第 5 章 其他算法

在计算机领域中，除了包含前述比较常用且难度中等的算法，同时还包含一些复杂的算法，而这些算法将是未来面临更难问题时的重要助手。比方说本章中的机器学习，对于现在可以说是非常热门的话题。毋庸置疑，它们是比较复杂的，既可能涉及复杂逻辑的理解，还可能涉及数学计算。不过，读者可以放心学习，因为本章是按照从简到难的顺序进行排列的：分治算法、贪心算法为第一梯队，它们相对来讲简单一些；哈希算法、数值分析法为第二梯队，因为它们涉及一些复杂的数学计算；机器学习和神经网络则为第三梯队，因为它们既包含复杂的运算，同时也有诸多难以理解的逻辑过程。

5.1 分治算法

分治算法，字面意思就是“分而治之”，即将复杂问题拆解为多个更小的子问题，并递归求解这些子问题，最终合并后便会得到原问题的解。它在许多经典算法中扮演着核心角色，例如二分查找和快速排序等。通过深入理解分治思想，开发者不仅能更高效地解决各种复杂问题，还能为程序性能优化奠定坚实的基础。

5.1.1 图解分治算法

1. 斐波那契函数

分治算法通常依赖于递归来实现，而递归的实现方式需要遵循两个重要的原则。首先是定义递归的基本情况，即递归的终止条件；其次是定义递归的规则或逻辑，这涉及递归函数的调用方式和数据传递。

在经典的 factorial() 函数中，假设需要计算一个正整数 n 的阶乘，递归的终止条件是当 n 等于 0 或 1 时返回数值 1。这是因为 0 的阶乘和 1 的阶乘都等于 1。递归的逻辑部分则是当 n 不等于 0 或 1 时，函数会通过自身调用来继续计算，即 n 乘以 factorial（$n-1$）。这种通过不断减小问题规模（将 n 逐步减小）的方式，使递归最终能到达终止条件。

在 Python 语言中，递归的实现不仅要求程序员明确递归的逻辑和终止条件，还需要考虑递归调用的深度。Python 编译器对递归调用的深度有一定限制，这意味着在处理非常大

的问题时，递归可能会引发栈溢出。

递归算法的详细教程见 3.1 节。斐波那契函数实现过程的代码如下：

```
// 第 5 章 / 斐波那契函数 .py
def factorial(n):
    # 递归基本情况：当 n 等于 0 或 1 时，阶乘为 1
    if n == 0 or n == 1:
        return 1
    # 递归规则：阶乘的递归定义为 n * (n-1)!
    return n * factorial(n - 1)

# 测试递归函数
print(factorial(5))  # 输出 120，即 5 的阶乘
```

2. 归并排序

在讨论归并排序与递归算法时，需要明确一点，也就是归并排序和递归算法并不能简单地等同。归并排序是一种算法思维方式，而递归则是一种实现这一算法的常见方法。可以理解为归并排序是想出来怎么排序，而递归则是实现这种排序的方法。不过归并排序既可以通过递归来实现，也可以通过迭代的方式来实现，因此递归只是归并排序的一种实现手段，而并非归并排序的唯一表达形式。

举个例子，假设存在一个包含 8 个元素的数组，数组内容为（6，3，8，1，2，4，9，5）。可以通过分治算法中的归并排序对其进行排序。归并排序的核心思想是将数组不断地分解为更小的子数组，然后对子数组进行排序，最后通过合并这些已经排序的子数组，逐步得到排序完成后的完整数组。

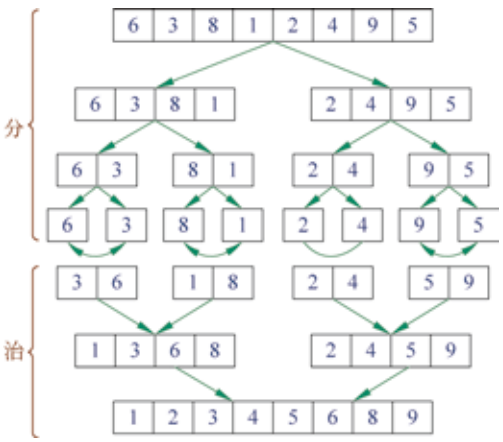


图 5-1　分治算法示意图

具体来讲，首先将数组分解为两部分，每部分包含 4 个元素。接着，再对每个子数组继续进行折半分割，直到每个子数组只包含一个元素。在这一过程中，数组最终被分割为单个元素的集合。如图 5-1 所示，原数组最终被分割为 {6}，{3}，{8}，{1}，{2}，{4}，{9}，{5} 这 8 个单一元素。

当数组被分割为单一的元素后，下一步便是对这些元素进行排序和合并。在此过程中，由于要求结果为升序排列，因此每两个元素在合并前都需要进行比较。这种情况称为治理。例如，当元素 6 和元素 3 需要合并时，首先进行比较。由于元素 6 大于元素 3，因此根据降序的规则，需要将它们交换位置，最终结果是 {3，6}。经过这样的比较操作后，两个元素被合并为一个有序的小数组，即 {3，6}。后续的单一元素都将按照类似的步骤进行处理，直至所有元素都被合并为多个小数组。

