计算出 d_k , 计算 γ_k , 若 $\gamma_k < \frac{1}{4}$, 令 $\Delta_{k+1} = \frac{1}{4}\Delta_k$; 若 $\gamma_k > \frac{3}{4}$ 且 $\|d_k\| = \Delta_k$, 则令 $\Delta_{k+1} = \min\{2\Delta_k, \hat{\Delta}\}$; 否则, 令 $\Delta_{k+1} = \Delta_k$

步 4. 若 $\gamma_k > \eta$, 令 $x_{k+1} = x_k + d_k$; 否则 $x_{k+1} = x_k$. 令 $k = k + 1$, 转步 2.

4.3.7 LMF 算法

LMF 方法是 LM 方法加信赖域调参策略得到的算法, 能够提升 LM 方法的性能.

步 1. 选取初始点 x_0 , 参数 $\mu_0 > 0, \alpha, \beta \in (0, 1), \eta \in [0, \frac{1}{4})$, $\varepsilon \geq 0$, 令 $k = 0$.

步 2. 若 $\|g_k\| \leq \varepsilon$, 算法终止, 否则进入下一步.

步 3. 求解方程 $(J_k^T J_k + \mu_k I) d = -J_k^T r_k$ 计算迭代方向 d_k^{LM} , 计算下降率 γ_k , 若 $\gamma_k < \frac{1}{4}$, 令 $\mu_{k+1} = \frac{\mu_k}{\alpha}$; 若 $\gamma_k > \frac{3}{4}$, 则令 $\mu_{k+1} = \beta \mu_k$. 否则, $\mu_{k+1} = \mu_k$

步 4. 若 $\gamma_k > \eta$, 令 $x_{k+1} = x_k + d_k$; 否则 $x_{k+1} = x_k$. 令 $k = k + 1$, 转步 2.

4.4 几类带约束优化算法框架

下面是三种带约束优化算法^[3].

4.4.1 积极集方法算法

框架如下：

取第 k 步迭代点 $x_k \in \Omega$, 选取 $\mathcal{A}(x_k)$ 的估计指标集 S_k 满足 $\mathcal{E} \subset S_k \subset \mathcal{A}(x_k)$

计算搜索方向: d_k 迭代步长: α_k

更新迭代点: $x_{k+1} = x_k + \alpha_k d_k \in \Omega, Q(x_{k+1}) < Q(x_k)$,

更新指标集: $\mathcal{E} \subset S_{k+1} \subset \mathcal{A}(x_{k+1})$.

4.4.2 投影梯度算法

框架如下：

初始步取 $\varepsilon > 0, \beta > 0, \sigma, \gamma \in (0, 1), x_0 \in \Omega$. 令 $k = 0$

迭代步令 $x_k(1) = P_\Omega(x_k - \nabla f(x_k))$. 若 $\|x_k - x_k(1)\| \leq \varepsilon$, 算法终止. 否则, 令

$x_{k+1} = P_\Omega(x_k - \alpha_k \nabla f(x_k))$, 其中, $\alpha_k = \beta \gamma^{m_k}$, m_k 为满足下式的最小非负整数:

$$f(P_\Omega(x_k - \beta \gamma^m \nabla f(x_k))) \leq f(x_k) + \sigma \langle \nabla f(x_k), P_\Omega(x_k - \beta \gamma^m \nabla f(x_k)) - x_k \rangle.$$

4.4.3 乘子罚函数算法

框架如下：

步 1. 给定 $\pi_0 > 0, \lambda^0 = \mathbf{0}$, 初始点 $x_{-1} \in \mathbb{R}^n$, 增长因子 $\gamma > 1$ 和允许误差 $\varepsilon > 0$. 今 $\square k = 0$.

步 2. 以 x_{k-1} 为初始点, 用无约束优化方法计算函数 $P(x, \lambda^k, \pi_k)$ 的最小值点 x_k .

步 3. 若 $\max\{|c_i(x_k)| \mid i \in \mathcal{E}\} \leq \varepsilon$, 算法终止. 否则, 转下一步.

步 4. 若 $\|c(x_k)\|_\infty \geq \|c(x_{k-1})\|_\infty$, 令 $\pi_{k+1} = \gamma \pi_k, \lambda^{k+1} = \lambda^k$, 置 $k = k + 1$, 转步 2. 否则, 转步 5.

步 5. 若 $\pi_k > \pi_{k-1}$ 或 $\|c(x_k)\|_\infty \leq \frac{1}{4} \|c(x_{k-1})\|_\infty$, 令 $\pi_{k+1} = \pi_k$, 根据 (10.3.3) 调整 λ^{k+1} , 置 $k = k + 1$, 转步 2. 否则, 令 $\pi_{k+1} = \gamma \pi_k, \lambda^{k+1} = \lambda^k$, 置 $k = k + 1$, 转步 2.

第 5 章 强化学习与最优化

在人工智能领域，和最优化在技术上联系最多的是深度学习和机器学习，最优化为损失函数的最小化提供了各种解决办法。但是要说在领域发展之初，算法一开始建立时的“motivation”与最优化理论联系最紧密的，当属强化学习。

强化学习将序列决策过程建模成马尔科夫决策过程 (**Markov Decision Process, MDP**)，定义为由状态空间、动作空间、转移概率、奖赏函数和折现因子组成的五元组，然后主要研究序列决策情形下的若干问题。所谓序列决策的情形，如图 1 所示，是指智能体 (**Agent**) 与环境 (**Environment**) 进行交互，每次智能体观测到环境的一个状态 (**State**)，然后根据自己的策略 (**Policy**) 选择一个动作 (**Action**) 并且施加给环境，环境根据这个动作转移到下一个状态并且反馈回一个奖赏 (**Reward**) 信号，智能体再选择下一个动作施加给环境，如此循环。在这个过程中涉及到的一个关键概念就是策略，策略决定着智能体如何选择下一步动作，即如何与环境进行交互，因此，强化学习主要研究如何找到最优的策略，使得智能体最终获得的累积奖赏最大。其四要素如图 5.1 所示。

表 5.1 强化学习四要素与其输入输出

4 elements of RL	input	output
policy	state	action
reward signal	action state	reward
value function	state	reward
model of the environment	action state	state reward

一方面，强化学习的理论框架非常完善，二十年来几乎没有人能跳脱出该框架之外，而是在 MDP 这个理论框架内不断设计算法，提高算法性能。另一方面，尽管理论框架很完善，但是距离实际应用却仍有很大距离。例如 AlphaGo 虽然在围棋上能够达到远超人类的水平，但这只是炫技的成分大过实用性，归根结底是因为围棋的规则是容易量化表示的，而现实中的规则却很难量化。所以在现实生活中的很多问题中强化学习距离达到人类的平均水平还有很长一段路要走。

困难就是机会，在强化学习领域，众多研究人员不断提出新的算法。每一次算法的改进，都能朝着最优策略迈进。在这个过程中，大家好像都不约而同地选择了优化的知识来作为设计算法的出发点，SARSA 和 Q-learning 是，DQN 也是，甚至 TRPO 直接借用共轭梯度法把算法性能提高了一倍。所以说强化学习的算法的“motivation”与最优化理论的联系是非常紧密的。其实这并不是巧合，从强化学习的目标来看，寻找最优策略本质上就是一个抽象的优化问题：在策略函数空间中不断接近最优的那个策略函数。

在这一部分，作者首先介绍了强化学习的理论框架：马氏决策过程 (MDP^[5])，然后介绍了动态规划^{[6][7]} 算法来表达 Bellman 方程^[8] 的重要意义，然后介绍 SARSA^[9] 和 Q-learning^{[10][11]}

算法和最优化的联系，然后介绍带领强化学习走向深度强化学习的“领路人”：DQN^{[12][13]} 算法以及和最优化理论的联系。最后介绍如何用共轭梯度法设计强化学习算法（TRPO 算法^[14]），以及如何简化从而进一步提高算法性能（PPO 算法^[15]）。有趣之极，令人惊叹！

5.1 有限维马氏决策过程

^[5] 在有限马氏决策过程（MDP）中，状态、动作和奖励（ S , A 和 R ）的集合都具有有限数量的元素。随机变量 R_t 和 S_t 具有良好定义的离散概率分布，且仅取决于先前的状态和动作。在给定先前状态和动作的特定值的情况下，这些值在时间 t 出现的概率：

$$p(s', r | s, a) \doteq \Pr \{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$$

注意： $\sum_{s' \in S} \sum_{r \in R} p(s', r | s, a) = 1$, for all $s \in S, a \in \mathcal{A}(s)$ 。

基于上式可以求得下列几个函数：

状态转移函数（三元函数）：

$$p(s' | s, a) \doteq \Pr \{S_t = s' | S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r | s, a) .$$

State – action 期望奖励（二元函数）：

$$r(s, a) \doteq \mathbb{E} [R_t | S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r | s, a) ,$$

State – action – next state 期望奖励（三元函数）：

$$r(s, a, s') \doteq \mathbb{E} [R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \frac{p(s', r | s, a)}{p(s' | s, a)}$$

MDP 的基本步骤如下图所示：

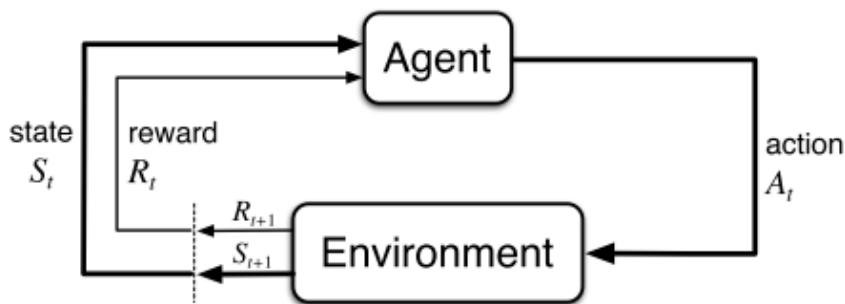


图 5.1 智能体与环境交互流程（MDP）

5.1.1 值函数与动作值函数

在 MDP 框架下，给定策略 π ，具体而言为智能体在给定状态下选择动作的方式（可以以概率形式给出），即 $p(s \text{ 状态下选择动作 } a) = \pi(a|s)$ 。定义状态 s 的值函数如下：

$$v_{\pi}(s) \doteq \mathbb{E}_{\pi} [G_t | S_t = s] = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \text{ for all } s \in \mathcal{S}$$

表示未来的期望收益。可以写成如下的迭代形式：

$$\begin{aligned} v_{\pi}(s) &\doteq \mathbb{E}_{\pi} [G_t | S_t = s] \\ &= \mathbb{E}_{\pi} [R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a | s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \mathbb{E}_{\pi} [G_{t+1} | S_{t+1} = s']] \\ &= \sum_a \pi(a | s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_{\pi}(s')], \quad \text{for all } s \in \mathcal{S} \end{aligned}$$

上式即为贝尔曼方程。

下图为在均匀随机策略下的 **Grid World** 案例。在此例中，除移动至左上角或右下角的 **Reward** 为 0 外，其他状态之下的动作 **Reward** 均为-1。

	1	2	3	4
1	0.0	-14.0	-20.0	-22.0
2	-14.0	-18.0	-20.0	-20.0
3	-20.0	-20.0	-18.0	-14.0
4	-22.0	-20.0	-14.0	0.0

图 5.2 Grid World 中均匀随机策略下的值函数

同样可以定义状态 s 和动作 a 下的动作值函数：

$$q_{\pi}(s, a) \doteq \mathbb{E}_{\pi} [G_t | S_t = s, A_t = a] = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

5.1.2 最优策略和最优值函数

强化学习的目标是求解问题的最优策略，即每一次动作的选择都遵循未来期望奖励最大的原则。从值函数的角度来看，就是求解最大值函数对应的策略 π 。即最优值函数：

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s)$$

对所有 $s \in S$.

相应的可以定义最优动作值函数：

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a)$$

对所有 $s \in S$ 和 $a \in \mathcal{A}(s)$ 。

做如下推导将值函数和动作值函数连接起来：

$$q_*(s, a) \doteq \mathbb{E}_* [R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] = \mathbb{E} [R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

再用上式推导**值函数的贝尔曼最优方程**。因为最优值函数需要选取接下来使得未来期望奖励最大的动作。即：

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*} [G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*} [R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E} [R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_*(s')] . \end{aligned}$$

相应的有**动作值函数的贝尔曼最优方程**。

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

5.2 动态规划求解最优策略和最优值函数

接下来利用上面求得两个贝尔曼最优方程迭代求解最优策略和最优值函数^[5]。

5.2.1 策略评价

因为贝尔曼（最优）方程已经被写成迭代式的形式。所以可以用迭代的方法来求解给定策略的值函数。即

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_{\pi} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_k(s')] \end{aligned}$$

所以求解策略的值函数的迭代过程可以写成下列形式：

策略评价迭代算法，即 $V \approx v_\pi$

输入： π , 给定的策略

算法参数： $\theta > 0$ 迭代终止条件

初始化 $V(s)$ ：除了 $V(\text{最终状态}) = 0$ 外，其他 $s \in S^+$ 的值函数任取。

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in S$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a | s) \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

5.2.2 策略改进

若对于策略 π 和 π' 来说，如果有式子 $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ 成立，则认为后者优于前者。

证明：

$$\begin{aligned}
 v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
 &= \mathbb{E} [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)] \\
 &= \mathbb{E}_{\pi'} [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
 &\leq \mathbb{E}_{\pi'} [R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
 &= \mathbb{E}_{\pi'} [R_{t+1} + \gamma \mathbb{E}_{\pi'} [R_{t+2} + \gamma v_\pi(S_{t+2})] | S_t = s] \\
 &= \mathbb{E}_{\pi'} [R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \\
 &\leq \mathbb{E}_{\pi'} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s] \\
 &\vdots \\
 &\leq \mathbb{E}_{\pi'} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots | S_t = s] \\
 &= v_{\pi'}(s)
 \end{aligned}$$

所以基于策略 π ，选择策略 π' 的方式是：

$$\begin{aligned}
 \pi'(s) &\doteq \arg \max_a q_\pi(s, a) \\
 &= \arg \max_a \mathbb{E} [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\
 &= \arg \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')],
 \end{aligned}$$

我们称这样选择出来的策略 π' 为贪婪策略，因为它始终选择最优动作。

如果 $v_\pi = v_{\pi'}$ 那么有：

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

所以 π 和 π' 满足贝尔曼最优方程。进而两个策略相同且为最优策略。

5.2.3 策略迭代

将策略评价部分和策略改进部分连接起来，就可以得到一个完整的求解最优策略和最优值函数的算法。即动态规划算法。

策略迭代算法，即 $\pi \approx \pi^*$

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in S$

2. Policy Evaluation

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in S$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a | s) \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

3. Policy Improvement

policy-stable \leftarrow true

For each $s \in S$:

old-action $\leftarrow \pi(s)$

$\pi(s) \leftarrow \arg \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$

If *old-action* $\neq \pi(s)$, then *policy-stable* \leftarrow false

If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

上述算法即为传统的 **DP** 算法。虽然可以求解最优策略和最优值函数，但是由于策略评价和策略改进两个部分的运算量差别很大，可能会导致计算过程中长时间陷入第一个部分，并且对于求解最优策略来说这些计算量是无意义或者意义不大的。

对此我们也有相应的优化方法，如在策略评价部分只迭代一次来减少这一部分的计算量，尽管理论上这种方法收敛至最优解需要无限步迭代。（ps: 蒙特卡洛方法估计动作值函数是没有被严格证明的但也很常用，因为它是很显然的）。或者在迭代部分不做求和而是求最大动作对应的值函数，这种方法的理论依据是贝尔曼最优方程。由于策略是逐渐接近最优策略的（由第二部分可以保证），这样求解的值函数也会逐渐收敛至最优策略的值函数，并且减少了很多的计算量。这种方法称为值迭代方法。

上面两种方法的思想是异步动态规划的思想。其本质是减少优化过程中不必要的计算，提高其计算效率。按照这种思想可以设计和上述方法类似的其他算法来减少计算量。不论如何，按照上面的思想设计算法的方式有一个统一的模式：**Generalized Policy Iteration(GPI)**（广义策略迭代）。

5.3 Sarsa 算法

介绍完强化学习基本的理论框架并且使用经典的动态规划进行求解之后，现在来介绍强化学习和最优化的联系。仔细观察上面动态规划算法我们可以发现：如果要使用动态规划算法进行求解，就需要提前知道所有的状态空间和动作空间的转移概率和奖励，然而这在现实中是不可行的。大多数情况下我们只能知道一部分的采样值，所以只能根据这些采样值来进行计算。由此产生了 Sarsa、和 Q-Learning 算法。

从优化的角度来看，既然强化学习的目标为寻找最优策略，且由 bellman 最优方程知寻找最优策略和寻找最优价值函数是等价的，所以可以仅使用采样数据由 bellman 方程的迭代形式设计如下迭代形式来逐步接近最优价值函数。

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

然后我们用贪婪算法来选取在某个状态下动作价值最大的那个动作，即 $\arg \max_a Q(s, a)$ 。这样似乎已经形成了一个完整的强化学习算法：用贪婪算法根据动作价值选取动作来和环境交互，再根据得到的数据用时序差分算法更新动作价值估计。