以这个具体的例子为基础，在初步的合并阶段，最终将 8 个单一的元素分别合并为以下四个小数组：{3，6}，{1，8}，{2，4}，{5，9}。接下来，仍然采用相同的比较与合并步骤，将这些小数组两两合并，从而形成更大的数组。以 {3，6} 和 {1，8} 为例，首先对这两个数组进行比较和合并。准备一个新的空数组用于存储合并结果，然后同时遍历两个数组，在遍历过程中，比较两个数组的当前元素。例如，若数组 *a* 的第一个元素大于数组 *b* 的第一个元素（在本例中，先比较元素 3 和元素 1），则将数组 *b* 中较小的元素 {1} 加入到新数组中。接着，数组 *b* 的索引值加 1，再次与数组 *a* 的第一个元素 {3} 进行比较，发现元素 8 大于元素 3，于是将元素 3 加入合并数组当中。这一操作会持续进行，直至某个数组的所有元素都被放入合并数组中。若一个数组的元素全部处理完毕，剩余的元素将直接从另一个数组中依次加入到合并数组中。举例来说，在处理完较小的元素后，若 *a* 数组中的所有元素都已合并完，则剩余的 *b* 数组元素将直接添加到合并结果中。以此类推，经过多次两两合并，最终会得到一个完整的、排序完成的数组。

归并操作实现过程的代码如下：

```
// 第 5 章 / 归并排序 .py
def merge_and_sort(a, b):
    merged_array = []                       # 创建一个空数组，用于存放合并后的结果
    i = len(a) - 1                          # 初始化 a 数组的指针，从末尾开始
    j = len(b) - 1                          # 初始化 b 数组的指针，从末尾开始

    # 循环直到 a 和 b 数组的指针都大于或等于 0
    while i >= 0 and j >= 0:
        if a[i] > b[j]:
            merged_array.append(a[i])       # 将 a 中较大的元素添加到合并数组中
            i -= 1
        else:
            merged_array.append(b[j])       # 将 b 中较大的元素添加到合并数组中
            j -= 1

    # 将剩余未处理的元素添加到合并数组中
    while i >= 0:
        merged_array.append(a[i])
        i -= 1
    while j >= 0:
        merged_array.append(b[j])
        j -= 1

    # 返回合并后的数组
    return merged_array

# 示例
a = [6, 3]
b = [8, 1]
merged_and_sorted = merge_and_sort(a, b)
print("Merged and sorted array:", merged_and_sorted)
```

5.1.2 分治算法代码写作指导

分治算法的代码编写通常可以分为两个核心步骤：首先是“分”，即将问题逐步拆解为更小的子问题；其次是“治”，即通过对子问题的解进行组合，最终合并回到原始问题的解决方案。本节通过 Karatsuba 算法来展示如何编写分治算法的代码。

Karatsuba 算法，即大整数乘法，在日常生活中的应用非常广泛。Karatsuba 算法通俗的解释即小学中所学到列竖式相乘，如图 5-2 所示。假设现在有两个数，分别是 13 和 24，希望通过列竖式来计算它们的乘积。首先，将第 1 个数的个位和第 2 个数的个位相乘，即 3 乘以 4，得到 12。此时，个位数为 2，进位 1。接着，将第 1 个数的十位与第 2 个数的个位相乘，即 1 乘以 4，得到 4，再加上之前的进位 1，得到 5。接下来，继续处理第 2 个数的十位。将第 2 个数的十位与第 1 个数的个位相乘，即 2 乘以 3，得到 6。最后，将第 2 个数的十位与第 1 个数的十位相乘，即 1 乘以 2，得到 2。将这些结果相加，最终得到的乘积为 312。

将上述乘法进行公式化，原本第 1 个数是 13，第 2 个数是 24，现在通过字母代替数字的方式对其进行抽象化，如图 5-3 所示。b 和 d 进行相乘，其结果为 bd，a 和 d 相乘，其结果为 ad，以此类推。最终结果 bd 和 0 相加，其结果还是 bd；ad 和 bc 相加，其结果是 $ad+bc$；ac 和 0 相加，其结果是 ac。

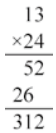


图 5-2 竖式乘法表达式

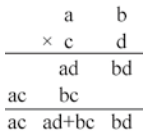


图 5-3 抽象化竖式相乘

Karatsuba 乘法的实现过程可以分为几个明确的步骤，首先要判断参与相乘的两个数字是否为单个数字。若其中有一个是单个数字，那么直接进行乘法运算即可，否则需要使用 Karatsuba 乘法的分治思想进行运算，即通过递归分解和合并来实现大整数的乘法。

当两个数字都不为单个数字时，Karatsuba 算法就会采用分治策略。分治算法的第 1 步是“分”，即将每个数字按位数分解，例如将十位与个位分离。以 Python 为例，利用基础的运算符取整（//）和 取余（%），可以方便地获取数字的十位和个位。例如，对于两位数 n，n//10 可以得到十位，而 n % 10 则可以得到个位。这种简单的数值分解是 Karatsuba 算法的基础。

在 Karatsuba 乘法中如果有两个数字，则需要用到分治算法。分治算法首先需要分，要将一个数字的个位和十位进行分离，对个位和十位的值以如图 5-3 所示的方法进行赋值。通过 Python 的基础运算法则，通过取整方法和取余方法分别获得十位和个位。

仍以 13 和 24 的乘法为例，详细说明 Karatsuba 算法的运作过程。当 13 与 24 相乘时，由于这两个数都不是单个数字，因此进入下一步，将它们分解为 a、b、c、d 四部分。其中，a 是 13 的十位，b 是 13 的个位；同样，c 是 24 的十位，d 是 24 的个位。具体来讲，在这个例子中，a 的值为 1，b 的值为 3，c 的值为 2，d 的值为 4。

一旦完成分解，下一步就是递归地进行相应的乘法运算，例如，首先需要计算 *b* 与 *d* 的乘积，即 3 乘以 4。此时，由于 3 和 4 都是单个数字，Karatsuba 算法中的 if 条件判断成立（如果两个数中有一个是单个数字，则直接相乘），所以可以直接得到结果 12。接下来，以类似的方法进行其他的乘法运算，包括 *a* 与 *c* 的乘积、*a* 与 *d* 的乘积、*b* 与 *c* 的乘积等。

通过对这些局部的乘积进行相加和适当地进行位移（根据十位和个位的差异），可以最终合并成完整的乘积结果。在这个例子中，通过这些递归调用和局部相加，13 乘以 24 的最终结果是 312。

Karatsuba 乘法实现过程的代码如下：

```
// 第 5 章 /Karatsuba 乘法 .py
def karatsuba(x, y):
    # 判断输入的数字是否为单个数字
    if x < 10 and y < 10:
        return x * y

    # 计算数字的位数
    n = max(len(str(x)), len(str(y)))
    half_n = n //2

    # 将数字分解为两部分
    a = x //10 ** half_n
    b = x % (10 ** half_n)
    c = y //10 ** half_n
    d = y % (10 ** half_n)

    # 递归计算 3 个乘积
    ac = karatsuba(a, c)
    bd = karatsuba(b, d)
    ad_bc = karatsuba(a + b, c + d) - ac - bd

    # 合并乘积便可得到结果
    result = ac * (10 ** (2 * half_n)) + ad_bc * (10 ** half_n) + bd

    return result

# 测试
x = 13
y = 24
print("Result:", karatsuba(x, y))
```

5.1.3　分治算法实际应用介绍

随机森林是一种集成学习算法，其核心思想是在多棵决策树的基础上进行预测，并通过集成的方式提升模型的准确性和稳健性。在这种算法的设计中融入了分治算法的思想，该思想不仅体现在每棵决策树的构建过程中，还在于预测的过程中如何有效地组合不同树

的结果，从而实现最终的决策。

在随机森林中，每棵决策树是通过对训练数据的子集进行训练而生成的。在这一过程中，分治算法的思想非常明显，具体体现为将原始数据集划分为若干个子集，分别训练多棵决策树，每棵树独立地对数据进行学习和预测。最终，随机森林集成所有决策树的预测结果，常见的方式是通过投票机制（对于分类问题）或取平均值（对于回归问题）来输出最终的预测值。通过这种集成方式，随机森林有效地减小了单棵决策树可能产生的过拟合问题，提高了模型的泛化能力。

决策树是一种经典的机器学习算法，它采用树结构来帮助模型进行决策。在构建决策树的过程中，内部节点用于表示对一个特征的测试，而每个分支则代表测试结果的不同分支情况。最终，每个叶节点对应于一个类别标签（用于分类任务）或一个预测值（用于回归任务）。决策树的构建过程可以分为两个主要步骤：首先是树的生成过程，随后是剪枝操作。

机器学习是人工智能领域的一个重要分支，专注于通过数据驱动的方式进行模型构建，使计算机系统能够在没有明确编程指令的情况下，从数据中自动学习和提取规律。这一过程的目标是使计算机系统能够基于这些学习到的模式进行预测或决策，而无须人为干预。换句话说，机器学习通过对大量数据进行分析和建模，使系统具备自我学习的能力，并能根据所学内容做出适当的反应或选择。关于机器学习的具体算法及其详细实现，将在 5.5 节中更深入地进行探讨，这里仅作简单介绍，以帮助读者对其有基本的理解。

1. 随机森林数据集选取

在随机森林算法中，为了构建每棵决策树，模型会从原始数据集中随机抽取一定数量的样本和特征进行训练。这一策略旨在降低模型的过拟合风险，通过引入随机性，使每棵决策树都能够在不同的数据子集和特征子集上进行学习，从而提高模型的泛化能力和稳健性。这种方法能够有效地减少各棵树之间的相关性，进而提升整体模型的性能。

本节使用经典的波士顿房价数据集作为示例数据集，以展示随机森林算法的应用。波士顿房价数据集由哈里斯和鲁宾菲尔德于 1978 年收集，包含了波士顿郊区住房的相关信息。该数据集是回归分析中的一个标准数据集，被广泛地应用于机器学习和统计建模的研究与实践中。

波士顿房价数据集包含 506 个房屋样本，每个样本对应于一处住房的价格信息。此外，数据集中还包含 14 个相关特征，这些特征对房屋价格的预测起着重要作用。相关特征包括房屋的位置、房间数量、地块面积、距离市中心的远近、邻近学校的质量等。具体特征的详细描述和数值范围，如表 5-1 所示，能够帮助我们理解这些因素对房价的影响。

表 5-1　房价相关特征表

变量名称	变量含义
CRIM	城镇人均犯罪率
ZN	住宅用地超过 25 000 平方英尺的比例
INDUS	城镇非零售营业面积占比
CHAS	查尔斯河亚变量（如果临河有大片土地，则为 1，否则为 0）

续表

变量名称	变量含义
NOX	一氧化氮浓度（千万分之一）
RM	平均每户的房间数
AGE	1940 年以前建成的自用住房比例
DIS	到 5 个波士顿就业中心的加权距离
RAD	辐射可达的公路的指数
TAX	每 10 000 美元的全额财产的税率
PTRATIO	城镇师生比例
B	1000 人中非裔美国人的比例
LSTAT	地位较低人口的百分比
MEDV	自住房中位价（以千美元为单位）

在波士顿房价数据集中，选择将 MEDV（自住房中位价）作为目标变量。目标变量是希望模型预测的主要输出，而解释变量则是我们用来预测目标变量的一组特征。在这个分析过程中，将从剩余的特征中选择一个或多个变量，作为解释变量来研究它们与 MEDV 之间的关系。

2. 相关性及相关系数

在建立回归模型之前，对数据进行分析是至关重要的一步。通过分析数据中可能与房价存在相关关系的特征，能够更好地理解影响房价的各种因素。这一过程通常采用数据可视化的方法，以便直观地呈现变量之间的关系。为了实现这一目标，绘制散点图矩阵和关联矩阵是比较常见的技术手段。