然而这个简单的算法存在两个需要进一步考虑的问题。第一，如果要用时序差分算法来准确地估计策略的状态价值函数，我们需要用极大量的样本来进行更新。但实际上我们可以忽略这一点，直接用一些样本来评估策略，然后就可以更新策略了。我们可以这么做的原因是策略提升可以在策略评估未完全进行的情况进行，动态规划里的价值迭代就是这样，这其实是广义策略迭代 (generalized policy iteration) 的思想。第二，如果在策略提升中一直根据贪婪算法得到一个确定性策略，可能会导致某些状态动作对 (s, a) 从不出现，以至于无法对其动作价值进行估计，进而无法保证策略提升后的策略比之前的好。简单常用的解决方案是不再一味使用贪婪算法，而是采用一个 ϵ -贪婪策略：有 $1 - \epsilon$ 的概率采用动作价值最大的那个动作，另外有 ϵ 的概率从动作空间中随机采取一个动作，其公式表示为：

$$\pi(a | s) = \begin{cases} \epsilon/|\mathcal{A}| + 1 - \epsilon & \text{如果 } a = \arg \max_{a'} Q(s, a') \\ \epsilon/|\mathcal{A}| & \text{其他动作} \end{cases}$$

现在，我们就可以得到一个实际的基于时序差分方法的强化学习算法。这个算法被称为 Sarsa，因为它的动作价值更新用到了当前状态 s 、当前动作 a 、获得的奖励 r 、一个状态 s' 和下一个动作 a' ，将这些符号拼接后就得到了算法名称。Sarsa 的具体算法如下：

- 初始化 $Q(s, a)$
- for 序列 $e = 1 \rightarrow E$ do:
- - 得到初始状态 s
- - 用 ϵ -greedy 策略根据 Q 选择当前状态 s 下的动作 a
- - for 时间步 $t = 1 \rightarrow T$ do:
- - - 得到环境反馈的 r, s'
- - - 用 ϵ -greedy 策略根据 Q 选择当前状态 s' 下的动作 a'
- - - $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$

```

- - -  $s \leftarrow s', a \leftarrow a'$ 
- -end for
- end for
    
```

而从最优化的角度出发，这件事情就变得非常显而易见，我的目标为寻找真实的价值函数，最简单的方法就是设计公式不断朝这个方向迭代。即不断更新价值函数使价值函数接近 $r + \gamma Q(s', a')$ 。从而得出真实的价值函数。有了真实的价值函数那么自然就可以导出最优策略。（真实与最优的关系：价值函数是用来刻画状态未来期望价值的，如果价值函数能够刻画现实中的动作状态对，即是“真实的”，那么从其上导出的贪婪策略必为最优策略）。下图为在经典控制问题“悬崖漫步”上 SARSA 算法的效果图：

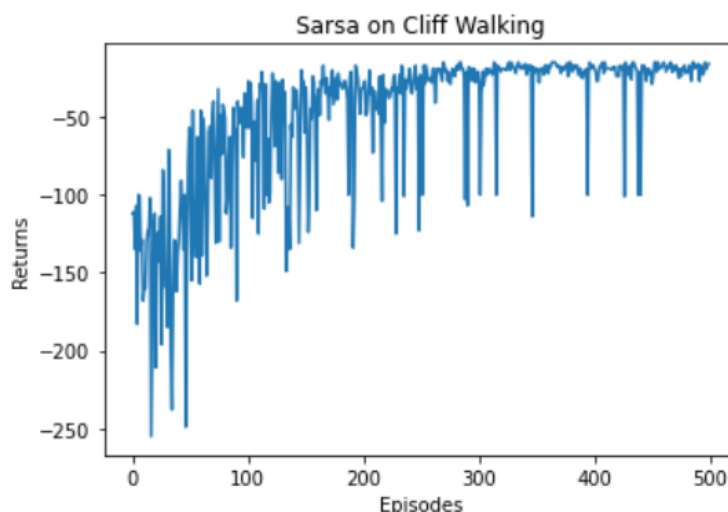


图 5.3 SARSA 算法效果图（on Cliff Walking）

5.4 Q-learning 算法

不断更新价值函数使价值函数接近 $r + \gamma Q(s', a')$ ，目标是寻找真实的价值函数，但是 SARSA 算法里使用的是 ϵ -greedy 策略，这与我们分析的“真实的价值函数的贪婪策略为最优策略”是相矛盾的。所以按照优化的思想，这里应该是按照贪婪策略取价值函数。基于这个思想的就是非常著名的基于时序差分算法的强化学习算法——Q-learning。该方法自 1994 年被提出^[10]后，就一直被广泛使用。直到现在，这个方法及其理论仍然经久不衰。

Q-learning 和 Sarsa 的最大区别在于 Q-learning 的时序差分更新方式为

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Q-learning 算法的具体流程如下：

```

- 初始化  $Q(s, a)$ 
- for 序列  $e = 1 \rightarrow E$  do:
- -得到初始状态  $s$ 
- - for 时间步  $t = 1 \rightarrow T$  do :
    
```

```

--- 用  $\epsilon$ -greedy 策略根据  $Q$  选择当前状态  $s$  下的动作  $a$ 
--- 得到环境反馈的  $r, s'$ 
       $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
---    $s \leftarrow s'$ 
-- end for
- end for
  
```

用优化的思想来理解 Q-learning，即 Q-learning 是在直接迭代接近 Q^* ，因为动作价值函数的贝尔曼最优方程是

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) \max_{a'} Q^*(s', a')$$

怎么样，从优化的角度看，这些算法设计的动机是不是就显得非常简洁？所以说，强化学习的算法可以看作是在特定任务情景下的优化算法。下图为 Q-learning 算法在“悬崖漫步”问题上的效果图：



图 5.4 Q-learning 算法效果图 (on Cliff Walking)

5.5 DQN 算法

虽然上述两种算法直接借用 Bellman 方程的性质巧妙地设计了优化算法，但是算法的性能较差。一方面，这是由于采样数据的不全面。另一方面是因为策略函数和值函数是非常复杂的抽象函数，直接使用这种迭代的形式进行优化很难接近其最优解。并且随着状态动作空间维数的上升，上述两种方法均不可行。

我们知道神经网络具有强大的表达能力，所以 2013 年有学者使用深度神经网络来近似动作值函数 Q (value-base)，于是强化学习算法便和优化算法更为紧密联系了起来（因为深度神经网络中最重要的一步就是基于损失函数做梯度下降^[16]）。

5.5.1 DQN 算法原理

那么 Q 网络的损失函数是什么呢？回顾一下 Q-learning 的更新规则：

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a) \right]$$

上述公式用时序差分 (temporal difference, TD) 学习目标 $r + \gamma \max_{a' \in \mathcal{A}} Q(s', a')$ 来增量式更新 $Q(s, a)$ ，也就是说要使 $Q(s, a)$ 和 TD 目标 $r + \gamma \max_{a' \in \mathcal{A}} Q(s', a')$ 靠近。于是，对于一组数据 $\{(s_i, a_i, r_i, s'_i)\}$ ，我们可以很自然地将 Q 网络的损失函数构造为均方误差的形式：

$$\omega^* = \arg \min_{\omega} \frac{1}{2N} \sum_{i=1}^N \left[Q_{\omega}(s_i, a_i) - \left(r_i + \gamma \max_{a'} Q_{\omega}(s'_i, a') \right) \right]^2$$

所以，一方面强化学习算法现在被归结为一类非线性最小二乘问题。另一方面将 Q-learning 扩展到神经网络形式——深度 Q 网络 (deep Q network, DQN) 算法。这极大的提高了算法的可操作性，因为我们有很多方式来处理最小二乘和神经网络这两类问题。由于 DQN 是离线策略算法，因此我们在收集数据的时候可以使用一个 ϵ -贪婪策略来平衡探索与利用，将收集到的数据存储起来，在后续的训练中使用。从而帮助 DQN 取得稳定、出色的性能。下图为 DQN 算法在 Open AI 提供的 gym 训练环境：cartpole-v0 上的效果图：

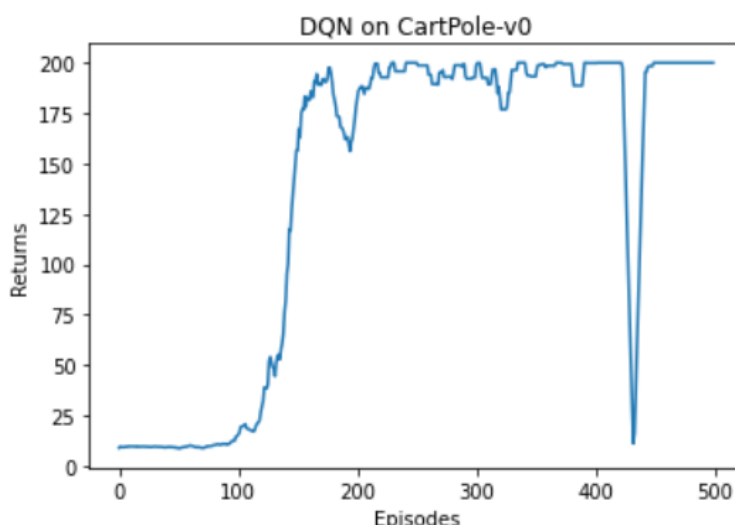


图 5.5 DQN 算法效果图（车杆模型）

5.6 Actor-Critic 算法

DQN 用神经网络近似了价值函数，自然地可以问策略函数是否可以近似。在 DQN 之后有学者建立了策略梯度的理论（这个理论非常有趣，但它过于复杂，可以参考博客^[17]），该理论直接建立了动作值函数和策略梯度的关系，于是可以使用神经网络来直接近似策略函数。所以 DQN 之后就有了 Actor-Critic 算法。

5.6.1 策略梯度理论

要求策略函数的梯度，首先需要将策略参数化。假设目标策略 π_θ 是一个随机性策略，并且处处可微，其中 θ 是对应的参数。我们可以用一个线性模型或者神经网络模型来为这样一个策略函数建模，输入某个状态，然后输出一个动作的概率分布。我们的目标是要寻找一个最优策略并最大化这个策略在环境中的期望回报。我们将策略学习的目标函数定义为：

$$J(\theta) = \mathbb{E}_{s_0} [V^{\pi_\theta}(s_0)]$$

其中， s_0 表示初始状态。现在有了目标函数，我们将目标函数对策略 θ 求导，得到导数后，就可以用梯度上升方法来最大化这个目标函数，从而得到最优策略。

定义策略 π 下的状态访问分布，在此用 v^π 表示。然后我们对目标函数求梯度，可以得到如下式子，更详细的推导过程在^[17]给出。

$$\begin{aligned} \nabla_\theta J(\theta) &\propto \sum_{s \in S} v^{\pi_\theta}(s) \sum_{a \in A} Q^{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a | s) \\ &= \sum_{s \in S} v^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a | s) Q^{\pi_\theta}(s, a) \frac{\nabla_\theta \pi_\theta(a | s)}{\pi_\theta(a | s)} \\ &= \mathbb{E}_{\pi_\theta} [Q^{\pi_\theta}(s, a) \nabla_\theta \log \pi_\theta(a | s)] \end{aligned}$$

在策略梯度中，可以把梯度写成下面这个形式：

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

其中 ψ_t 可以有很多种形式：

1. $\sum_{t'=0}^{\infty} \gamma^{t'} r_{t'}$ ：轨迹的总回报
2. $\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}$ ：动作 a_t 之后的回报
3. $\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} - b(s_t)$ ：基准线版本的改进
4. $Q^{\pi_\theta}(s_t, a_t)$ ：动作价值函数
5. $A^{\pi_\theta}(s_t, a_t)$ ：优势函数
6. $r_t + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)$ ：时序差分残差

5.6.2 Actor-Critic 算法原理与优化方法

Actor-Critic 算法一方面用神经网络近似策略函数，利用上面的结果，进行梯度下降，这一部分称为 Actor 部分；另一方面借鉴 DQN 算法，用神经网络近似 Q 函数，这一部分称为 Critic 部分。这样就将问题变成了两个部分的优化问题，增强了算法稳定性，提高了策略的表达能力。Actor-Critic 算法框架如下：

- 初始化策略网络参数 θ ，价值网络参数 ω
- 不断进行如下循环 (每个循环是一条序列)：
 - 用当前策略 π_θ 采样轨迹 $\{s_1, a_1, r_1, s_2, a_2, r_2 \dots\}$
 - 为每一步数据计算： $\delta_t = r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t)$

- 更新价值参数 $w = w + \alpha_w \sum_t \delta_t \nabla_w V_w(s)$
- 更新策略参数 $\theta = \theta + \alpha_\theta \sum_t \delta_t \nabla_\theta \log \pi_\theta(a | s)$

实际实验表明，将问题切分为两个优化问题来互相帮助进行学习，可以有效提高算法性能。下图为 Actor-Critic 算法在车杆模型上的效果图。相比与 DQN 算法，算法性能和稳定性提高了许多。

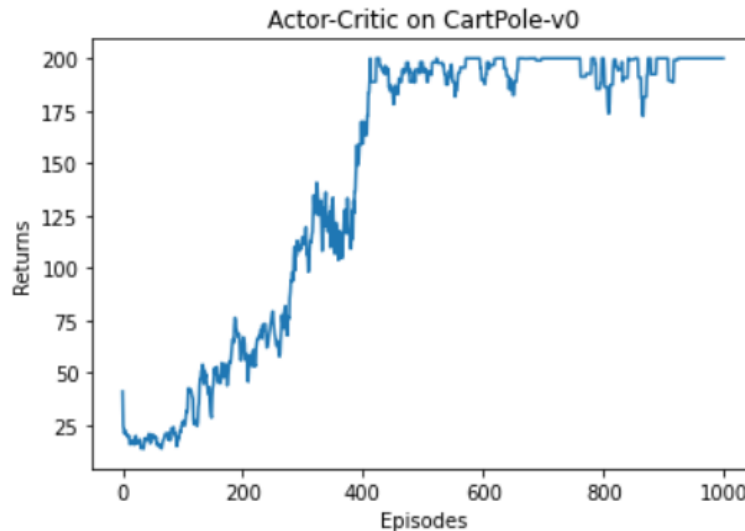


图 5.6 Actor-Critic 算法效果图（车杆模型）

5.7 TRPO 算法

[14] 尽管有了 Actor-Critic 算法，但是人们发现这个算法的效果仍然不理想，这是因为神经网络本身的不稳定性。所以人们从最优化的角度对该问题进行了重构，提出了基于共轭梯度法的强化学习算法 TRPO，这使得算法的效果大大提升。

5.7.1 TRPO 算法原理

Actor-Critic 算法是设计衡量策略好坏的价值函数 $J(\theta)$ ，通过梯度上升的方法来最大化这个价值函数。但缺点是当用深度模型作为策略网络时，若步长太长，策略会突然显著变差。即方差大。所以 TRPO 考虑在更新时找到一块信赖域 (Trust Region)，在这个区域上更新策略时可以保证策略性能是上升的。这样就可以平稳地进行训练。

TRPO 用下式说明了新旧策略的价值函数之间相差了一个新策略下的 TD 误差的期望（即等式右边第二项）。

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a | s) A_\pi(s, a)$$

只要保证信赖域里该期望为正值即可保证策略性能提升。但同时这是一个 NP 难问题，因为新策略未知。于是 TRPO 作者用当前状态密度近似新策略下的状态密度。

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a | s) A_\pi(s, a)$$

但这又遇到了新问题，这个近似操作只有在两个分布相似度较高时才能有效（即下列不等式成立），因此要给出两个策略之间差别的刻画：KL 散度^[14]。即下式：

$$\eta(\tilde{\pi}) \leq L_{\pi}(\tilde{\pi}) + CD_{\text{KL}}^{\max}(\pi, \tilde{\pi}), \text{ where } C \text{ is constant}$$

综上所述，可以给出一个整体的带约束优化公式，不等式约束给出了一个信赖区域，强化学习算法发展到这里，直接借鉴了约束优化理论来建立问题框架。

$$\begin{aligned} & \underset{\theta}{\text{minimize}} L_{\theta_{\text{old}}}(\theta) \\ & \text{subject to } D_{\text{KL}}^{\max}(\theta_{\text{old}}, \theta) \leq \delta. \end{aligned}$$

直接求解该问题比较麻烦，所以需要对问题进行一阶、二阶近似。为了求解此优化问题，TRPO 接着对上式进行泰勒展开，用 KKT 条件得到了上述问题的解：

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

其中， g 为梯度， H 为约束的 Hessian 矩阵。

由于神经网络表示的策略函数的参数数量非常多，一般为 10 的三次方数量级以上，这样计算和储存 Hessian 矩阵非常困难。所以作者使用共轭梯度法避开了这个问题。从而设计出可行的 TRPO 算法。算法框架如下：

初始化策略网络参数 θ 和价值网络参数 ω

- for 序列 $e = 1 \rightarrow E$ do:

-- 用当前策略 π_{θ} 采样轨迹 $\{s_1, a_1, r_1, s_2, a_2, r_2, \dots\}$

-- 根据收集到的数据和价值网络估计每个状态动作对的优势 $A(s_t, a_t)$

-- 计算策略目标函数的梯度 g

-- 用共轭梯度法计算 $x = H^{-1}g$

-- 用线性搜索找到一个 i 值，并更新策略网络参数

-- $\theta_{k+1} = \theta_k + \alpha^i \sqrt{\frac{2\delta}{x^T H x}} x$, 其中 $i \in \{1, 2, \dots, K\}$

-- 为能提升策略并满足 KL 距离限制的最小整数

-- 更新价值网络参数 (与 Actor-Critic 中的更新方法相同)