首先，需要人工选取想要观察的变量。波士顿房价数据集包含 14 个特征变量，探查所有变量可能会耗费过多时间，因此选择对房价影响较大的特征变量进行重点分析显得尤为重要。在本书中，选取了城镇人均犯罪率（CRIM）、平均每户房间数（RM）、每 10 000 美元的全额财产税率（TAX）、城镇师生比例（PTRATIO）和自住房中位价（MEDV）这 5 个变量进行详细探讨。

其次，通过初步考虑这些变量的意义，推测它们与房价之间的潜在关系。如果城镇中人均犯罪率较高，则可能会导致居住人口减少，进而对当地房价产生负面影响，如果平均每户的房间数较少，则可能意味着居住条件较差，从而降低房屋的市场价值。税率方面，如果纳税额过高，则可能会导致高收入人群流失，从而影响区域的房价。此外，城镇的师生比例如果过低，意味着教育资源不足，则可能会导致家庭迁出，进而造成房屋供过于求，进一步压低房价。

再次，在数据可视化过程中，使用 Python 的 Seaborn 库中的 sns.pairplot() 函数绘制散点图矩阵。该矩阵展示了各变量之间的两两关系，虽然绘制的图形可能显得庞大，但只需关注与 MEDV 的相关关系图。通过对散点图矩阵进行观察，可以直观地识别出变量之间的关系模式。某些变量之间可能呈现线性关系，而其他变量则可能呈现非线性关系，如图 5-4 所示。

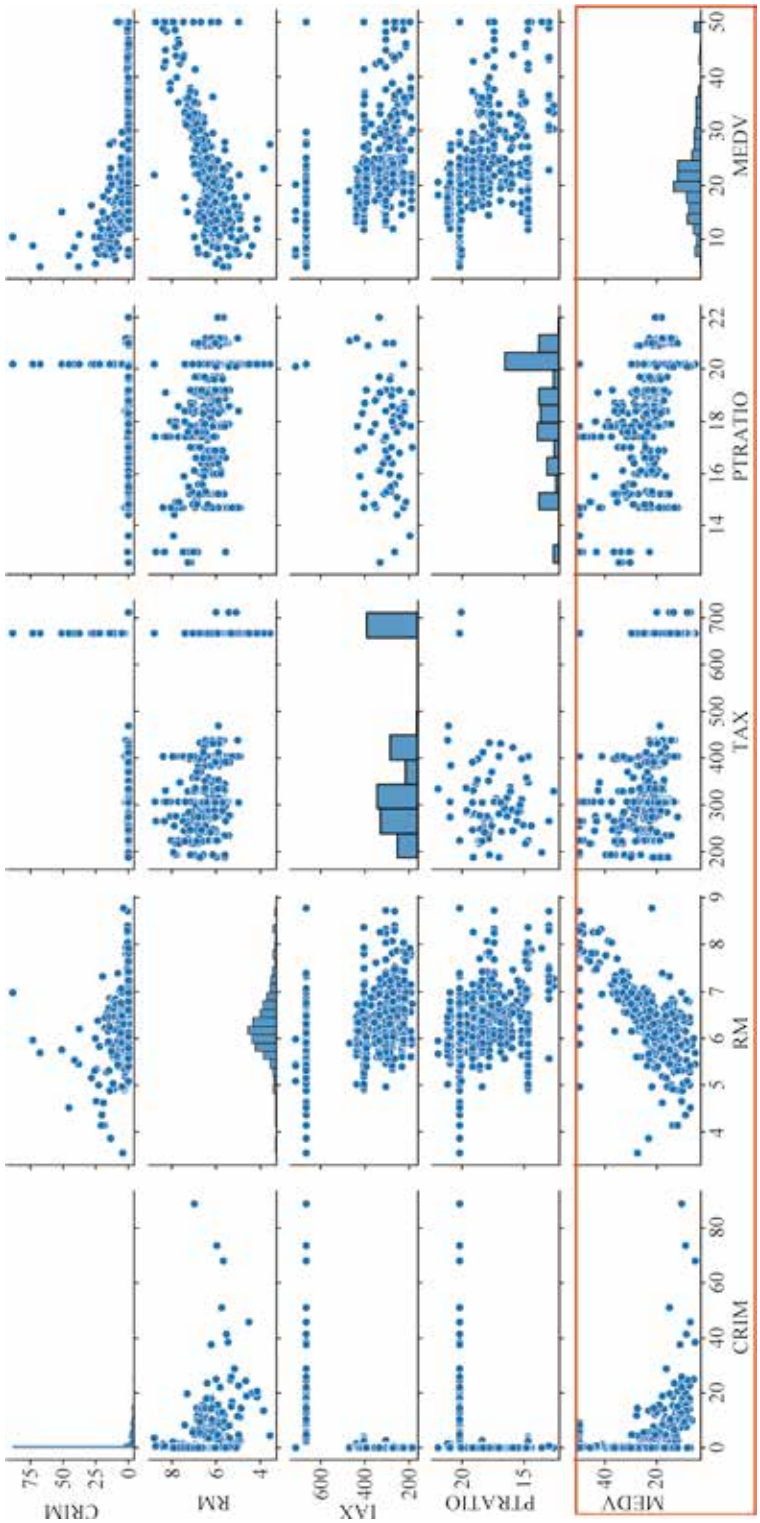


图 5-4　变量相关关系图

从图中可以明显地看出，MEDV 与 CRIM 之间呈现出非线性关系，而 MEDV 与 RM 之间则可能存在一种线性关系。

最后，为了更加准确地分析特征之间的相关性，接下来绘制关联矩阵。关联矩阵是一种工具，用于显示不同变量之间的相关关系，其取值范围通常为 −1~1。当相关系数接近 1 时，表示特征之间存在较强的正相关关系，而当相关系数接近 −1 时，则表示特征之间存在较强的负相关关系。通过这种方式能够更直观地识别出影响目标变量的关键因素。