- end for

其中共轭梯度法计算 $x = H^{-1}g$ 框架如下：

- 初始化 $r_0 = g - Hx$, $p_0 = r_0$, $x_0 = 0$

- for $k = 0 \rightarrow N$ do:

-- $\alpha_k = \frac{r_k^T r_k}{p_k^T H p_k}$

-- $x_{k+1} = x_k + \alpha_k p_k$

-- $r_{k+1} = r_k - \alpha_k H p_k$

-- 如果 $r_{k+1}^T r_{k+1}$ 非常小，则退出循环

-- $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$

```

- -  $p_{k+1} = r_{k+1} + \beta_k p_k$ 
- end for
- 输出  $x_{N+1}$ 

```

实验表明，借助共轭梯度法建立的 TRPO 算法效果远优于前面几种直接进行梯度下降的算法。

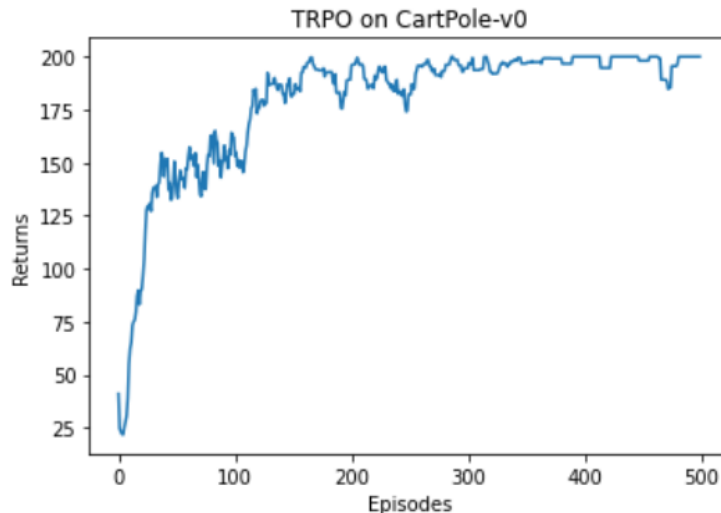


图 5.7 TRPO 算法效果图（车杆模型）

5.8 PPO 算法

虽然 TRPO 用共轭梯度法避开了 Hessian 矩阵的直接计算和储存，但是算法每次更新的计算量仍然很大。所以 TRPO 的作者两年后提出了 PPO 算法用两种方式克服了这个问题。由于 TRPO 的优化问题为不等式约束问题，所以人们将问题转化为罚函数的方法建立了 PPO 算法，它的效果更佳，打败了当时所有的强化学习算法。这使得 PPO 算法称为了强化学习领域近五年来最炙手可热的算法。这再一次地说明了善用最优化带来好处是无穷无尽的。

在前面优化算法的基础之上，PPO 做出的改变非常简单，把 TRPO 里的优化问题写为重要性采样的形式，然后用两种方式避开了 Hessian 矩阵的求解。

$$\begin{aligned}
 & \underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] \\
 & \text{subject to} \quad \hat{\mathbb{E}}_t \left[\text{KL} \left[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t) \right] \right] \leq \delta
 \end{aligned}$$

方法 1：自适应罚函数方法

$$L^{KL PEN}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t - \beta \text{KL} \left[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t) \right] \right]$$

```

- 计算  $d = \hat{\mathbb{E}}_t \left[ \text{KL} \left[ \pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t) \right] \right]$ 
- If  $d < d_{\text{targ}}/1.5, \beta \leftarrow \beta/2$ 

```

– If $d > d_{\text{targ}} \times 1.5, \beta \leftarrow \beta \times 2$

方法 2：截断法, 即把大的重要比直接截断掉。这种方法更为直接。

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t [r_t(\theta) \hat{A}_t]$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t [\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t)]$$

虽然这两种方法非常的简单，但是效果却非常的好，这说明这两个操作对于 TRPO 中表述的优化问题的近似程度更高。算法收敛速度更快，稳定性更好，下面是 PPO 在车杆环境的效果图：

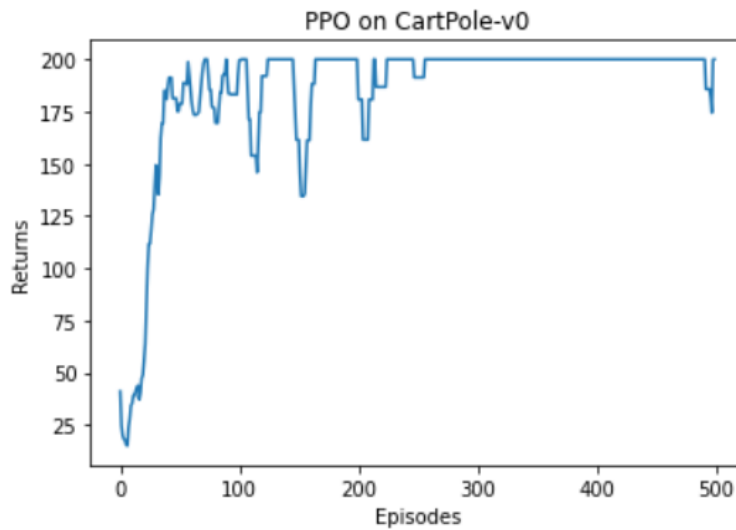


图 5.8 PPO 算法效果图（车杆模型）

5.9 小结

至此，我们介绍了强化学习的理论框架：马氏决策过程（MDP^[5]），介绍了动态规划^{[6][7]}算法，介绍了 SARSA^[9] 和 Q-learning^{[10][11]} 算法和最优化的联系，介绍了 DQN^{[12][13]} 算法以及与最优化理论的联系。介绍了如何用共轭梯度法设计强化学习算法（TRPO 算法^[14]），以及如何简化从而进一步提高算法性能（PPO 算法^[15]）。在整个强化学习的算法发展历史上，优化的思想贯穿始终，发挥了至关重要的作用。所以说，**强化学习的算法可以看作是在特定任务情景下的优化算法**。由此，最优化理论对强化学习的重要性可见一斑。

第 6 章 基于 HMM 模型的非线性共轭梯度法

这一部分要介绍优化算法在预测任务情景下的应用。HMM 模型是预测领域一类重要的模型^[18]，通过设计转移概率公式可以很好的刻画问题情景。但是随着参数的增多，传统的极大似然估计方法不能使用，需要通过优化算法来近似求解。本研究就是使用共轭梯度法最优化一类 HMM 模型的参数。

6.1 问题背景

近年来直播行业非常火爆，主播在直播时通过各种才艺表演，获得关注关注、点赞、打赏。在直播情境下，随着主播对顾客施加的刺激以及刺激强度的不同，顾客在观看过程中的内在状态也会发生不同的变化。依据其所处的内在状态表现出相应的行为。

为探究直播中主播刺激对顾客内在状态和行为的影响机制，本研究建立了直播情景下顾客行为的隐状态动态优化模型，该模型将顾客在直播中的行为表现视为隐马尔可夫过程，顾客在直播中的行为是其所处隐状态的外在表现，认为顾客行为和状态的变化取决于前一时刻顾客的行为和主播在直播间中施加的刺激，并通过实际数据对模型中两类影响因素的参数进行拟合验证。

6.2 HMM 模型

隐马尔科夫模型（Hidden Markov Model，以下简称 HMM）是比较经典的机器学习模型，它在语言识别，自然语言处理，模式识别等领域得到广泛的应用。

对于 HMM 模型，首先我们假设 Q 是所有可能的隐藏状态的集合， V 是所有可能的观测状态的集合，即：

$$Q = \{q_1, q_2, \dots, q_N\}, V = \{v_1, v_2, \dots, v_M\}$$

其中， N 是可能的隐藏状态数， M 是所有的可能的观察状态数。

对于一个长度为 T 的序列， I 对应的状态序列， O 是对应的观察序列，即：

$$I = \{i_1, i_2, \dots, i_T\}, O = \{o_1, o_2, \dots, o_T\}$$

HMM 模型做了两个很重要的假设如下：

1) 齐次马尔科夫链假设。即任意时刻的隐藏状态只依赖于它前一个隐藏状态。当然这样假设有点极端，因为很多时候我们的某一个隐藏状态不仅仅只依赖于前一个隐藏状态，可能是前两个或者是前三个。但是这样假设的好处就是模型简单，便于求解。如果在时刻 t 的隐藏状态是 $i_t = q_i$ ，在时刻 $t+1$ 的隐藏状态是 $i_{t+1} = q_j$ ，则从时刻 t 到时刻 $t+1$ 的 HMM 状态转移概率 a_{ij} 可以表示为：

$$a_{ij} = P(i_{t+1} = q_j | i_t = q_i)$$

这样 a_{ij} 可以组成马尔科夫链的状态转移矩阵 A :

$$A = [a_{ij}]_{N \times N}$$

2) 观测独立性假设。即任意时刻的观察状态只仅仅依赖于当前时刻的隐藏状态，这也是一个为了简化模型的假设。如果在时刻 t 的隐藏状态是 $i_t = q_j$ ，而对应的观察状态为 $o_t = v_k$ ，则该时刻观察状态 v_k 在隐藏状态 q_j 下生成的概率 $b_j(k)$ 满足：

$$b_j(k) = P(o_t = v_k | i_t = q_j)$$

这样 $b_j(k)$ 可以组成观测状态生成的概率矩阵 B :

$$B = [b_j(k)]_{N \times M}$$

除此之外，我们需要一组在时刻 $t = 1$ 的隐藏状态概率分布 Π :

$$\Pi = [\pi(i)]_N \text{ 其中 } \pi(i) = P(i_1 = q_i)$$

一个 HMM 模型，可以由隐藏状态初始概率分布 Π ，状态转移概率矩阵 A 和观测状态概率矩阵 B 决定。 Π, A 决定状态序列， B 决定观测序列。因此，HMM 模型可以由一个三元组 λ 表示如下： $\lambda = (A, B, \Pi)$

6.3 模型建立

根据上述分析，我们用以下公式来表示该解析的 HMM 模型。首先是给定隐状态下表现为特定的行为状态的概率公式：

$$P(Y_i, S_{K_1}) = \frac{\exp(\mu_{(s_{K_1}, Y_i)} - \beta_{(s_{K_1}, Y_i)}^T Y_i)}{\sum_{j'=1}^{N_1} \exp(\mu_{(s_{K_1}, Y_{j'})} - \beta_{(s_{K_1}, Y_{j'})}^T Y_{j'})}$$

其次是隐状态之间的状态转移概率：

$$P_k(S_{K_2}, S_{K_1}) = \frac{\exp(\gamma_{s_{(K_2, k)}} - \alpha_{s_{(K_1, k)}}^T W_k)}{\sum_{j'=1}^{N_2} \exp(\gamma_{s_{(j', k)}} - \alpha_{s_{(K_1, k)}}^T W_k)}$$

其中 $\sigma = (\mu, \beta, \gamma, \alpha)$ 为模型参数。

模型的观测变量为一个三维向量： $Y_{i,t} = (X_{1,t}, X_{2,t}, X_{3,t})^T$ 。 i 为编号， t 为时间。

将上述数据转化为这样的形式： Y_j ，表示第 j 种行为向量。其中， $j = 1, 2, \dots, N_1, N_1 = n_1 * n_2 * n_3$ ， n_1, n_2, n_3 分别为行为向量第 1, 2, 3 维度上的离散值种数。

于是，可以定义观测矩阵 $B_k = (P_k(Y_i, Y_j))_{N_1 \times N_1}$ 。其中 $k = 1, 2, \dots, |W|$, W 为思维刺激向量。其转移概率公式如下：

$$\begin{aligned} P_k(Y_i, Y_j) &= \sum_{K_1, K_2}^{N_2} P(Y_i, S_{K_1}) P_k(S_{K_2}, S_{K_1}) P(Y_j, S_{K_2}) \\ &= \sum_{K_1, K_2}^{N_2} \frac{\exp(\mu_{(s_{K_1}, Y_i)} - \beta_{(s_{K_1}, Y_i)}^T Y_i)}{\sum_{j'=1}^{N_1} \exp(\mu_{(s_{K_1}, Y_{j'})} - \beta_{(s_{K_1}, Y_{j'})}^T Y_{j'})} \\ &\quad * \frac{\exp(\gamma_{s_{(K_2, k)}} - \alpha_{s_{(K_1, k)}}^T W_k)}{\sum_{j'=1}^{N_2} \exp(\gamma_{s_{(j', k)}} - \alpha_{s_{(K_1, k)}}^T W_k)} \\ &\quad * \frac{\exp(\mu_{(s_{K_2}, Y_j)} - \beta_{(s_{K_2}, Y_j)}^T Y_j)}{\sum_{j'=1}^{N_1} \exp(\mu_{(s_{K_2}, Y_{j'})} - \beta_{(s_{K_2}, Y_{j'})}^T Y_{j'})} \end{aligned}$$

虽然上述观测状态转移式只是基于一种刺激的 HMM 模型，缺乏普遍性。但是它有很好的理论性质，所以可以通过全概率公式在这基础上自然导出基于多种刺激的隐马尔科夫链状态转移矩阵。令 N_1 为观测状态个数， N_2 为隐藏状态个数。它们分别的状态转移概率公式为：

$$\begin{aligned} P_{k_1, k_2}(Y_1, Y_3) &= \sum_{j=1}^{N_1} P_{k_1}(Y_1, Y_j) P_{k_2}(Y_j, Y_3) \\ P_{k_1, k_2}(S_1, S_3) &= \sum_{j=1}^{N_2} P_{k_1}(S_1, S_j) P_{k_2}(S_j, S_3) \end{aligned}$$

上述公式为基于两个刺激 k_1, k_2 的状态转移概率公式。同时可通过归纳法进一步给出 k 种刺激的状态转移公式。

然后，我们使用一种最幼稚的方式的求解上述公式的参数。首先根据数据将对应的观测转移概率估计出来，利用估计值和理论值的均方误差来设计优化算法。即下列优化问题：

$$\begin{aligned} \min_{\sigma} \quad & \sum_{k=1}^{N_3} \|P_k(Y_i, Y_j|\sigma) - \hat{P}_k(Y_i, Y_j|\sigma)\|_F^2 \text{ (Frobenius norm)} \\ \sigma &= (\mu, \beta, \gamma, \alpha), \quad k = 1, 2, 3, 4. \\ \mu &= (\mu_{(s_i, y_j)})^T \in R^{N_1 \times N_2}, \quad \gamma = (\gamma_{(y_i, w_k)})^T \in R^{|W| \times N_2}, \\ \beta &= (\beta_{y_i}) \in R^{|Y| \times N_1 \times N_2}, \quad \beta_{y_i} = (\beta_{(s_j, y_k)})^T \\ \alpha &= (\alpha_k) \in R^{|W| \times N_2}, \quad \alpha_k = (\alpha_{(s_i, w_k)})^T \end{aligned}$$

其中： N_3 为刺激种类数，

$$\begin{aligned} \hat{P}_k(Y_t, Y_{t-1}|\sigma) &= \frac{1}{n} \sum_{i=1}^n \hat{P}_k(Y_{i,t}, Y_{i,t-1}|\sigma) \\ P_k(Y_t, Y_{t-1}|\sigma) &= \frac{1}{n} \sum_{i=1}^n P_k(Y_{i,t}, Y_{i,t-1}|\sigma) \end{aligned}$$

上述问题可以使用最优化相关理论和算法进行分析求解。在这里使用非线性共轭梯度法来求解这个问题。算法框架为

步 1. 取初始点 x_0 及终止参数 $\varepsilon \geq 0, d_0 = -g_0$. 令 $k = 0$.

步 2. 若 $\|g_k\| \leq \varepsilon$, 算法终止, 否则进入下一步.

步 3. 按照一定的步长规则 (FR, PR, CW, Dixon 规则) 选取步长 α_{k1}^1

步 4. 计算 $x_{k+1} = x_k + \alpha_k d_k, g_{k+1} = \nabla f(x_{k+1})$ 。

核心代码为:

非线性共轭梯度算法框架

```
def nonlinear_cg(self, sigma0, Y, W, Pij, searching,
    beta_para):
    print('-----BEGIN-----')
    start = time.time()
    x_old = sigma0
    g_old = self.gfun(x_old, Y, W, Pij)
    # print("l:", g_old)
    d_old = -g_old
    f_list = []
    for k in range(self.kmax):
        f_old = self.fun(x_old, Y, W, Pij)
        print("当前残差函数值:", f_old)
        if k % 100 == 0:
            print(f_old)
        f_list.append(f_old[0])
        g_old_norm = np.linalg.norm(g_old, 2)
        print('当前迭代第{}步, 当前精度值 {:.8f}'.format(k,
            g_old_norm))
        if g_old_norm < self.eps:
            break
        alpha = searching(x_old, Y, W, Pij, d_old, g_old,
            f_old)
        # 更新 x
        x_new = x_old + alpha * d_old
        # 计算当前迭代点的梯度
        g_new = self.gfun(x_new, Y, W, Pij)
        beta = beta_para(g_old, g_new)
        d_new = -g_new + beta * d_old

        #更新参数
        d_old = d_new
        g_old = g_new
```

```
x_old = x_new
end = time.time()
time_use = end - start
return x_new, f_list, k, time_use
```

6.4 问题求解

为简化问题，实验中只选取一种刺激进行实验。实验结果如下图。代码见附录（函数和梯度函数很长）。

在模型参数拟合过程中，精确值随着迭代次数的增加首先急速下降，随后在第 25 次迭代后逐渐趋于平稳，并且精确值处于较小的可接受水平。该结果表明，模型参数拟合效果较好，本模型在直播情境中较为适用，模型参数值可以较为合理地描述直播过程中主播施加的刺激对顾客状态和行为表现的影响及程度。

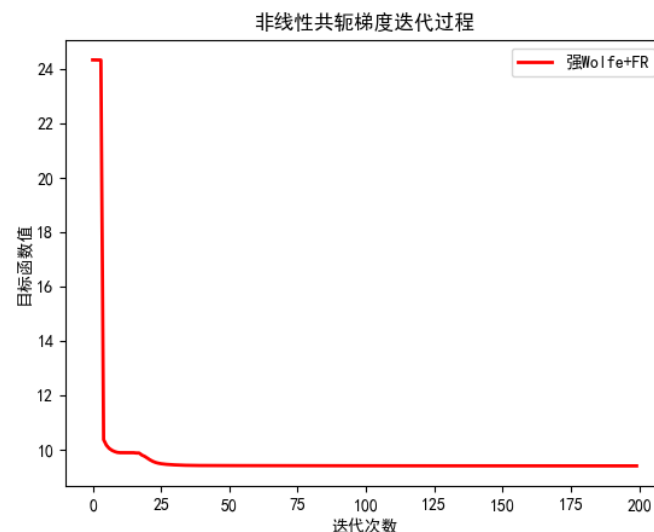


图 6.1 共轭梯度法算法效果图（车杆模型）

本模型具有较强的实际应用价值，不仅可以动态估计直播过程中各刺激对顾客状态动态演化的影响，还可以估计顾客在直播间的外在行为所处隐状态的概率分布，从而衡量和验证顾客隐状态划分的合理性和刺激因素影响的有效性，利于研究人员和主播从业者了解直播间内的顾客特点和情况，同时主播可根据本直播间刺激因素对顾客行为的影响效果制定直播策略，有助于主播根据直播效果目标和顾客行为优化直播内容，提升直播质量，提高直播情境下顾客关系管理的效率与科学性。

第 7 章 旅行商问题求解

7.1 旅行商问题

旅行商问题 (TravelingSalesmanProblem, TSP) 是一个经典的组合优化问题。经典的 TSP 可以描述为：一个商品推销员要去若干个城市推销商品，该推销员从一个城市出发，需要经过所有城市后，回到出发地。应如何选择行进路线，以使总的行程最短。

从图论的角度来看，该问题实质是在一个带权完全无向图中，找一个权值最小的 Hamilton 回路。由于该问题的可行解是所有顶点的全排列，随着顶点数的增加，会产生组合爆炸，它是一个 NP 完全问题。由于其在交通运输、电路板线路设计以及物流配送等领域内有着广泛的应用，国内外学者对其进行了大量的研究。早期的研究者使用精确算法求解该问题，常用的方法包括：分枝定界法、线性规划法、动态规划法等。但是，随着问题规模的增大，精确算法将变得无能为力，因此，在后来的研究中，国内外学者重点使用近似算法或启发式算法，主要有遗传算法、模拟退火法、蚁群算法、禁忌搜索算法、贪婪算法和神经网络等。^[19]

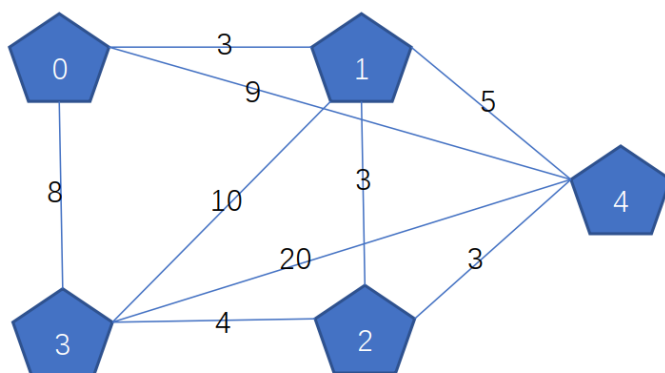


图 7.1 旅行商问题简单示例

TSP 的研究历史很久，最早的描述是 1759 年欧拉研究的骑士环游问题，即对于国际象棋棋盘中的 64 个方格，走访 64 个方格一次且仅一次，并且最终返回到起始点。1954 年，George Danzig 等人用线性规划的方法取得了旅行商问题的历史性的突破——解决了美国 49 个城市的巡回问题。这就是割平面法，这种方法在整数规划问题上也广泛应用。后来还提出了一种方法叫做分枝限界法，所谓限界，就是求出问题解的上、下界，通过当前得到的限界值排除一些次优解，为最终获得最优解提示方向。每次搜索下界最小的分枝，可以减小计算量。^[20]

最早的旅行商问题的数学规划是由 Dantzig (1959) 等人提出，并且是在最优化领域进行了深入研究。许多优化方法都用它作为一个测试基准。尽管问题在计算上很困难，但已经有了大量的启发式算法和精确方法来求解数量上万的实例，并且能将误差控制在 1% 内。^[21] 所以接下来我们就使用几种代表性的启发式算法来求解上述问题。

7.2 启发式算法解决旅行商问题 (TSP)

7.2.1 启发式算法

[22] 启发式算法 (*Heuristic Algorithm*) 可以这样定义：一个基于直观或经验构造的算法，在可接受的花费（指计算时间和空间）下给出待解决组合优化问题每一个实例的一个可行解，该可行解与最优解的偏离程度一般不能被预计。现阶段，启发式算法以仿自然体算法为主，主要有蚁群算法、模拟退火法、禁忌搜索法等。[23]

但是传统启发式算法存在一些问题：一是传统启发式算法依赖于算法的组织结构信息，通用性不高，二是容易在一些并不好的解之间陷入循环。

随着启发式算法的发展，出现了元启发式算法。元启发式算法大在传统启发式算法的思想增加了随机搜索的思想，并且不过分依赖于算法的组织结构信息。

启发式算法中的传统启发式算法和元启发式算法都不能保证获得全局最优解，但是元启发式算法由于增加了随机搜索的思想，反复执行可能收敛到全局最优解，并且元启发式算法不过分依赖算法组织结构信息，拥有更加通用启发式策略。

分类如下：

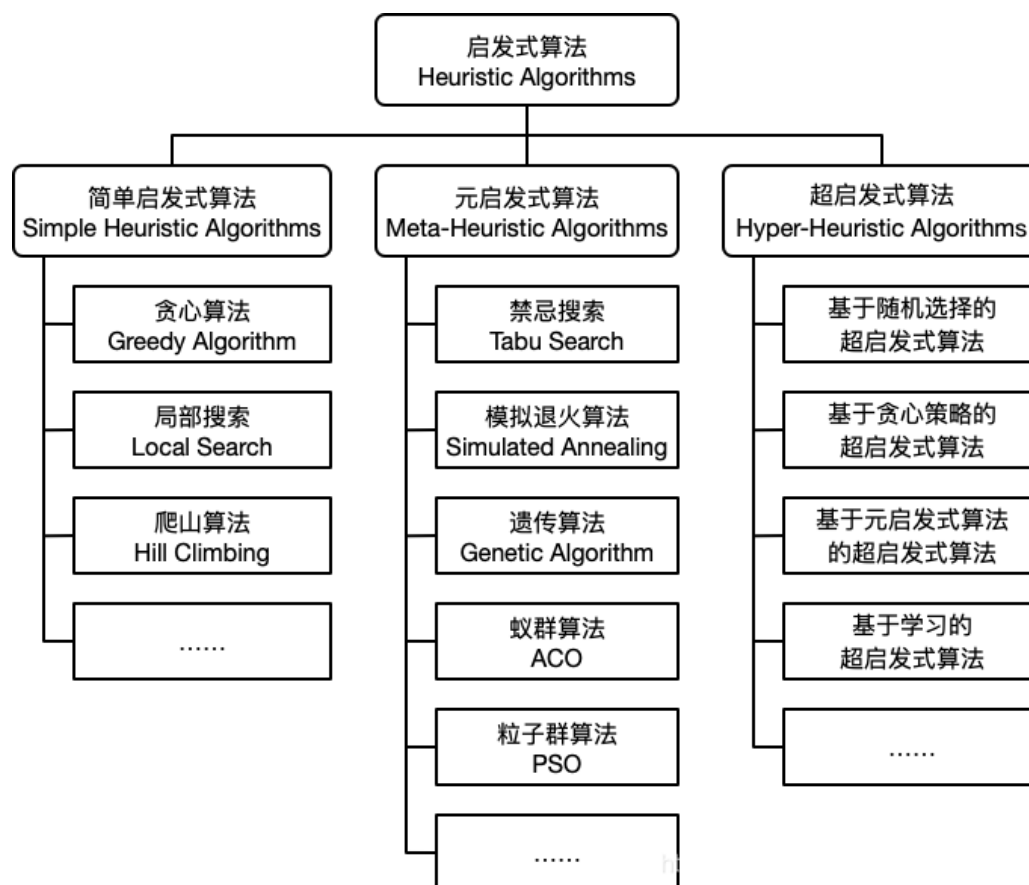


图 7.2 启发式算法分类

下面就四种启发式算法（模拟退火算法 (SA)，遗传算法 (GA)，蚁群算法 (ACO)，禁忌搜索算法 (TS)）简单介绍，然后使用这四种算法来求解旅行商问题。

(1) 模拟退火算法 (SA)

[24] 模拟退火算法思路来自于物理退火过程，在系统温度越高，退火现象指物体逐渐降温的物理现象，最终达到结晶状态，系统的能量状态最低。系统从高温降温过程中，随机选取可行解邻域内值，计算目标函数的增量值，如果系统能量下降，则接受该可行解，否则依照概率接受该解。在达到结束条件时结束循环。具体步骤如下：

- 1. 初始化温度和终止温度，最大迭代次数，可行解
- 2. 随机选取可行解的邻域值，计算系统能量增量，如果增量小于 0 则无条件接受该值；否则按照概率接受该值
- 3. 降低温度
- 4. 如果达到终止温度 / 最大迭代次数，则停止算法；否则跳回 2

从状态 1 到状态 2，*Metropolis* 接受准则：

$$p(1 \rightarrow 2) = \begin{cases} 1 & E(x_{new}) < E(x_{old}) \\ \exp\left(-\frac{E(x_{new}) - E(x_{old})}{T}\right) & E(x_{new}) \geq E(x_{old}) \end{cases}$$

模拟退火算法的难点在于求出产生新解的函数以及合理地调整参数提高算法的效率。

(2) 遗传算法 (GA)[25]

遗传算法的思路来自于遗传与进化理论，主要包含三个阶段：自然选择，基因重组，变异。每一条染色体对应一种解决方案，首先要寻找一种从表现型到基因型的数字化编码方式，即我们需要的表现到数字化的表示过程，然后再随机初始化种群/染色体群，为了选择出适应度强的个体，我们需要构造适应度函数。随后的过程就是：选择、交叉、变异，并且为了保证种群个数不变，在选择时按照适应度高低决定被保留的概率；交叉类似于基因重组，有单点交叉、多点交叉、均匀交叉，交叉点间之间的部分依照交叉概率进行交换，最终形成新的个体；变异即是染色体任意位置编码值以一定概率反转。此算法遵循的原则与达尔文进化论的原则一致：优胜劣汰，适者生存；通过基因的交叉匹配来扩大解的区域，持续优化问题；通过基因变异添加随机扰动，避免局部最优。

步骤：

- 1. 随机初始化种群
- 2. 评估每条染色体的适应度
- 3. 遵循适应度越高，被抽取概率越大的原则，从种群中选取个体作为父母本
- 4. 父母本染色体进行交叉，产生子代
- 5. 对自带染色体进行变异
- 6. 在未达到循环终止条件时，跳回 2

适应度函数用于评价某个个体的适应度，表现出群体中不同个体的好坏，一般总是非负的，由目标函数变换而来。选择函数用于抽取亲代中的个体，常用的选择算子有：轮盘赌选择，随即竞争选择，最佳保留选择。

(3) 蚁群算法 (ACO)

1991 年意大利米兰理学院 M. Dorigo 提出 Ant System^[26], 用于求解 TSP 等组合优化问题, 这是一种模拟蚂蚁觅食行为的优化算法。

蚂蚁在运动过程中, 能够在它所经过的路径上留下外激素, 而且蚂蚁在运动过程中能够感知外激素的存在及其强度, 并以此指导自己的运动方向, 蚂蚁倾向于朝着外激素强度高的方向移动. 由大量蚂蚁组成的蚁群的集体行为便表现出一种信息正反馈现象: 某一路径上走过的蚂蚁越多, 则后来者选择该路径的概率就越大. 蚂蚁个体之间就是通过这种信息的交流达到搜索食物的目的。

蚁群算法计算过程如下:

算法原理: 假如蚁群中所有蚂蚁的数量为 m , 所有城市之间的信息素用矩阵 *Pheromone* 表示, 最短路径为 *BestLength*, 最佳路径为 *BestTour*. 每只蚂蚁都有自己的内存, 内存中用一个禁忌表 *Tabu* 来存储该蚂蚁已经访问过的城市, 表示其在以后的搜索中将不能访问这些城市; 还有用另外一个允许访问的城市表 *Allowed* 来存储它还可以访问的城市; 另外还用了一个矩阵 Δ 来存储它在一个循环 (或者迭代) 中给所经过的路径释放的信息素; 还有另外一些数据, 例如一些控制参数 (α, β, ρ, Q), 该蚂蚁行走玩全程的总成本或距离 *TourLength*, 等等. 假定算法总共运行 *MAX_GEN* 次, 运行时间为 t 。

- 1. 初始化。
- 2. 为每只蚂蚁选择下一个节点。
- 3. 更新信息素矩阵。
- 4. 检查终止条件
- 5. 输出最优值

如果达到最大代数 *MAX_GEN*, 算法终止, 转到第 5 步; 否则, 重新初始化所有的蚂蚁的 Δ 矩阵所有元素初始化为 0, *Tabu* 表清空, *Allowed* 表中加入所有的城市节点。随机选择它们的起始位置 (也可以人工指定)。在 *Tabu* 中加入起始节点, *Allowed* 中去掉该起始节点, 重复执行 2, 3, 4 步。

(4) 禁忌搜索算法 (TS)

禁忌搜索算法 TS (Tabu Search)^[27] 是由美国科罗拉多州大学的 Fred Glover 教授在 1986 年左右提出来的, 是一个用来跳出局部最优的搜寻方法。

禁忌搜索是一种亚启发式随机搜索算法, 它从一个初始可行解出发, 选择一系列的特定搜索方向 (移动) 作为试探, 选择实现让特定的目标函数值变化最多的移动。为了避免陷入局部最优解, TS 搜索中采用了一种灵活的“记忆”技术, 对已经进行的优化过程进行记录和选择, 指导下一步的搜索方向。

TS 是人工智能的一种体现, 是局部领域搜索的一种扩展。禁忌搜索是在领域搜索的基础上, 通过设置禁忌表来禁忌一些已经历的操作, 并利用藐视准则来奖励一些优良状态, 其中涉及邻域、禁忌表、禁忌长度、候选解、藐视准则等影响禁忌搜索算法性能的关键因素。迄今为止, TS 算法在组合优化等计算机领域取得了很大的成功, 近年来又在函数全局优化方面得到较多的研究, 并大有发展的趋势。

算法步骤如下：

- 1. 给以禁忌表 $H = \Phi$ ，并选定一个初始解 x_{now} ；
- 2. 满足停止规则时，停止计算，输出结果；否则，在 x_{now} 的邻域 $N(x_{now})$ 中选择不受禁忌的候选集 $Can_N(x_{now})$ ；在 $Can_N(x_{now})$ 中选一个评价价值最佳的解 x_{next} , $x_{now} = x_{next}$ ；更新历史记录 H ，保存 $f(x_{now})$ ，重复 2；
- 3. 在保存的众多 f 中，挑选最小（大）值作为解；

7.2.2 实例求解

我们使用四种算法来计算旅行商问题，分别为模拟退火算法 (SA)，遗传算法 (GA)，蚁群算法 (ACO)，禁忌搜索算法 (TS)。我们的问题是求解三十个城市的旅行商问题。从第一个城市出发，位置坐标以二维向量的形式给出，如 $[(41, 94), (37, 84), (54, 67), (25, 62), \dots]$ 。下面是最优路径的距离关于算法迭代次数变化的可视化结果即图 7.3（代码见附录）：