在本节中将采用 Seaborn 图形可视化库来绘制关联矩阵。Seaborn 是基于 Matplotlib 的高级数据可视化库，能够更方便地创建美观的统计图表，使用 seaborn.heatmap() 函数来生成热图形式的关联矩阵，如图 5-5 所示。

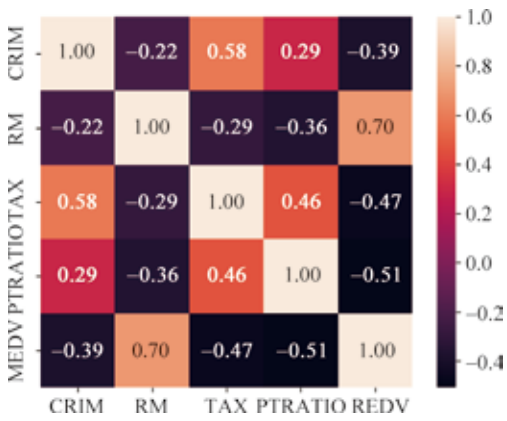


图 5-5　特征相关系数表

从关联矩阵中，可以清晰地观察到各个特征之间的相关系数。在前面的散点图矩阵中，初步预测 MEDV（自住房中位价）和 RM（平均每户房间数）之间存在一定的相关关系，而在绘制的关联矩阵中，这一关系得到了进一步验证。具体而言，MEDV 与 RM 之间的相关系数为 0.7，这表明它们之间存在较强的正相关关系。这意味着，随着平均每户房间数的增加，房屋的中位价格也有可能随之上升。

3. 随机森林建立

通过数据可视化和相关性分析确认了 MEDV（自住房中位价）与 RM（平均每户房间数）之间存在显著的相关关系，并且得出的相关系数为 0.7，表明这两个特征之间具有较强的正相关性。基于这一发现，接下来将以 RM 和 MEDV 为特征与目标变量，分别建立随机森林回归模型。

在构建随机森林回归模型之前，需要对数据集进行划分，以便于模型进行训练和评估。数据集通常被分为训练集和测试集，训练集用于训练模型，而测试集则用于评估模型在未见数据上的表现。为了确保模型的泛化能力，训练集和测试集的划分应当保持随机性，以反映出数据的整体特征。

在 Python 语言中使用 train_test_split() 函数来对训练集和测试集进行划分。设定合适的比例：70% 的数据作为训练集，30% 的数据作为测试集。训练完成后，模型将能够根据训练集中的模式进行学习，并形成能够预测房价的模型。

随机森林建立过程的代码如下：

```
// 第 5 章 / 波士顿房价 .py
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

#Load data
df = pd.read_csv('.\housing.data', header=None, sep='\s+')
df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']

#Visualize data
cols = ['CRIM', 'RM', 'TAX', 'PTRATIO', 'MEDV']
sns.pairplot(df[cols], height=2.5)
plt.tight_layout()
plt.show()

#Visualize correlation matrix
cm = np.corrcoef(df[cols].values.T)
hm = sns.heatmap(cm, cbar=True, square=True, fmt='.2f', annot=True, annot_kws=
{'size':15}, yticklabels=cols, xticklabels=cols)
plt.show()


#Prepare data for model
X = df['RM'].values.reshape(-1, 1)
y = df['MEDV'].values
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.4, random_
state=1)

#Train model
forest = RandomForestRegressor(n_estimators=1000, criterion='squared_error',
random_state=1, n_jobs=-1)
forest.fit(X_train,y_train)

# 模型预测
y_test_pred = forest.predict(X_test)

# 模型性能评估
mse_test = mean_squared_error(y_test, y_test_pred)
r2_test = r2_score(y_test, y_test_pred)
print("mse_test={:.2f} r2_test={:.2f}".format(mse_test, r2_test))

# 绘制残差图
plt.scatter(y_test_pred, y_test_pred-y_test, c='steelblue', edgecolor='white',
marker='s', s=35, alpha=0.9,label='test data')
plt.xlabel = ('Predicted values')
plt.ylabel = ('Residuals')
```

```
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
plt.xlim([10,48])
plt.tight_layout()
plt.show()
```

4. 分治算法在其中的应用

在构建决策树的过程中，分治算法发挥了重要的作用，主要用于确定最佳的特征及其对应的划分点。具体而言，在构建决策树时，需要从多个候选特征中选择最优特征，以最大限度地提高模型的准确性。在本节中，选取了城镇人均犯罪率、平均每户的房间数等特征作为解释变量。这些特征可能会被用作决策树节点的划分特征，帮助模型更好地理解数据的内在结构。

分治算法对每个特征的不同取值进行遍历，尝试不同的划分点。每次划分时，算法会计算划分后的子节点的纯度，通常使用基尼系数或熵作为衡量标准。目的是使每个子节点尽可能单一，提升节点的纯净度。例如，如果以城镇人均犯罪率为划分特征，则分治算法会寻找一个阈值，使低于该阈值的样本和高于该阈值的样本在后续的节点中表现出显著的不同的特征分布。

在对新样本进行房价预测时，分治算法可再次发挥其优势。模型会遍历每棵决策树的节点，并根据节点的划分规则决定样本应该向左子树还是向右子树移动。每个节点的划分标准都会引导样本逐步接近叶节点。叶节点代表最终的预测结果，通常是某个类别或一个具体的预测值。

当所有的决策树完成预测后，随机森林将通过统计或综合各棵树的预测结果来得到最终的预测房价。常用的方法包括投票机制或平均值计算。在这种集成学习的框架下，随机森林能够有效地降低模型的方差，提高预测的稳健性。这种方法尤其适用于处理复杂的数据集，能够捕捉到数据中的非线性关系，从而提供更为准确的预测结果。