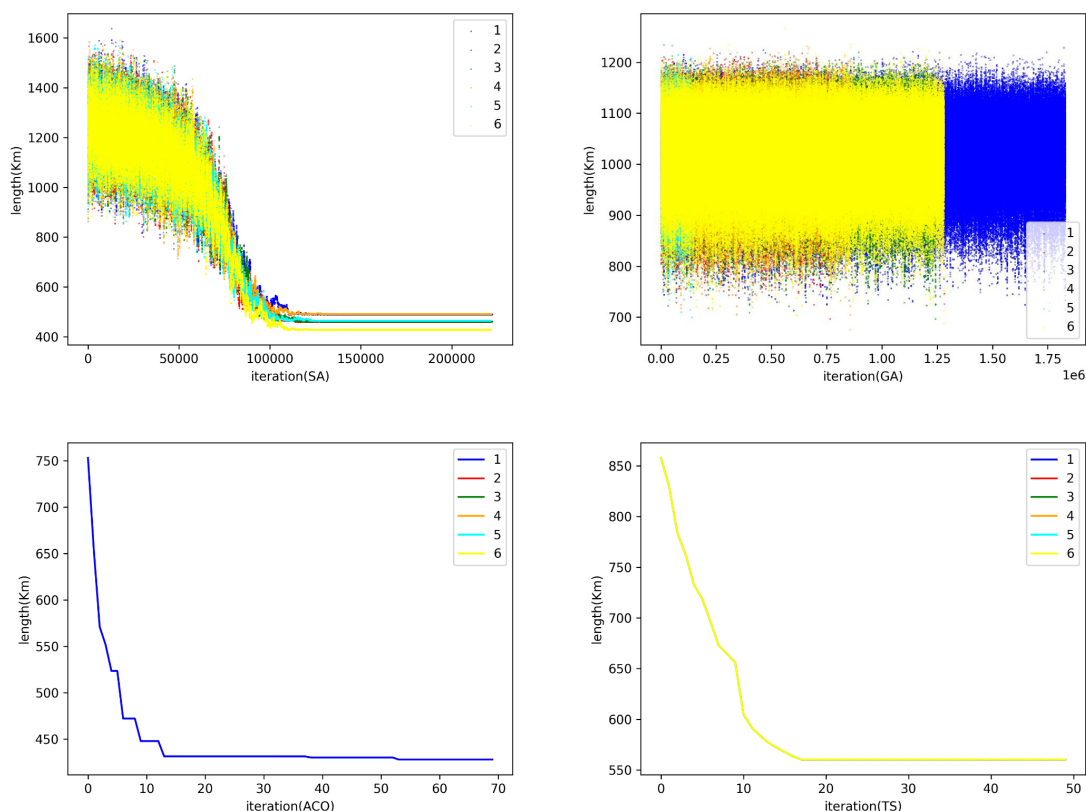


图 7.3 距离随迭代次数下降图像

从收敛速度上看，模拟退火算法和遗传算法在此问题中效果并不好，前者大约需要十万次迭代才能收敛；而后者完全不收敛，虽然有一些解可以达到和其他算法的解相差很少的程度，但这强烈依赖于终止条件，如果终止条件设置不好会导致算法效果大幅下降。后两种算法大幅度优于前两种算法，ACO 和 TS 分别只需要约 15 次和 20 次即可收敛。比前两种方法快了四个数量级。

从全局最优的角度上看，SA 算法和 ACO 算法大约可以达到 450km 的最优解。而其他两种算法分别在 700km 和 550km 左右，这说明 SA 和 ACO 算法在全局最优的角度上优于其他两种算法。

综上所述，考虑到当今时代对计算量的要求并不是很高，就算是使用 Intel-i7-8750H（五年前的老版本八代 CPU）版本的 CPU 进行计算，SA 算法和 ACO 算法相差的时间仅为几秒（分别为 5 秒左右和毫秒级别）；加之此问题并不需要即时的求解，对时间的要求并不高。所以我们认为用计算量换取更优的解是更值得的。因此我们将上述四种算法在旅行商问题上的性能从优到劣进行排序，分别为： $ACO > SA > TS > GA$ 。

参考文献

- [1] 陈宝林. 最优化理论与算法[M]. 清华大学出版社有限公司, 2005.
- [2] BOYD S, BOYD S P, VANDENBERGHE L. Convex optimization[M]. Cambridge university press, 2004.
- [3] 王宜举, 修乃华. 非线性最优化理论与方法[M]. 科学出版社, 2016.
- [4] 王周宏. 运筹学基础[M]. Qing hua da xue chu ban she, 2011.
- [5] SUTTON R S, BARTO A G. Reinforcement learning: An introduction[M]. MIT press, 2018.
- [6] GERAMIFARD A, WALSH T J, TELLEX S, et al. A tutorial on linear function approximators for dynamic programming and reinforcement learning[J]. Foundations and Trends® in Machine Learning, 2013, 6(4): 375-451.
- [7] SZEPESVÁRI C, LITTMAN M L. Generalized markov decision processes: Dynamic-programming and reinforcement-learning algorithms[C]. in: Proceedings of International Conference of Machine Learning: vol. 96. 1996.
- [8] OTTERLO M V, WIERING M. Reinforcement learning and markov decision processes[G]. in: Reinforcement learning. Springer, 2012: 3-42.
- [9] JAAKKOLA T, JORDAN M, SINGH S. Convergence of stochastic iterative dynamic programming algorithms[J]. Advances in neural information processing systems, 1993, 6.
- [10] RUMMERY G A, NIRANJAN M. On-line Q-learning using connectionist systems[M]. Citeseer, 1994.
- [11] MELO F S. Convergence of Q-learning: A simple proof[J]. Institute Of Systems and Robotics, Tech. Rep, 2001: 1-4.
- [12] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Playing atari with deep reinforcement learning[J]. ArXiv preprint arXiv:1312.5602, 2013.
- [13] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Human-level control through deep reinforcement learning[J]. Nature, 2015, 518(7540): 529-533.
- [14] SCHULMAN J, LEVINE S, ABBEEL P, et al. Trust region policy optimization[C]. in: International conference on machine learning. 2015: 1889-1897.
- [15] SCHULMAN J, WOLSKI F, DHARIWAL P, et al. Proximal policy optimization algorithms[J]. ArXiv preprint arXiv:1707.06347, 2017.
- [16] 王珏, 周志华, 周傲英. 机器学习及其应用[M]. 清华大学出版社有限公司, 2006.
- [17] 策略梯度理论证明[EB/OL]. 2018. <https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html>.
- [18] 马少辉, 谭慧, 代逸生. 客户关系动态优化模型与实证研究[J]. 运筹与管理, 2014, 23(4): 117-123.
- [19] 陈文兰, 戴树贵. 旅行商问题算法研究综述[J]. 滁州学院学报, 2006, 8(3): 1-6.
- [20] 马少平. 人工智能[Z].
- [21] FLOOD M M. The traveling-salesman problem[J]. Operations research, 1956, 4(1): 61-75.

- [22] BURKARD R E, DEINEKO V G, VAN DAL R, et al. Well-solvable special cases of the traveling salesman problem: a survey[J]. SIAM review, 1998, 40(3): 496-546.
- [23] 丛明煜, 王丽萍. 现代启发式算法理论研究[J]. 高技术通讯, 2003, 13(5): 105-110.
- [24] BERTSIMAS D, TSITSIKLIS J. Simulated annealing[J]. Statistical science, 1993, 8(1): 10-15.
- [25] MIRJALILI S. Genetic algorithm[G]. in: Evolutionary algorithms and neural networks. Springer, 2019: 43-55.
- [26] LI H, ZHANG M, ZENG C. Circular Jaccard distance based multi-solution optimization for traveling salesman problems[J]. Mathematical Biosciences and Engineering, 2022, 19(5): 4458-4480.
- [27] HE P, HAO J K. Hybrid search with neighborhood reduction for the multiple traveling salesman problem[J]. Computers & Operations Research, 2022: 105726.

附录 A 附录代码

A.1 SARSA 算法 *pytorch* 代码

```
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm # tqdm 是显示循环进度条的库

class CliffWalkingEnv:
    def __init__(self, ncol, nrow):
        self.nrow = nrow
        self.ncol = ncol
        self.x = 0 # 记录当前智能体位置的横坐标
        self.y = self.nrow - 1 # 记录当前智能体位置的纵坐标

    def step(self, action): # 外部调用这个函数来改变当前位置
        # 4种动作, change[0]:上, change[1]:下, change[2]:左,
        # change[3]:右。坐标系原点(0,0)
        # 定义在左上角
        change = [[0, -1], [0, 1], [-1, 0], [1, 0]]
        self.x = min(self.ncol - 1, max(0, self.x + change[
            action][0]))
        self.y = min(self.nrow - 1, max(0, self.y + change[
            action][1]))
        next_state = self.y * self.ncol + self.x
        reward = -1
        done = False
        if self.y == self.nrow - 1 and self.x > 0: # 下一个位
            置在悬崖或者目标
            done = True
            if self.x != self.ncol - 1:
                reward = -100
        return next_state, reward, done
```

```
def reset(self): # 回归初始状态,坐标轴原点在左上角
    self.x = 0
    self.y = self.nrow - 1
    return self.y * self.ncol + self.x

class Sarsa:
    """ Sarsa 算法 """
    def __init__(self, ncol, nrow, epsilon, alpha, gamma,
                 n_action=4):
        self.Q_table = np.zeros([nrow * ncol, n_action]) # 初始化  $Q(s, a)$  表格
        self.n_action = n_action # 动作个数
        self.alpha = alpha # 学习率
        self.gamma = gamma # 折扣因子
        self.epsilon = epsilon #  $\epsilon$ -贪婪策略中的参数

    def take_action(self, state): # 选取下一步的操作,具体实现为  $\epsilon$ -贪婪
        if np.random.random() < self.epsilon:
            action = np.random.randint(self.n_action)
        else:
            action = np.argmax(self.Q_table[state])
        return action

    def best_action(self, state): # 用于打印策略
        Q_max = np.max(self.Q_table[state])
        a = [0 for _ in range(self.n_action)]
        for i in range(self.n_action): # 若两个动作的价值一样,都会记录下来
            if self.Q_table[state, i] == Q_max:
                a[i] = 1
        return a

    def update(self, s0, a0, r, s1, a1):
        td_error = r + self.gamma * self.Q_table[s1, a1] - self.Q_table[s0, a0]
        self.Q_table[s0, a0] += self.alpha * td_error
```

```
ncol = 12
nrow = 4
env = CliffWalkingEnv(ncol, nrow)
np.random.seed(0)
epsilon = 0.1
alpha = 0.1
gamma = 0.9
agent = Sarsa(ncol, nrow, epsilon, alpha, gamma)
num_episodes = 500 # 智能体在环境中运行的序列的数量

return_list = [] # 记录每一条序列的回报
for i in range(10): # 显示10个进度条
    # tqdm的进度条功能
    with tqdm(total=int(num_episodes / 10), desc='Iteration %d' % i) as pbar:
        for i_episode in range(int(num_episodes / 10)): # 每个
            进度条的序列数
            episode_return = 0
            state = env.reset()
            action = agent.take_action(state)
            done = False
            while not done:
                next_state, reward, done = env.step(action)
                next_action = agent.take_action(next_state)
                episode_return += reward # 这里回报的计算不进
                    行折扣因子衰减
                agent.update(state, action, reward, next_state,
                    next_action)
                state = next_state
                action = next_action
            return_list.append(episode_return)
        if (i_episode + 1) % 10 == 0: # 每10条序列打印一下
            这10条序列的平均回报
            pbar.set_postfix({
                'episode':
                '%d' % (num_episodes / 10 * i + i_episode +
```

```
        1),  
        'return':  
        '%.3f' % np.mean(return_list[-10:])  
    })  
    pbar.update(1)  
  
episodes_list = list(range(len(return_list)))  
plt.plot(episodes_list, return_list)  
plt.xlabel('Episodes')  
plt.ylabel('Returns')  
plt.title('Sarsa on {}'.format('Cliff Walking'))  
plt.show()
```

A.2 Q-learning 算法 *pytorch* 代码

```
class QLearning:  
    """ Q-learning 算法 """  
    def __init__(self, ncol, nrow, epsilon, alpha, gamma,  
        n_action=4):  
        self.Q_table = np.zeros([nrow * ncol, n_action]) # 初  
            始化  $Q(s, a)$  表格  
        self.n_action = n_action # 动作个数  
        self.alpha = alpha # 学习率  
        self.gamma = gamma # 折扣因子  
        self.epsilon = epsilon # epsilon-贪婪策略中的参数  
  
    def take_action(self, state): # 选取下一步的操作  
        if np.random.random() < self.epsilon:  
            action = np.random.randint(self.n_action)  
        else:  
            action = np.argmax(self.Q_table[state])  
        return action  
  
    def best_action(self, state): # 用于打印策略  
        Q_max = np.max(self.Q_table[state])  
        a = [0 for _ in range(self.n_action)]
```

```
        for i in range(self.n_action):
            if self.Q_table[state, i] == Q_max:
                a[i] = 1
        return a

    def update(self, s0, a0, r, s1):
        td_error = r + self.gamma * self.Q_table[s1].max(
        ) - self.Q_table[s0, a0]
        self.Q_table[s0, a0] += self.alpha * td_error

np.random.seed(0)
epsilon = 0.1
alpha = 0.1
gamma = 0.9
agent = QLearning(ncol, nrow, epsilon, alpha, gamma)
num_episodes = 500 # 智能体在环境中运行的序列的数量

return_list = [] # 记录每一条序列的回报
for i in range(10): # 显示10个进度条
    # tqdm的进度条功能
    with tqdm(total=int(num_episodes / 10), desc='Iteration %d'
    % i) as pbar:
        for i_episode in range(int(num_episodes / 10)): # 每个
            进度条的序列数
            episode_return = 0
            state = env.reset()
            done = False
            while not done:
                action = agent.take_action(state)
                next_state, reward, done = env.step(action)
                episode_return += reward # 这里回报的计算不进
                    行折扣因子衰减
                agent.update(state, action, reward, next_state)
                state = next_state
            return_list.append(episode_return)
        if (i_episode + 1) % 10 == 0: # 每10条序列打印一下
```


这10条序列的平均回报

```
pbar.set_postfix({
    'episode':
        '%d' % (num_episodes / 10 * i + i_episode +
                1),
    'return':
        '%.3f' % np.mean(return_list[-10:])
})
pbar.update(1)

episodes_list = list(range(len(return_list)))
plt.plot(episodes_list, return_list)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('Q-learning on {}'.format('Cliff Walking'))
plt.show()

action_meaning = ['^', 'v', '<', '>']
print('Q-learning 算法最终收敛得到的策略为：')
print_agent(agent, env, action_meaning, list(range(37, 47)),
            [47])
```

A.3 DQN 算法 *pytorch* 代码

```
import random
import gym
import numpy as np
import collections
from tqdm import tqdm
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
import rl_utils

class ReplayBuffer:
    ''' 经验回放池 '''
```

```
def __init__(self, capacity):
    self.buffer = collections.deque(maxlen=capacity)  # 队列, 先进先出

def add(self, state, action, reward, next_state, done):  # 将数据加入 buffer
    self.buffer.append((state, action, reward, next_state, done))

def sample(self, batch_size):  # 从 buffer 中采样数据, 数量为 batch_size
    transitions = random.sample(self.buffer, batch_size)
    state, action, reward, next_state, done = zip(*transitions)
    return np.array(state), action, reward, np.array(next_state), done

def size(self):  # 目前 buffer 中数据的数量
    return len(self.buffer)

class Qnet(torch.nn.Module):
    ''' 只有一层隐藏层的Q网络 '''
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(Qnet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))  # 隐藏层使用 ReLU 激活函数
        return self.fc2(x)

class DQN:
    ''' DQN 算法 '''
    def __init__(self, state_dim, hidden_dim, action_dim, learning_rate, gamma, epsilon, target_update, device):
        self.action_dim = action_dim
```

```
self.q_net = Qnet(state_dim, hidden_dim,
                  self.action_dim).to(device) # Q网络

# 目标网络
self.target_q_net = Qnet(state_dim, hidden_dim,
                          self.action_dim).to(device)

# 使用Adam优化器
self.optimizer = torch.optim.Adam(self.q_net.parameters()
                                   ,
                                   lr=learning_rate)

self.gamma = gamma # 折扣因子
self.epsilon = epsilon # epsilon-贪婪策略
self.target_update = target_update # 目标网络更新频率
self.count = 0 # 计数器,记录更新次数
self.device = device

def take_action(self, state): # epsilon-贪婪策略采取动作
    if np.random.random() < self.epsilon:
        action = np.random.randint(self.action_dim)
    else:
        state = torch.tensor([state], dtype=torch.float).to(
            self.device)
        action = self.q_net(state).argmax().item()
    return action

def update(self, transition_dict):
    states = torch.tensor(transition_dict['states'],
                          dtype=torch.float).to(self.device)

    actions = torch.tensor(transition_dict['actions']).view(
        (-1, 1)).to(
            self.device)

    rewards = torch.tensor(transition_dict['rewards'],
                          dtype=torch.float).view(-1, 1).
        to(self.device)

    next_states = torch.tensor(transition_dict['next_states'],
                              dtype=torch.float).to(self.
```

```
device)

dones = torch.tensor(transition_dict['dones'],
                      dtype=torch.float).view(-1, 1).to(
                      self.device)

q_values = self.q_net(states).gather(1, actions) # Q值
# 下个状态的最大Q值
max_next_q_values = self.target_q_net(next_states).max
(1)[0].view(
    -1, 1)
q_targets = rewards + self.gamma * max_next_q_values *
(1 - dones
)
```