5.2　贪心算法

贪心算法（又称为贪婪算法）是一类在问题求解过程中每步都选择当前状态下最优解的算法。这类算法的核心思想是，在每个阶段做出对当前状态看似最优的选择，期望能够在全局上得到最优解。然而，贪心算法的局限性在于，它只专注于每个阶段的局部最优解，而忽略了全局视角。由于这种局部的贪心选择，该算法并不能保证全局最优解。换句话说，贪心算法的每步决策虽然在局部是最优的，但累积起来并不一定能形成整体上的最优解。

其实在现实里也是这样，我们每个人都期待把当今的每步都做到最优解，以为每步的最优合起来就是全局最优，但是结果往往不是这样。当你局限于眼前，就忘记了去统筹自己的整个人生……在使用贪心算法时，必须仔细分析问题的特性，确认贪心策略能够保证全局最优解，或至少能够在精度和效率之间找到合适的平衡。

5.2.1 图解贪心算法

在解决复杂问题时，贪心算法通常会将整个问题分解为一系列相互关联的子问题，并试图通过每个子问题的最优解来构建最终问题的解。这一过程依赖于一个假设：如果在每步选择局部最优的方案，则最终也可以得到全局最优解，然而，贪心算法并非总能保证全局最优解，这也是其主要局限性所在。与贪心算法相提并论的是动态规划，两者在问题求解的方式上存在显著差异。

动态规划通过保存每个子问题的中间结果，能够避免重复计算，并且具备“回退”的能力。具体来讲，动态规划可以在后续的决策中参考先前的计算结果，从而做出更加全局化的判断和选择，因此动态规划的策略是通过综合考虑所有可能的解路径来确保最终得到全局最优解。

相比之下，贪心算法没有记忆功能，也不具备回退机制。它的每步决策都是基于当前的局部最优选择，无法根据历史信息进行调整，因此在绝大多数情况下，贪心算法无法保证全局最优解。然而，在某些特定的问题上，贪心算法可以巧妙地利用其简单高效的特点，找到全局最优解。例如，在某些结构简单的问题中，局部最优解的累积正好构成了全局最优解。

贪心算法的另一个显著优点是，它的时间复杂度和空间复杂度相对较低。由于不需要像动态规划那样保存和重用中间计算结果，贪心算法省去了对历史数据的存储操作，从而减少了空间占用。同时，贪心算法的决策过程相对直接，不需要对先前结果进行反复检查，这使其时间复杂度也随之降低。因此，在某些需要快速求解且对精确性要求不高的场景下，贪心算法是一个较好的选择。

通过对时间复杂度和空间复杂度的对比可以看出，虽然贪心算法在解决很多问题上效率更高，但其获得的解往往并不是全局最优。在实际应用中，需要根据问题的具体性质来权衡算法的效率与精度。

一个经典的贪心算法应用场景是找零问题。假设你是一家小店的店主，手头拥有面值为 1 元、5 元、10 元、20 元和 50 元的纸币和硬币。现在有一位顾客购买了价值 93 元的商品，并支付了 150 元。如何找零，才能使纸币和硬币的数量最少？问题的核心在于如何通过面值最大的纸币或硬币尽可能多地减少剩余找零的金额。

在这种场景下，贪心算法会首先选择面值最大的纸币，即 50 元，然后从剩余的金额中继续选择面值最大的纸币或硬币，具体步骤如下：支付 150 元，扣除商品金额 93 元，剩余找零为 57 元。首先，选择一张 50 元的纸币，此时剩余金额为 7 元；接着，选择一张 5 元的纸币，剩余金额为 2 元；最后，选择两枚 1 元的硬币，正好将 57 元全部找出。通过这种方式，贪心算法能够迅速地求解出一个局部最优的找零方案，即找零的纸币和硬币数量最少，如图 5-6 所示。

假设有 5 种面额的硬币或纸币：1 元、5 元、10 元、20 元和 50 元。现在需要找零的金额为 57 元，目标是使用尽可能少的纸币和硬币完成找零任务。在这种情况下，贪心算

法是一种有效的策略，因为它每步都选择当前剩余金额下面值最大的货币，确保在局部最优的基础上减少剩余的找零金额。

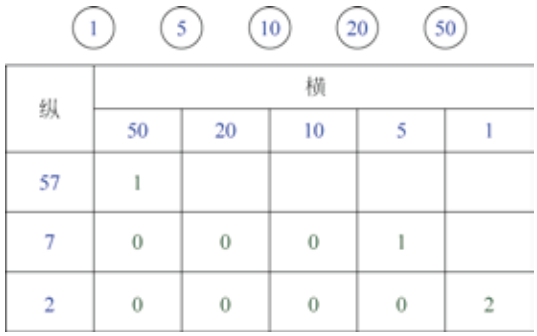


图 5-6　贪心算法示意图

首先，将 57 元与最大面值的 50 元进行比较。由于 57 元大于 50 元，因此可以选择一张 50 元纸币作为找零的一部分，此时剩余金额为 7 元。接下来，继续对剩余的 7 元进行处理。7 元小于 50 元、小于 20 元，也小于 10 元，因此这些面值的纸币或硬币都不适用。接着，7 元与 5 元进行比较，发现 7 元大于 5 元，因此选择一张 5 元纸币，剩余金额减至 2 元。现在剩下的 2 元继续与已有的面额进行比较。2 元小于 50 元、20 元、10 元和 5 元，但可以用 1 元的硬币或纸币进行找零。由于 2 元等于两枚 1 元的硬币之和，因此可以直接使用两枚 1 元硬币完成找零任务。

最终，经过上述贪心算法的选择，找零的结果是一张 50 元纸币、一张 5 元纸币和两枚 1 元硬币。这种策略通过逐步减少剩余金额，确保每步都使用面值最大的货币，从而在有限的找零次数内完成任务。