TD
误差
目标

```
dqn_loss = torch.mean(F.mse_loss(q_values, q_targets))
# 均方误差损失函数
self.optimizer.zero_grad() # PyTorch中默认梯度会累积,
# 这里需要显式将梯度置为0
dqn_loss.backward() # 反向传播更新参数
self.optimizer.step()

if self.count % self.target_update == 0:
    self.target_q_net.load_state_dict(
        self.q_net.state_dict()) # 更新目标网络
self.count += 1
```

```
lr = 2e-3
num_episodes = 500
```

```
hidden_dim = 128
gamma = 0.98
epsilon = 0.01
target_update = 10
buffer_size = 10000
minimal_size = 500
batch_size = 64
device = torch.device("cuda") if torch.cuda.is_available() else
    torch.device(
        "cpu")

env_name = 'CartPole-v0'
env = gym.make(env_name)
random.seed(0)
np.random.seed(0)
env.seed(0)
torch.manual_seed(0)
replay_buffer = ReplayBuffer(buffer_size)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = DQN(state_dim, hidden_dim, action_dim, lr, gamma,
    epsilon,
        target_update, device)

return_list = []
for i in range(10):
    with tqdm(total=int(num_episodes / 10), desc='Iteration %d'
        % i) as pbar:
        for i_episode in range(int(num_episodes / 10)):
            episode_return = 0
            state = env.reset()
            done = False
            while not done:
                action = agent.take_action(state)
                next_state, reward, done, _ = env.step(action)
                replay_buffer.add(state, action, reward,
                    next_state, done)
```

```
state = next_state
episode_return += reward
# 当 buffer 数据的数量超过一定值后, 才进行 Q 网络训练
if replay_buffer.size() > minimal_size:
    b_s, b_a, b_r, b_ns, b_d = replay_buffer.
        sample(batch_size)
    transition_dict = {
        'states': b_s,
        'actions': b_a,
        'next_states': b_ns,
        'rewards': b_r,
        'done': b_d
    }
    agent.update(transition_dict)
return_list.append(episode_return)
if (i_episode + 1) % 10 == 0:
    pbar.set_postfix({
        'episode':
            '%d' % (num_episodes / 10 * i + i_episode +
                1),
        'return':
            '%.3f' % np.mean(return_list[-10:])
    })
pbar.update(1)
```

A.4 Actor-Critic 算法 *pytorch* 代码

```
import gym
import torch
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
import rl_utils

class PolicyNet(torch.nn.Module):
```



```
def __init__(self, state_dim, hidden_dim, action_dim):
    super(PolicyNet, self).__init__()
    self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
    self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

def forward(self, x):
    x = F.relu(self.fc1(x))
    return F.softmax(self.fc2(x), dim=1)

class ValueNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim):
        super(ValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)

class ActorCritic:
    def __init__(self, state_dim, hidden_dim, action_dim,
        actor_lr, critic_lr,
            gamma, device):
        # 策略网络
        self.actor = PolicyNet(state_dim, hidden_dim,
            action_dim).to(device)
        self.critic = ValueNet(state_dim, hidden_dim).to(device)
        # 价值网络
        # 策略网络优化器
        self.actor_optimizer = torch.optim.Adam(self.actor.
            parameters(),
                lr=actor_lr)
        self.critic_optimizer = torch.optim.Adam(self.critic.
            parameters(),
                lr=critic_lr)
        # 价值网络
        # 优化器
```

```
self.gamma = gamma
self.device = device

def take_action(self, state):
    state = torch.tensor([state], dtype=torch.float).to(
        self.device)
    probs = self.actor(state)
    action_dist = torch.distributions.Categorical(probs)
    action = action_dist.sample()
    return action.item()

def update(self, transition_dict):
    states = torch.tensor(transition_dict['states'],
                           dtype=torch.float).to(self.device)
    actions = torch.tensor(transition_dict['actions']).view(
        (-1, 1)).to(
            self.device)
    rewards = torch.tensor(transition_dict['rewards'],
                           dtype=torch.float).view(-1, 1).
        to(self.device)
    next_states = torch.tensor(transition_dict['next_states'],
                               dtype=torch.float).to(self.
                                                       device)
    dones = torch.tensor(transition_dict['dones'],
                         dtype=torch.float).view(-1, 1).to(
                            self.device)

    # 时序差分目标
    td_target = rewards + self.gamma * self.critic(
        next_states) * (1 -

    td_delta = td_target - self.critic(states) # 时序差分
    误差
```

```
log_probs = torch.log(self.actor(states).gather(1,
    actions))
actor_loss = torch.mean(-log_probs * td_delta.detach())
# 均方误差损失函数
critic_loss = torch.mean(
    F.mse_loss(self.critic(states), td_target.detach())
)
self.actor_optimizer.zero_grad()
self.critic_optimizer.zero_grad()
actor_loss.backward() # 计算策略网络的梯度
critic_loss.backward() # 计算价值网络的梯度
self.actor_optimizer.step() # 更新策略网络的参数
self.critic_optimizer.step() # 更新价值网络的参数

actor_lr = 1e-3
critic_lr = 1e-2
num_episodes = 1000
hidden_dim = 128
gamma = 0.98
device = torch.device("cuda") if torch.cuda.is_available() else
    torch.device(
        "cpu")

env_name = 'CartPole-v0'
env = gym.make(env_name)
env.seed(0)
torch.manual_seed(0)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = ActorCritic(state_dim, hidden_dim, action_dim, actor_lr
    , critic_lr,
                    gamma, device)

return_list = rl_utils.train_on_policy_agent(env, agent,
    num_episodes)

episodes_list = list(range(len(return_list)))
```

```
plt.plot(episodes_list, return_list)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('Actor-Critic on {}'.format(env_name))
plt.show()

mv_return = rl_utils.moving_average(return_list, 9)
plt.plot(episodes_list, mv_return)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('Actor-Critic on {}'.format(env_name))
plt.show()
```

A.5 TRPO 算法 *pytorch* 代码

```
import torch
import numpy as np
import gym
import matplotlib.pyplot as plt
import torch.nn.functional as F
import rl_utils
import copy

def compute_advantage(gamma, lambda, td_delta):
    td_delta = td_delta.detach().numpy()
    advantage_list = []
    advantage = 0.0
    for delta in td_delta[::-1]:
        advantage = gamma * lambda * advantage + delta
        advantage_list.append(advantage)
    advantage_list.reverse()
    return torch.tensor(advantage_list, dtype=torch.float)

class PolicyNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(PolicyNet, self).__init__()
```

```
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return F.softmax(self.fc2(x), dim=1)

class ValueNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim):
        super(ValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)

class TRPO:
    """ TRPO 算法 """
    def __init__(self, hidden_dim, state_space, action_space,
                 lambda,
                 kl_constraint, alpha, critic_lr, gamma, device
                 ):
        state_dim = state_space.shape[0]
        action_dim = action_space.n
        # 策略网络参数不需要优化器更新
        self.actor = PolicyNet(state_dim, hidden_dim,
                               action_dim).to(device)
        self.critic = ValueNet(state_dim, hidden_dim).to(device)
        self.critic_optimizer = torch.optim.Adam(self.critic.
                                                  parameters(),
                                                  lr=critic_lr)

        self.gamma = gamma
        self.lambda = lambda # GAE 参数
```

```
self.kl_constraint = kl_constraint # KL距离最大限制
self.alpha = alpha # 线性搜索参数
self.device = device

def take_action(self, state):
    state = torch.tensor([state], dtype=torch.float).to(
        self.device)
    probs = self.actor(state)
    action_dist = torch.distributions.Categorical(probs)
    action = action_dist.sample()
    return action.item()

def hessian_matrix_vector_product(self, states,
    old_action_dists, vector):
    # 计算黑塞矩阵和一个向量的乘积
    new_action_dists = torch.distributions.Categorical(self
        .actor(states))
    kl = torch.mean(
        torch.distributions.kl.kl_divergence(
            old_action_dists,
                                                    new_action_dists
        )) # 计算
        平均KL距离

    kl_grad = torch.autograd.grad(kl,
                                    self.actor.parameters(),
                                    create_graph=True)
    kl_grad_vector = torch.cat([grad.view(-1) for grad in
        kl_grad])
    # KL距离的梯度先和向量进行点积运算
    kl_grad_vector_product = torch.dot(kl_grad_vector,
        vector)
    grad2 = torch.autograd.grad(kl_grad_vector_product,
        self.actor.parameters())
    grad2_vector = torch.cat([grad.view(-1) for grad in
        grad2])
    return grad2_vector
```



```
def conjugate_gradient(self, grad, states, old_action_dists
): # 共轭梯度法求解方程
    x = torch.zeros_like(grad)
    r = grad.clone()
    p = grad.clone()
    rdotr = torch.dot(r, r)
    for i in range(10): # 共轭梯度主循环
        Hp = self.hessian_matrix_vector_product(states,
                                                old_action_dists,
                                                p)

        alpha = rdotr / torch.dot(p, Hp)
        x += alpha * p
        r -= alpha * Hp
        new_rdotr = torch.dot(r, r)
        if new_rdotr < 1e-10:
            break
        beta = new_rdotr / rdotr
        p = r + beta * p
        rdotr = new_rdotr
    return x

def compute_surrogate_obj(self, states, actions, advantage,
old_log_probs,
                        actor): # 计算策略目标
    log_probs = torch.log(actor(states).gather(1, actions))
    ratio = torch.exp(log_probs - old_log_probs)
    return torch.mean(ratio * advantage)

def line_search(self, states, actions, advantage,
old_log_probs,
                old_action_dists, max_vec): # 线性搜索
    old_para = torch.nn.utils.convert_parameters.
parameters_to_vector(
    self.actor.parameters())
    old_obj = self.compute_surrogate_obj(states, actions,
advantage,
                                        old_log_probs,
```

```
self.actor)

for i in range(15): # 线性搜索主循环
    coef = self.alpha**i
    new_para = old_para + coef * max_vec
    new_actor = copy.deepcopy(self.actor)
    torch.nn.utils.convert_parameters.
        vector_to_parameters(
            new_para, new_actor.parameters())
    new_action_dists = torch.distributions.Categorical(
        new_actor(states))
    kl_div = torch.mean(
        torch.distributions.kl.kl_divergence(
            old_action_dists,
            new_action_dists
        ))
    new_obj = self.compute_surrogate_obj(states,
        actions, advantage,
        old_log_probs,
        new_actor)

    if new_obj > old_obj and kl_div < self.
        kl_constraint:
            return new_para
    return old_para

def policy_learn(self, states, actions, old_action_dists,
    old_log_probs,
        advantage): # 更新策略函数
    surrogate_obj = self.compute_surrogate_obj(states,
        actions, advantage,
        old_log_probs,
        self.
        actor)
    grads = torch.autograd.grad(surrogate_obj, self.actor.
        parameters())
    obj_grad = torch.cat([grad.view(-1) for grad in grads])
        .detach()
    # 用共轭梯度法计算  $x = H^{-1}g$ 
```

```
descent_direction = self.conjugate_gradient(obj_grad,
                                             states,
                                             old_action_dists
                                             )

Hd = self.hessian_matrix_vector_product(states,
                                         old_action_dists,
                                         descent_direction
                                         )

max_coef = torch.sqrt(2 * self.kl_constraint /
                      (torch.dot(descent_direction, Hd)
                       + 1e-8))

new_para = self.line_search(states, actions, advantage,
                             old_log_probs,
                             old_action_dists,
                             descent_direction *
                             max_coef) # 线性搜索

torch.nn.utils.convert_parameters.vector_to_parameters(
    new_para, self.actor.parameters()) # 用线性搜索后
                                     的参数更新策略

def update(self, transition_dict):
    states = torch.tensor(transition_dict['states'],
                          dtype=torch.float).to(self.device)

    actions = torch.tensor(transition_dict['actions']).view
        (-1, 1).to(
            self.device)

    rewards = torch.tensor(transition_dict['rewards'],
                          dtype=torch.float).view(-1, 1).
        to(self.device)

    next_states = torch.tensor(transition_dict['next_states'],
                              dtype=torch.float).to(self.
                              device)

    dones = torch.tensor(transition_dict['dones'],
                        dtype=torch.float).view(-1, 1).to(
```

```
self.device)

td_target = rewards + self.gamma * self.critic(
    next_states) * (1 -

td_delta = td_target - self.critic(states)
advantage = compute_advantage(self.gamma, self.lmbda,
                               td_delta.cpu()).to(self.
                               device)
old_log_probs = torch.log(self.actor(states).gather(1,
                                                    actions
                                                    )
                           )
                           .
                           detach
                           ()

old_action_dists = torch.distributions.Categorical(
    self.actor(states).detach())
critic_loss = torch.mean(
    F.mse_loss(self.critic(states), td_target.detach())
)
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()  # 更新价值函数
# 更新策略函数
self.policy_learn(states, actions, old_action_dists,
                  old_log_probs,
                  advantage)

num_episodes = 500
hidden_dim = 128
gamma = 0.98
lmbda = 0.95
critic_lr = 1e-2
kl_constraint = 0.0005
```

```
alpha = 0.5
device = torch.device("cuda") if torch.cuda.is_available() else
    torch.device(
        "cpu")

env_name = 'CartPole-v0'
env = gym.make(env_name)
env.seed(0)
torch.manual_seed(0)
agent = TRPO(hidden_dim, env.observation_space, env.
    action_space, lambda,
                kl_constraint, alpha, critic_lr, gamma, device)
return_list = rl_utils.train_on_policy_agent(env, agent,
    num_episodes)

episodes_list = list(range(len(return_list)))
plt.plot(episodes_list, return_list)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('TRPO on {}'.format(env_name))
plt.show()

mv_return = rl_utils.moving_average(return_list, 9)
plt.plot(episodes_list, mv_return)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('TRPO on {}'.format(env_name))
plt.show()
```

A.6 PPO 算法 *pytorch* 代码

```
import gym
import torch
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
```

```
import rl_utils

class PolicyNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(PolicyNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return F.softmax(self.fc2(x), dim=1)

class ValueNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim):
        super(ValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)

class PPO:
    ''' PPO算法,采用截断方式 '''
    def __init__(self, state_dim, hidden_dim, action_dim,
        actor_lr, critic_lr,
            lambda, epochs, eps, gamma, device):
        self.actor = PolicyNet(state_dim, hidden_dim,
            action_dim).to(device)
        self.critic = ValueNet(state_dim, hidden_dim).to(device)
        self.actor_optimizer = torch.optim.Adam(self.actor.
            parameters(),
                lr=actor_lr)
```

```
self.critic_optimizer = torch.optim.Adam(self.critic.parameters(),
                                          lr=critic_lr)

self.gamma = gamma
self.lmbda = lmbda
self.epochs = epochs # 一条序列的数据用来训练轮数
self.eps = eps # PPO中截断范围的参数
self.device = device

def take_action(self, state):
    state = torch.tensor([state], dtype=torch.float).to(
        self.device)
    probs = self.actor(state)
    action_dist = torch.distributions.Categorical(probs)
    action = action_dist.sample()
    return action.item()

def update(self, transition_dict):
    states = torch.tensor(transition_dict['states'],
                          dtype=torch.float).to(self.device)

    actions = torch.tensor(transition_dict['actions']).view(
        (-1, 1)).to(
            self.device)
    rewards = torch.tensor(transition_dict['rewards'],
                          dtype=torch.float).view(-1, 1).
        to(self.device)
    next_states = torch.tensor(transition_dict['next_states'],
                              dtype=torch.float).to(self.device)

    dones = torch.tensor(transition_dict['dones'],
                        dtype=torch.float).view(-1, 1).to(
                            self.device)
    td_target = rewards + self.gamma * self.critic(
        next_states) * (1 -
```



```
td_delta = td_target - self.critic(states)
advantage = rl_utils.compute_advantage(self.gamma, self
    .lmbda,
                                td_delta.cpu()).
                                to(self.
                                device)
old_log_probs = torch.log(self.actor(states).gather(1,
                                                    actions
                                                    )
                                                    )
                                                    .
                                                    detach
                                                    ()

for _ in range(self.epochs):
    log_probs = torch.log(self.actor(states).gather(1,
        actions))
    ratio = torch.exp(log_probs - old_log_probs)
    surr1 = ratio * advantage
    surr2 = torch.clamp(ratio, 1 - self.eps,
        1 + self.eps) * advantage # 截
                                   断
    actor_loss = torch.mean(-torch.min(surr1, surr2))
    # PPO损失函数
    critic_loss = torch.mean(
        F.mse_loss(self.critic(states), td_target.
            detach()))
    self.actor_optimizer.zero_grad()
    self.critic_optimizer.zero_grad()
    actor_loss.backward()
    critic_loss.backward()
    self.actor_optimizer.step()
    self.critic_optimizer.step()
```

```
actor_lr = 1e-3
critic_lr = 1e-2
num_episodes = 500
hidden_dim = 128
gamma = 0.98
lmbda = 0.95
epochs = 10
eps = 0.2
device = torch.device("cuda") if torch.cuda.is_available() else
    torch.device(
        "cpu")

env_name = 'CartPole-v0'
env = gym.make(env_name)
env.seed(0)
torch.manual_seed(0)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = PPO(state_dim, hidden_dim, action_dim, actor_lr,
            critic_lr, lmbda,
                epochs, eps, gamma, device)

return_list = rl_utils.train_on_policy_agent(env, agent,
        num_episodes)

episodes_list = list(range(len(return_list)))
plt.plot(episodes_list, return_list)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('PPO on {}'.format(env_name))
plt.show()

mv_return = rl_utils.moving_average(return_list, 9)
plt.plot(episodes_list, mv_return)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('PPO on {}'.format(env_name))
```

```
plt.show()
```

A.7 共轭梯度法解 HMM 模型 *python* 代码

```
import numpy as np
from matplotlib import pyplot as plt
import time
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

class NCG():
    def __init__(self):
        # 给定相关种类参数:
        self.N1 = 4 # 观测状态种数
        self.N2 = 4 # 隐状态种数
        self.Y_dim = 3
        self.W_dim = 4
        # 模型参数:
        self.sum_dim = self.N2*self.N1 + self.N2*self.N1*self.
            Y_dim + self.N2 + self.N2*self.W_dim
        self.sigma0 = np.random.normal(0, 1, (self.sum_dim, 1))
        # 初始参数服从 $N(0,1)$ 的正态分布
        #  $self.sigma0 = np.ones((self.sum_dim, 1)) * 0.21$ 
        # 参数分割:
        self.mu = self.sigma0[:self.N2 * self.N1, 0].reshape((
            self.N1, self.N2))
        beta = self.sigma0[self.N2 * self.N1:self.N2 * self.N1
            + self.N2 * self.N1 * self.Y_dim, 0]
        self.beta = beta.reshape((self.N1, self.N2, self.Y_dim)
            )
        N1 = self.N2 * self.N1 + self.N2 * self.N1 * self.Y_dim
        N2 = self.N2 * self.N1 + self.N2 * self.N1 * self.Y_dim
            + self.N2
        self.gamma = self.sigma0[N1:N2, 0].reshape((self.N2, 1)
            )
```

```

self.alpha = self.sigma0[N2:, 0].reshape((self.W_dim,
self.N2))
# self.Pij = np.random.uniform(0, 1, (self.N1, self.N1)
)
# 需要给定下面三个参数:
self.W = np.array([[31.167], [34.125], [26.667],
[27.271]])
Pij = pd.read_excel('./4 观测转换频率.xlsx', sheet_name=
"Sheet1", header=None)
self.Pij = Pij.values
print(self.Pij, self.Pij.shape)
Y_all = pd.read_excel('./4 观测转换频率 - 副本.xlsx',
sheet_name="Sheet1", header=None)
self.Y_all = Y_all.values
print(self.Y_all, self.Y_all.shape)
# self.Pij[0, 0] = 1
self.Pij = self.Pij / self.Pij.max(axis=1) # 行和置为1
# 算法参数:
self.kmax = 200
self.eps = 1e-6

#  $P(Y_i, Y_j)$  概率计算函数
def funPij(self, mu, beta, gamma, alpha, Yi, Yj, W, i, j):
    Pij = 0
    for K1 in range(self.N2):
        for K2 in range(self.N2):
            P_Sk1_Yi_1 = np.exp(mu[i, K1] - beta[i, K1, :].
reshape((1, self.Y_dim))@Yi)
            P_Sk1_Yi_2 = 0
            for j1 in range(self.N1):
                P_Sk1_Yi_2 = P_Sk1_Yi_2+np.exp(mu[j1, K1]-
beta[j1, K1, :].reshape((1,self.Y_dim))
@self.Y_all[j1,:])
            P_Sk1_Yi = P_Sk1_Yi_1 / P_Sk1_Yi_2

            P_Sk2_Sk1_1 = np.exp(gamma[K2, 0] - alpha[:, K1
].T@W)

```

```

        P_Sk2_Sk1_2 = 0
        for j1 in range(self.N2):
            P_Sk2_Sk1_2 = P_Sk2_Sk1_2 + np.exp(gamma[j1, 0] - alpha[:, K1].T @ W)
        P_Sk2_Sk1 = P_Sk2_Sk1_1 / P_Sk2_Sk1_2

        P_Sk2_Yi_1 = np.exp(mu[j, K2] - beta[j, K2, :].
            reshape((1, self.Y_dim)) @ Yj)
        P_Sk2_Yi_2 = 0
        for j1 in range(self.N1):
            P_Sk2_Yi_2 = P_Sk2_Yi_2 + np.exp(mu[j1, K2] -
                beta[j1, K2, :].reshape((1, self.Y_dim))
                @ self.Y_all[j1, :])
        P_Sk2_Yi = P_Sk2_Yi_1 / P_Sk2_Yi_2

        Pij = Pij + P_Sk1_Yi * P_Sk2_Sk1 * P_Sk2_Yi
    return Pij

def P_Y_S_fun(self, mu, beta, gamma, alpha):
    P_Y_S = np.zeros((self.N1, self.N2))
    for i in range(self.N1):
        for j in range(self.N2):
            P_Yi_Sj_1 = np.exp(mu[i, j] - beta[i, j, :].
                reshape((1, self.Y_dim)) @ self.Y_all[i, :])
            P_Yi_Sj_2 = 0
            for j1 in range(self.N1):
                P_Yi_Sj_2 += np.exp(mu[j1, j] - beta[j1, j, :].
                    reshape((1, self.Y_dim)) @ self.
                    Y_all[j1, :])
            P_Y_S[i, j] = P_Yi_Sj_1 / P_Yi_Sj_2
    return P_Y_S

# P(Yi, Yj) 梯度计算函数
def gfunPij(self, mu, beta, gamma, alpha, Yi, Yj, W, i, j):
    g_mu = mu - mu
    g_beta = beta - beta
    g_gamma = gamma - gamma

```

```
g_alpha = alpha - alpha

for K1 in range(self.N2):
    for K2 in range(self.N2):
        P1 = np.exp(mu[i, K1] - beta[i, K1, :].reshape(
            (1, self.Y_dim))@Yi)
        P2 = 0
        for j1 in range(self.N1):
            P2 = P2 + np.exp(mu[j1, K1] - beta[j1, K1,
                :].reshape((1, self.Y_dim)) @ self.Y_all
                [j1, :])
        P_Sk1_Yi = P1 * (P2-P1) / P2 / P2

        P_Sk2_Sk1_1 = np.exp(gamma[K2, 0] - alpha[:, K1
            ].T @ W)
        P_Sk2_Sk1_2 = 0
        for j1 in range(self.N2):
            P_Sk2_Sk1_2 = P_Sk2_Sk1_2 + np.exp(gamma[j1
                , 0] - alpha[:, K1].T @ W)
        P_Sk2_Sk1 = P_Sk2_Sk1_1 / P_Sk2_Sk1_2

        P_Sk2_Yj_1 = np.exp(mu[j, K2] - beta[j, K2, :].
            reshape((1, self.Y_dim)) @ Yj)
        P_Sk2_Yj_2 = 0
        for j1 in range(self.N1):
            P_Sk2_Yj_2 = P_Sk2_Yj_2+np.exp(mu[j1, K2]-
                beta[j1, K2, :].reshape((1,self.Y_dim))
                @self.Y_all[j1,:])
        P_Sk2_Yj = P_Sk2_Yj_1 / P_Sk2_Yj_2

        for j1 in range(self.N1):
            if j1 == i:
                g_mu[i, K1] = g_mu[i, K1] + P_Sk1_Yi *
                    P_Sk2_Sk1 * P_Sk2_Yj
                g_beta[i, K1, :] = g_beta[i, K1, :] +
                    (-Yi)*P_Sk1_Yi * P_Sk2_Sk1 *
                    P_Sk2_Yj
```

```

else:
    P3 = np.exp(mu[j1, K1] - beta[j1, K1,
        :].reshape((1, self.Y_dim))@self.
        Y_all[j1, :])
    P8 = (-P3*P1/P2/P2) * P_Sk2_Sk1 *
        P_Sk2_Yj
    g_mu[j1, K1] = g_mu[j1, K1] + P8
    g_beta[j1, K1, :] = g_beta[j1, K1, :] +
        (self.Y_all[j1, :])*(P3*P1/P2/P2)*
        P_Sk2_Sk1*P_Sk2_Yj

P_Sk2_Yj = P_Sk2_Yj_1 * (P_Sk2_Yj_2-P_Sk2_Yj_1)
        / P_Sk2_Yj_2 / P_Sk2_Yj_2
P_Sk1_Yi = P1 / P2

for j1 in range(self.N1):
    if j1 == j:
        g_mu[j, K2] = g_mu[j, K2] + P_Sk1_Yi *
            P_Sk2_Sk1 * P_Sk2_Yj
        g_beta[j, K2, :] = g_beta[j, K2, :] +
            (-Yj)*P_Sk1_Yi * P_Sk2_Sk1 *
            P_Sk2_Yj
    else:
        P3 = np.exp(mu[j1, K2] - beta[j1, K2,
            :].reshape((1, self.Y_dim))@self.
            Y_all[j1, :])
        P8 = P_Sk1_Yi * P_Sk2_Sk1 * (-P3*
            P_Sk2_Yj_1/P_Sk2_Yj_2/P_Sk2_Yj_2)
        g_mu[j1, K2] = g_mu[j1, K2] + P8
        g_beta[j1, K2, :] = g_beta[j1, K2, :] +
            (-self.Y_all[j1, :])*P8

P_Sk2_Yj = P_Sk2_Yj_1 / P_Sk2_Yj_2
P_Sk2_Sk1 = P_Sk2_Sk1_1 * (P_Sk2_Sk1_2-
        P_Sk2_Sk1_1) / P_Sk2_Sk1_2 / P_Sk2_Sk1_2

for j1 in range(self.N2):

```



```

        if j1 == K2:
            g_gamma[K2, 0] = g_gamma[K2, 0] +
                P_Sk1_Yi * P_Sk2_Sk1 * P_Sk2_Yj
            P6 = W - W
            for j2 in range(self.N2):
                P6 = P6 + (-W)*np.exp(gamma[j2, 0]
                    - alpha[:, K1].T @ W)
            P7 = ((-W) * P_Sk2_Sk1_1) @ P_Sk2_Sk1_2
            P6 = P6 @ P_Sk2_Sk1_1
            g_alpha[:, K1] = g_alpha[:, K1] + ((P7-
                P6) / P_Sk2_Sk1_2[0] / P_Sk2_Sk1_2
                [0])*P_Sk1_Yi*P_Sk2_Yj
        else:
            P4 = np.exp(gamma[j1, 0] - alpha[:, K1]
                ].T @ W)
            P5 = (-P4 * P_Sk2_Sk1_1 / P_Sk2_Sk1_2 /
                P_Sk2_Sk1_2) * P_Sk1_Yi * P_Sk2_Yj
            g_gamma[j1, 0] = g_gamma[j1, 0] + P5

    return g_mu, g_beta, g_gamma, g_alpha

# 参数粘贴函数
def SigmaShape(self, mu, beta, gamma, alpha):
    sigma1 = mu.reshape(-1, 1)
    sigma2 = beta.reshape(-1, 1)
    sigma3 = gamma.reshape(-1, 1)
    sigma4 = alpha.reshape(-1, 1)
    return np.concatenate((sigma1, sigma2, sigma3, sigma4),
        axis=0)

# 参数分割函数
def SigmaReshape(self, sigma):
    mu = sigma[:self.N2 * self.N1, 0].reshape((self.N1,
        self.N2))
    beta = sigma[self.N2 * self.N1:self.N2 * self.N1 + self
        .N2 * self.N1 * self.Y_dim, 0]
    beta = beta.reshape((self.N1, self.N2, self.Y_dim))
    N1 = self.N2 * self.N1 + self.N2 * self.N1 * self.Y_dim

```

```
N2 = self.N2 * self.N1 + self.N2 * self.N1 * self.Y_dim
    + self.N2

gamma = sigma[N1:N2, 0].reshape((self.N2, 1))
alpha = sigma[N2:, 0].reshape((self.W_dim, self.N2))
return mu, beta, gamma, alpha

# 损失函数 (算法目标函数)
def fun(self, sigma, Y, W, Pij):
    mu, beta, gamma, alpha = self.SigmaReshape(sigma)
    loss = 0
    for i in range(self.N1):
        for j in range(self.N1):
            pij = self.funPij(mu, beta, gamma, alpha, Y[i,
                :].T, Y[j, :].T, W, i, j)
            loss = loss + np.power(pij-Pij[i, j], 2)
    return loss

# 损失函数梯度
def gfun(self, sigma, Y, W, Pij):
    mu, beta, gamma, alpha = self.SigmaReshape(sigma)
    g_mu = self.mu - self.mu
    # print(g_mu)
    g_beta = self.beta - self.beta
    g_gamma = self.gamma - self.gamma
    g_alpha = self.alpha - self.alpha
    for i in range(self.N1):
        for j in range(self.N1):
            g_mu_n, g_beta_n, g_gamma_n, g_alpha_n = self.
                gfunPij(mu,beta,gamma,alpha,Y[i,:].T, Y[j,
                :].T, W, i, j)
            pij = self.funPij(mu, beta, gamma, alpha, Y[i,
                :].T, Y[j, :].T, W, i, j)
            g_mu = g_mu + 2*(pij-Pij[i, j])*g_mu_n
            g_beta = g_beta + 2*(pij-Pij[i, j])*g_beta_n
            g_gamma = g_gamma + 2*(pij-Pij[i, j])*g_gamma_n
            g_alpha = g_alpha + 2*(pij-Pij[i, j])*g_alpha_n
    sigma = self.SigmaShape(g_mu, g_beta, g_gamma, g_alpha)
```

```
        return sigma

# nonlinear_cg --FR  $g_k^T g_k / (g_{k-1}^T g_{k-1})$ 
def FR(self, g1, g2):
    print("调用FR公式")
    return np.dot(g2.T, g2) / np.dot(g1.T, g1)

# nonlinear_cg --PR
def PR(self, g1, g2):
    print("调用PR公式")
    return max(0, np.dot(g2.T, (g2 - g1)) / (np.dot(g1.T, g1)))

# x 当前迭代点 d 迭代方向 g当前迭代点的梯度 f 目标函数在当前点的函数值
# Armijo 步长规则
def Armijo(self, sigma0, Y, W, Pij, d, g, f):
    alpha1 = 1
    sigma = 1e-3
    print("调用Armijo步长规则")
    for i in range(20):
        sigma_n = sigma0 + alpha1*d
        fn = self.fun(sigma_n, Y, W, Pij)
        if fn < f + sigma * alpha1 * np.dot(g.T, d):
            break
        alpha1 = alpha1 * 0.1
    return alpha1

# 非线性共轭梯度算法框架
def nonlinear_cg(self, sigma0, Y, W, Pij, searching, beta_para):
    print('-----BEGIN-----')
    start = time.time()
    x_old = sigma0
    g_old = self.gfun(x_old, Y, W, Pij)
    # print("l:", g_old)
    d_old = -g_old
```

```
f_list = []
for k in range(self.kmax):
    f_old = self.fun(x_old, Y, W, Pij)
    print("当前残差函数值:", f_old)
    if k % 100 == 0:
        print(f_old)
    f_list.append(f_old[0])
    g_old_norm = np.linalg.norm(g_old, 2)
    print('当前迭代第{}步,当前精度值{:.8f}'.format(k,
        g_old_norm))
    if g_old_norm < self.eps:
        break
    alpha = searching(x_old, Y, W, Pij, d_old, g_old,
        f_old)
    # 更新 x
    x_new = x_old + alpha * d_old
    # 计算当前迭代点的梯度
    g_new = self.gfun(x_new, Y, W, Pij)
    beta = beta_para(g_old, g_new)
    d_new = -g_new + beta * d_old

    #更新参数
    d_old = d_new
    g_old = g_new
    x_old = x_new
end = time.time()
time_use = end - start
return x_new, f_list, k, time_use

# 算法主框架:
def main(self):
    nlcg_x1, nlcg_f1, nlcg_k1, nlcg_time1 = self.
        nonlinear_cg(self.sigma0, self.Y_all, self.W, self.
            Pij, \
```

self

.