硬币找零问题的代码如下：

```
// 第 5 章 / 硬币找零 .py
def min_coins_change(amount, coins):
    # 初始化一个无限大的数组来保存每个金额所需的最少纸币或硬币数
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # 当金额为 0 时不需要任何纸币或硬币

    # 动态规划填充 dp 数组
    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    # 如果 dp[amount] 仍然是初始值，则说明无法凑齐金额
    if dp[amount] == float('inf'):
        return None
    else:
        return dp[amount]

def find_min_change(total_payment, item_price, coins):
    # 计算需要找零的金额
    change_amount = total_payment - item_price

    # 调用 min_coins_change 函数计算最少纸币或硬币数
    min_coins = min_coins_change(change_amount, coins)

    # 如果无法凑齐找零金额，则返回错误信息
```

```
        if min_coins is None:
            return "无法找零"

        # 构造找零方案
        change = []
        remaining = change_amount
        for coin in sorted(coins, reverse=True):
            while remaining >= coin:
                change.append(coin)
                remaining -= coin

        return change

# 定义店里有的纸币和硬币面值
coins = [50, 20, 10, 5, 1]

# 顾客支付的金额和商品的价格
total_payment = 150
item_price = 93

# 找出最少找零的纸币和硬币组合
min_change = find_min_change(total_payment, item_price, coins)

# 输出结果
if isinstance(min_change, list):
    print(f"找零方案：{min_change}，共需要{len(min_change)}张纸币或硬币。")
else:
    print(min_change)
```

5.2.2 贪心算法代码写作指导

要写好贪心算法，首先需要对问题的条件和约束有一个深刻的理解。贪心算法的核心思想是通过局部最优解逐步构建问题的解，但它并不总能保证全局最优解，因此在决定使用贪心算法之前，必须评估问题是否适合这种方法。对于可以通过局部最优解逐步推导出全局最优解的问题，贪心算法是非常高效的选择。

在明确问题适合使用贪心算法后，下一步是设定合理的贪心策略。贪心策略的选择必须符合问题的特点和规则。有时，最直观的策略可能并不是最优的。例如，在某些问题中，惯性思维可能会让我们误以为某个显而易见的决策是最佳的，但实际上正确的贪心策略可能与直觉相反。在短视频平台中经常会看到一些比赛，规则是最后骑自行车到达终点的人获胜，这时贪心策略不应是追求最快速度，而是应考虑减缓速度，确保最后到达终点，因此在确定贪心策略时必须深思熟虑，避免局限于简单的直觉判断。

在贪心策略明确后，接下来便是设计和实现算法的步骤。在编写代码时，首先要确保算法的正确性，即能够按照预期解决问题，不要报错，这是基本的要求。其次，还需要优化代码的时间复杂度和空间复杂度，确保算法不仅能正确运行，而且在大规模数据下依然

高效。这要求开发者对算法的性能瓶颈有清晰的认识，并采取适当的优化措施。

考虑一个典型问题：给定一个非负整数数组，每个元素代表当前位置可以跳跃的最大距离。问题是判断是否能够从数组的第 1 个位置到达最后一个位置。以数组［2，3，5，7，6，1，3］为例，使用贪心算法能够高效地解决这个问题。

解决的基本思路是从第 1 个位置出发，不断地更新当前能够到达的最远位置。如果最终能够到达数组的最后一个位置，则返回值为 True，否则返回值为 False。贪心算法的核心在于维护一个变量 max_reachable，用于记录当前能够跳跃到的最远位置。在遍历数组时，检查当前位置是否超出了 max_reachable 的范围。如果当前位置 i 超出了最远可达位置 max_reachable，则意味着无法到达当前位置，进而也无法到达最后一个位置，因此返回值为 False。

举个例子，在第 1 个位置时，索引的值为 0，最远可以到达的地方为 2，即 i 的值为 0，而 max_reachable 的值为 2，而题干中要求每个元素代表当前位置可以跳跃的最大距离，意味着每个值表示最大可以跳跃的距离，当然跳跃距离可以小于最大跳跃距离。

max_reachable 的值在算法执行过程中不断更新，表示在当前状态下能够到达的最远位置。对于每个数组元素，计算从当前位置 i 可以跳跃到的最远索引值。如果当前位置在 max_reachable 之内，则将 max_reachable 的值更新为 max（max_reachable，i+nums［i］），即以确保能够反映从当前位置可以达到的最远距离。其中，max_reachable 表示的含义是当前已经过位置中可以到达的最远距离，而 i+nums［i］表示的含义是当前位置可以跳跃到的距离。遍历数组完成后，如果 max_reachable 大于或等于数组的最后一个位置（索引 $n-1$），则说明可以到达最后一个位置，返回值为 True，否则说明无法到达，返回值为 False。

使用贪心算法求解数组跳越问题的代码如下：

```
// 第 5 章 / 数组跳跃问题 .py
def can_jump(nums):
    max_reachable = 0        # 当前能够到达的最远位置
    n = len(nums)

    for i in range(n):
        if i > max_reachable:
            return False     # 无法到达当前位置
        max_reachable = max(max_reachable, i + nums[i])
                             # 更新当前能够到达的最远位置
        if max_reachable >= n - 1:
                             # 如果最远位置能够到达数组的最后一个位置，则返回值为 True
            return True

    return False

# 测试
nums = [2, 3, 5, 7, 6, 1, 3]
print(can_jump(nums))        # 输出 True
```

5.2.3 贪心算法实际应用介绍

集合覆盖问题（Set Covering Problem，SCP）是组合优化问题中的经典范例，被广泛地应用于工业、物流、通信等领域。其核心是如何在资源有限的情况下，通过最少的选择覆盖所有必需的目标。在实际应用中，例如广告投放、传感器布置、设施选址等，集合覆盖问题都扮演着关键角色。