Ar

```
print("最优参数为:", self.SigmaReshape(nlcg_x1))
mu, beta, gamma, alpha = self.SigmaReshape(nlcg_x1)
print("观测概率为:", self.P_Y_S_fun(mu, beta, gamma,
    alpha))
data_df = pd.DataFrame(self.P_Y_S_fun(mu, beta, gamma,
    alpha)) # 关键1, 将ndarray格式转换为DataFrame
# 更改表的索引
data_df.columns = ['S1', 'S2', 'S3', 'S4'] # 将第一行的
    0, 1, 2, ..., 9 变成 A, B, C, ..., J
data_df.index = ['Y1', 'Y2', 'Y3', 'Y4']

# 将文件写入excel表格中
writer = pd.ExcelWriter('P_Y_S.xlsx') # 关键2, 创建名
    称为hhh的excel表格
data_df.to_excel(writer, 'page_1',
    float_format='%.5f') # 关键3,
    float_format 控制精度, 将data_df写
    到hhh表格的第一页中。若多个文件, 可
    以在page_2中写入

writer.save()
print('“Armijo&FR” 最优的目标函数值为{:.2f}, 使用时间:
    {:.2f}s'.format(nlcg_f1[-1], nlcg_time1))

plt.rcParams['font.sans-serif'] = ['SimHei'] # 指定默
    认字体 SimHei 为黑体
plt.rcParams['axes.unicode_minus'] = False
plt.plot(range(nlcg_k1 + 1), nlcg_f1, 'r', markersize
    =2, linewidth=2, label='强 Wolfe+FR')
plt.legend()
```

```
plt.xlabel("迭代次数")
plt.ylabel("目标函数值")
plt.title("非线性共轭梯度迭代过程")
plt.savefig("../figure/非线性共轭梯度法rand.png")
plt.show()

if __name__ == '__main__':
    NCG1 = NCG()
    NCG1.main()
```

A.8 模拟退火(SA)算法 python 代码

```
import random
import matplotlib.pyplot as plt
import numpy as np
import math
import pandas as pd

num_city=30#城市总数
initial_t=120#初始温度
lowest_t=0.001#最低温度
M=150#当连续多次都不接受新的状态，开始改变温度
iteration=500#设置迭代次数
color = ['blue', 'red', 'green', 'orange', 'cyan', 'yellow']

location=np.loadtxt('city_location.txt')

#=====
#对称矩阵，两个城市之间的距离
def distance_p2p_mat():
    dis_mat=[]
    for i in range(30):
        dis_mat_each=[]
        for j in range(30):
            dis=math.sqrt(pow(location[i][0]-location[j][0],2)+
                            pow(location[i][1]-location[j][1],2))
```

```
        dis_mat_each.append(dis)
    dis_mat.append(dis_mat_each)
# print(dis_mat)
    return dis_mat

#计算所有路径对应的距离
def cal_newpath(dis_mat,path):
    dis=0
    for j in range(num_city-1):
        dis=dis_mat[path[j]][path[j+1]]+dis
    dis=dis_mat[path[29]][path[0]]+dis#回家
    return dis
for i_record in range(6):
    #=====
    #点对点距离矩阵
    dis_mat=distance_p2p_mat()
    #初始路径
    path=list(range(30))
    #初始距离
    dis=cal_newpath(dis_mat,path)
    #初始温度
    t_current=initial_t
    dis_min_record = []
    path_min_record = []

    while (t_current>lowest_t):#外循环，改变温度
        count_m=0#M的计数
        count_iter=0#迭代次数计数
        while (count_m<M and count_iter<iteration):#内循环，连续多次不接受新的状态或者是迭代多次，跳出内循环
            i=0
            j=0
            while(i==j):#防止随机了同一城市
                i=random.randint(1,29)
                j=random.randint(1,29)
            path_new=path.copy()
            path_new[i],path_new[j]=path_new[j],path_new[i]#任
```



```
        意交换两个城市的位置,产生新解
    #计算新解的距离
    dis_new=cal_newpath(dis_mat,path_new)
    #求差
    dis_delta=dis_new-dis
    #取0-1浮点随机数
    rand=random.random()
    #计算指数函数的值
    exp_d=math.exp(-dis_delta/t_current)
    #选择
    if dis_delta<0:
        path=path_new
        dis=dis_new
    elif exp_d>rand:
        path=path_new
        dis=dis_new
    else:
        count_m=count_m+1
        count_iter=count_iter+1
        dis_min_record.append(dis)
        path_min_record.append(path)
        # print('0最短距离: ', dis_min)
        # print('0最短路径: ', path_min)
        t_current=0.99*t_current#改变温度
#外循环结束
dis_min_test=pd.DataFrame(columns=['Y'],data=dis_min_record
)
dis_min_test.to_csv("./SAdis_record0"+str(i_record)+".csv")
x_record = np.array(range(0, len(dis_min_test['Y']), 1))
dis_min_test_scatter = dis_min_test['Y']
plt.scatter(x_record, dis_min_test_scatter, s=0.05, c=color
            [i_record], alpha=1)
dis_min=dis
path_min=path
print('最短距离: ',dis_min)
print('最短路径: ',path_min)
plt.xlabel("iteration (SA)")
```

```
plt.ylabel("length (Km)")
plt.legend(['1', '2', '3', '4', '5', '6'])
plt.savefig(r'./SAlength_six_maxalpha.png', dpi=300)
plt.show()
```

A.9 遗传 (GA) 算法 python 代码

```
import random
import numpy as np
import math
import matplotlib.pyplot as plt
import pandas as pd
num_city=30#城市总数0-29
num_total=100#随机生成的初始解的总数
copy_num=70#保留的解的个数
cross_num=20#交叉解的个数
var_num=10#变异解的个数
color = ['blue', 'red', 'green', 'orange', 'cyan', 'yellow']
location=np.loadtxt('city_location.txt')
#print(location)

#随机生成初始解[[[]],[[]],[[]]...]
def generate_initial():
    initial=[]
    city=list(range(num_city))
    for i in range(num_total):
        random.shuffle(city)
        p=city.copy()
        while (p in initial):
            #print('2333')#随机了一个重复的解
            random.shuffle(city)
            p=city.copy()
        initial.append(p)
    return initial

#对称矩阵，两个城市之间的距离
```

```
def distance_p2p_mat():
    dis_mat=[]
    for i in range(30):
        dis_mat_each=[]
        for j in range(30):
            dis=math.sqrt(pow(location[i][0]-location[j][0],2)+
                            pow(location[i][1]-location[j][1],2))
            dis_mat_each.append(dis)
        dis_mat.append(dis_mat_each)
    # print(dis_mat)
    return dis_mat

#目标函数计算,适应度计算,中间计算。适应度为1/总距离*10000
def dis_adp_total(dis_mat,initial):
    dis_adp=[]
    # dis_test=[]
    for i in range(num_total):
        dis=0
        for j in range(num_city-1):
            dis=dis_mat[initial[i][j]][initial[i][j+1]]+dis
        dis=dis_mat[initial[i][29]][initial[i][0]]+dis#回家
    # dis_test.append(dis)
    dis_adp_each= 10000.0/dis
    dis_adp.append(dis_adp_each)
    # print(dis_test)
    return dis_adp

def choose_fromlast(dis_adp,answer_source):
    mid_adp=[]
    mid_adp_each=0
    for i in range(num_total):
        mid_adp_each=dis_adp[i]+mid_adp_each
        mid_adp.append(mid_adp_each)
    # print(mid_adp)
    #产生0-mid_adp[num_total-1]之间的随机数
    #选择n-1<随机数<n的那个n的解,保留
    copy_ans=[]
```

```
for p in range(copy_num):
    rand=random.uniform(0,mid_adp[num_total-1])#产生随机数
    # print(rand)
    # print(p)
    for j in range(num_total):
        if (rand<mid_adp[j]):#查找位置
            copy_ans.append(answer_source[j])
            break
        else:
            continue
    return copy_ans
```

#随机选择保留下来的70中的25个进行交叉

```
def cross_prnew(copy_ans):
    for i in range(cross_num):
        which=random.randint(0,copy_num-1)#选择对那个解交叉
        cross_list=copy_ans[which].copy()
        while (cross_list in copy_ans):
            p=random.randint(0,num_city-1)
            q=random.randint(0,num_city-1)
            cross_list[p],cross_list[q]=cross_list[q],
            cross_list[p]#第一次交换位置
            m=random.randint(0,num_city-1)
            n=random.randint(0,num_city-1)
            cross_list[m],cross_list[n]=cross_list[n],
            cross_list[m]#第二次交换位置
        copy_ans.append(cross_list)
    cross_ans=copy_ans.copy()
    return cross_ans
```

#随机选择那95中的5个进行变异

```
def var_prnew(cross_ans):
    for i in range(var_num):
        which=random.randint(0,copy_num+cross_num-1)#选择对那个
        解交叉
        var_list=cross_ans[which].copy()
        while (var_list in cross_ans):
```

```
p=random.randint(0,num_city-1)
q=random.randint(0,num_city-1)
var_list[p],var_list[q]=var_list[q],var_list[p] # 交换位置
cross_ans.append(var_list)
var_ans=cross_ans.copy()
return var_ans

for i_record in range(6):
    dis_min_record = []
    #path_min_record = []
    answer_source=generate_initial()
    dis_mat=distance_p2p_mat()
    #print(dis_mat)
    dis_adp=dis_adp_total(dis_mat,answer_source)
    adp_max_new=max(dis_adp)
    if (max(dis_adp)>10000/700):
        print('找到的最近距离是: ',max(dis_adp))

    else:
        print('哎呀没找到, 我再找找~')
        answer_new=answer_source
        dis_adp_new=dis_adp
        i_0 = 0
        while(adp_max_new<=10000/700):
            copy_answer=choose_fromlast(dis_adp_new,answer_new)
            cross_answer=cross_proneu(copy_answer)
            var_answer=var_proneu(cross_answer)
            answer_new=var_answer.copy()
            dis_adp_new=dis_adp_total(dis_mat,answer_new)
            adp_max_new=max(dis_adp_new)
            dis_min=10000/adp_max_new
            #print('这次是: ',dis_min)
            dis_min_record.append(dis_min)
            #if i_0 >= 200000: break
            #else: i_0 += 1
```

```
dis_min=10000/adp_max_new
print('终于找到你啦:',dis_min)
dis_min_test = pd.DataFrame(columns=['Y'], data=
dis_min_record)
dis_min_test.to_csv("./GAdis_record0" + str(i_record) + ".
csv")
x_record = np.array(range(0, len(dis_min_test['Y']), 1))
dis_min_test_scatter = dis_min_test['Y']
plt.scatter(x_record, dis_min_test_scatter, s=0.05, c=color
[i_record], alpha=1)
print('最短距离:', dis_min)
#print('最短路径:', path_min)
plt.xlabel("iteration (GA)")
plt.ylabel("length (Km)")
plt.legend(['1', '2', '3', '4', '5', '6'])
plt.savefig(r'./GAlength_six_maxalpha.png', dpi=300)
plt.show()
```

A.10 蚁群 (ACO) 算法 python 代码

```
import random
import numpy as np
import math
import matplotlib.pyplot as plt
import pandas as pd

location=np.loadtxt('city_location.txt')
num_ant=200 #蚂蚁个数
num_city=30 #城市个数
alpha=1 #信息素影响因子
beta=1 #期望影响因子
info=0.1 #信息素的挥发率
Q=1 #常数

count_iter = 0
iter_max = 70
```

```
color = [ 'blue', 'red', 'green', 'orange', 'cyan', 'yellow' ]
#dis_new=1000
#=====
#对称矩阵，两个城市之间的距离
def distance_p2p_mat():
    dis_mat=[]
    for i in range(num_city):
        dis_mat_each=[]
        for j in range(num_city):
            dis=math.sqrt(pow(location[i][0]-location[j][0],2)+
                           pow(location[i][1]-location[j][1],2))
            dis_mat_each.append(dis)
        dis_mat.append(dis_mat_each)
    # print(dis_mat)
    return dis_mat

#计算所有路径对应的距离
def cal_newpath(dis_mat,path_new):
    dis_list=[]
    for each in path_new:
        dis=0
        for j in range(num_city-1):
            dis=dis_mat[each[j]][each[j+1]]+dis
        dis=dis_mat[each[num_city-1]][each[0]]+dis#回家
        dis_list.append(dis)
    return dis_list

#=====
for i_record in range(6):
    #点对点距离矩阵
    dis_list=distance_p2p_mat()
    dis_mat=np.array(dis_list)#转为矩阵
    #期望矩阵
    e_mat_init=1.0/(dis_mat+np.diag([10000]*num_city))#加对角阵
    是因为除数不能是0
    diag=np.diag([1.0/10000]*num_city)
    e_mat=e_mat_init-diag#还是把对角元素变成0
```

```
#初始化每条边的信息素浓度，全1矩阵
pheromone_mat=np.ones((num_city,num_city))
#初始化每只蚂蚁路径，都从0城市出发
path_mat=np.zeros((num_ant,num_city)).astype(int)
dis_min_record = []
path_min_record = []

#while dis_new>400:
while count_iter < iter_max:
    for ant in range(num_ant):
        visit=0#都从0城市出发
        unvisit_list=list(range(1,30))#未访问的城市
        for j in range(1,num_city):
            #轮盘法选择下一个城市
            trans_list=[]
            tran_sum=0
            trans=0
            for k in range(len(unvisit_list)):
                trans +=np.power(pheromone_mat[visit][
                    unvisit_list[k]],alpha)*np.power(e_mat[
                    visit][unvisit_list[k]],beta)
                trans_list.append(trans)
                tran_sum =trans

            rand=random.uniform(0,tran_sum)#产生随机数

            for t in range(len(trans_list)):
                if(rand <= trans_list[t]):
                    visit_next=unvisit_list[t]

                    break
            else:
                continue
            path_mat[ant,j]=visit_next#填路径矩阵

            unvisit_list.remove(visit_next)#更新
```



```
visit=visit_next#更新

#所有蚂蚁的路径表填满之后，算每只蚂蚁的总距离
dis_allant_list=cal_newpath(dis_mat,path_mat)

#每次迭代更新最短距离和最短路径
if count_iter == 0:
    dis_new=min(dis_allant_list)
    path_new=path_mat[dis_allant_list.index(dis_new)].
        copy()
else:
    if min(dis_allant_list) < dis_new:
        dis_new=min(dis_allant_list)
        path_new=path_mat[dis_allant_list.index(dis_new
            )].copy()
    #print(dis_new)
dis_min_record.append(dis_new)
path_min_record.append(path_new)
# 更新信息素矩阵
pheromone_change=np.zeros((num_city,num_city))
for i in range(num_ant):
    for j in range(num_city-1):
        pheromone_change[path_mat[i,j]][path_mat[i,j
            +1]] += Q/dis_mat[path_mat[i,j]][path_mat[i,
            j+1]]
    pheromone_change[path_mat[i,num_city-1]][path_mat[i
        ,0]] += Q/dis_mat[path_mat[i,num_city-1]][
        path_mat[i,0]]
pheromone_mat=(1-info)*pheromone_mat+pheromone_change
count_iter += 1 #迭代计数+1，进入下一次

#print('最短距离: ',dis_new)
#print('最短路径: ',path_new)
dis_min_test = pd.DataFrame(columns=['Y'], data=
    dis_min_record)
dis_min_test.to_csv("./ACODis_record0" + str(i_record) + ".
    csv")
```

```
x_record = np.array(range(0, len(dis_min_test['Y']), 1))
dis_min_test_scatter = dis_min_test['Y']
plt.plot(x_record, dis_min_test_scatter, c=color[i_record],
         alpha=1)

dis_min = dis_new
path_min = path_new
print('最短距离: ', dis_min)
print('最短路径: ', path_min)
plt.xlabel("iteration (ACO)")
plt.ylabel("length (Km)")
plt.legend(['1', '2', '3', '4', '5', '6'])
plt.savefig(r'./ACOlenght_six_maxalpha.png', dpi=300)
plt.show()
```

A.11 禁忌搜索 (TS) 算法 python 代码

```
import random
import numpy as np
import math
import matplotlib.pyplot as plt
import pandas as pd
num_city=30#城市总数
table_len=20#禁忌表长度
location=np.loadtxt('city_location.txt')
taboo_table=[]
color = ['blue', 'red', 'green', 'orange', 'cyan', 'yellow']

#=====
#对称矩阵，两个城市之间的距离
def distance_p2p_mat():
    dis_mat=[]
    for i in range(30):
        dis_mat_each=[]
        for j in range(30):
            dis=math.sqrt(pow(location[i][0]-location[j][0],2)+
                           pow(location[i][1]-location[j][1],2))
```

```
        dis_mat_each.append(dis)
    dis_mat.append(dis_mat_each)
# print(dis_mat)
    return dis_mat

#计算所有路径对应的距离
def cal_newpath(dis_mat,path_new):
    dis_list=[]
    for each in path_new:
        dis=0
        for j in range(num_city-1):
            dis=dis_mat[each[j]][each[j+1]]+dis
        dis=dis_mat[each[29]][each[0]]+dis#回家
        dis_list.append(dis)
    return dis_list

#寻找上一个最优路径对应的所有领域解
def find_newpath(path_best):
    path_new=[]
    for i in range(1,num_city-1):
        for j in range(i+1,num_city):
            path=path_best.copy()
            path[i],path[j]=path[j],path[i]
            path_new.append(path)
    return path_new

#=====
for i_record in range(6):
    #点对点距离矩阵
    dis_mat=distance_p2p_mat()

    #设置初始解
    path_initial=[]
    initial=list(range(30))
    path_initial.append(initial)
    #print(path_initial)
```

```
#加入禁忌表
taboo_table.append(initial)

#求初始解的路径长度
dis_list=cal_newpath(dis_mat,path_initial)
dis_best=min(dis_list)#最短距离
path_best=path_initial[dis_list.index(dis_best)]#对应的最短
    路径方案
#print(path_best)

#初始期望
expect_dis=dis_best
expect_best=path_best

#记录值
dis_min_record = []
path_min_record = []
for iter in range(50):#迭代
    #寻找全领域新解
    path_new=find_newpath(path_best)
    #print(path_new)

    #求出所有新解的路径长度
    dis_new=cal_newpath(dis_mat,path_new)
    #print(dis_new)

    #选择路径
    dis_best=min(dis_new)#最短距离
    #print(dis_best)
    path_best=path_new[dis_new.index(dis_best)]#对应的最短
        路径方案
    dis_min_record.append(dis_best)
    path_min_record.append(path_best)
    if dis_best < expect_dis:#最短的<期望
        expect_dis=dis_best
        expect_best=path_best#更新两个期望
```

```

        if path_best in taboo_table:
            taboo_table.remove(path_best)
            taboo_table.append(path_best)
        else:
            taboo_table.append(path_best)
    else: #最短的还是不能改善期望
        if path_best in taboo_table: #在禁忌表里
            dis_new.remove(dis_best)
            path_new.remove(path_best)
            dis_best=min(dis_new) #求不在禁忌表中的最短距离
            path_best=path_new[dis_new.index(dis_best)] #对
                应的最短路径方案
            taboo_table.append(path_best)
        else: #不在禁忌表
            taboo_table.append(path_best)
        if len(taboo_table)>=table_len:
            del taboo_table[0]
    dis_min_test = pd.DataFrame(columns=['Y'], data=
        dis_min_record)
    dis_min_test.to_csv("./TS_dis_record0" + str(i_record) + ".
        csv")
    x_record = np.array(range(0, len(dis_min_test['Y']), 1))
    dis_min_test_scatter = dis_min_test['Y']
    plt.plot(x_record, dis_min_test_scatter, c=color[i_record],
        alpha=1)

    print('最短距离', expect_dis)
    print('最短路径: ', expect_best)
plt.xlabel("iteration (TS)")
plt.ylabel("length (Km)")
plt.legend(['1', '2', '3', '4', '5', '6'])
plt.savefig(r'./TSlength_six_maxalpha.png', dpi=300)
plt.show()

```