假设目前面临一个广播台覆盖问题：有若干个广播台，每个广播台都覆盖不同的地区，并且各广播台的覆盖范围可能存在交叉重叠的情况，如何用最少数量的广播台，确保覆盖所有地区。此问题在本质上是一个集合覆盖问题，其中每个广播台的覆盖范围对应一个集合，而需要覆盖的所有地区则对应全集。希望通过贪心算法找到一个近似解，即在每步选择当前能覆盖最多未覆盖地区的广播台，直到所有地区都被覆盖。

从题目描述可以得知，这道题是一个典型的贪心算法问题，想要通过最少数量的广播台达到最大范围的覆盖，每步都尽可能使一个广播台覆盖到更多的范围，从而使最少数量的广播台覆盖到最大范围的地区。

假设有以下地区需要覆盖：地区 1 包含 {'北京','上海','天津'}，地区 2 包含 {'广州','北京','深圳'}，地区 3 包含 {'成都','上海','杭州'}，地区 4 包含 {'上海','天津'}，地区 5 包含 {'杭州','大连'}。每个广播台的覆盖范围如下：广播台 1 覆盖 {'北京','上海','天津','广州'}，广播台 2 覆盖 {'北京','深圳','杭州'}，广播台 3 覆盖 {'成都','上海','杭州'}，广播台 4 覆盖 {'上海','天津','大连'}。

在使用贪心算法时，首先选择能够覆盖最多未覆盖地区的广播台。在第 1 步中，广播台 1 覆盖了 {'北京','上海','天津','广州'} 共 4 个地区，因此是最优选择。接着，将广播台 1 覆盖的地区从未覆盖的地区集合中移除，并将广播台 1 纳入最终的解决方案。在接下来的步骤中，继续选择能覆盖最多剩余未覆盖地区的广播台。以此类推，直到所有地区都被覆盖。

根据上述方法，经过多次迭代，最终得到的最优方案为首先选择广播台 1，覆盖 {'北京','上海','天津','广州'}；接着选择广播台 2，覆盖 {'北京','深圳','杭州'}；最后选择广播台 3，覆盖剩下的 {'成都'}，因此覆盖所有地区最少需要 3 个广播台。

使用贪心算法求解集合覆盖问题的代码如下：

```
// 第 5 章 / 集合覆盖问题 .py
def greedy_set_cover(states_needed, stations):
    final_stations = set()

    while states_needed:
        best_station = None
        states_covered = set()

        for station, states_for_station in stations.items():
            covered = states_needed & states_for_station
```

```
                if len(covered) > len(states_covered):
                    best_station = station
                    states_covered = covered

            if best_station:
                final_stations.add(best_station)
                states_needed -= states_covered

        return final_stations

states_needed = {'北京', '上海', '天津', '广州', '深圳', '成都', '杭州', '大连'}
stations = {
    '广播台 1': {'北京', '上海', '天津', '广州'},
    '广播台 2': {'北京', '深圳', '杭州'},
    '广播台 3': {'成都', '上海', '杭州'},
    '广播台 4': {'上海', '天津', '大连'}
}

# 调用贪心算法求解
selected_stations = greedy_set_cover(states_needed, stations)
print("最终选择的广播台 :", selected_stations)
```

在此代码实现中，greedy_set_cover( ) 函数的设计目标是通过贪心算法解决集合覆盖问题。该函数接受两个参数，分别是 states_needed 和 stations。其中，states_needed 参数要求是一个集合，包含需要被覆盖的所有地区；stations 参数格式要求是一个字典，其中的键为广播台的名称，值为该广播台能够覆盖的地区集合。

函数的工作原理基于贪心策略，即在每步选择当前能够覆盖最多未覆盖地区的广播台。首先，通过遍历每个广播台来计算其能够覆盖的未覆盖地区的数量。在每次迭代中，函数会选择覆盖最多未覆盖地区的广播台，并将其覆盖的地区从 states_needed 变量中移除。这确保了每次迭代后，未覆盖的地区逐渐减少，直到所有地区都被覆盖。

5.3　哈希算法

哈希算法是多个加密算法的统称，其核心功能是将任意长度的输入数据，通过特定的规则转换为固定长度的输出，这个输出称为哈希值。无论输入数据的长度如何，哈希算法都会生成相同长度的结果。哈希算法被广泛地应用于数据加密、数字签名、数据校验和密码存储等场景。哈希算法由于加密过程不可逆，无法通过哈希值还原原始输入，因此更适合用于验证数据完整性和存储敏感信息，而非用于加密传输。

在实际应用中，哈希算法不仅可以确保数据在传输过程中的完整性，还通过其不可逆性增强了数据安全性。例如，系统通常存储的是用户密码的哈希值，而非密码本身。当用户登录时，输入的密码会被哈希处理并与存储的哈希值进行比对，确保安全性和隐私性。

5.3.1 图解哈希算法

哈希算法在数据加密中有着重要的应用，它的不可逆性确保了数据的安全性。在加密过程中，输入的数据通过特定的哈希函数处理，转换为一个固定长度的哈希值。该哈希值通常表现为一个整数，并且与原始数据无直接关联。这一特性确保了即使获取哈希值，也无法通过逆向计算恢复出原始数据，因此大大地提升了数据的安全性。

哈希算法通常处理的输入数据集包括数字和字符串两类。无论输入数据多么复杂，经过哈希函数的处理后其结果始终是定长的输出，而哈希算法的设计目标之一就是尽可能地确保不同的输入产生不同的哈希值，即最大限度地减少哈希冲突。虽然哈希值不会与原始输入直接相关，但其一致性和高效性使它在数据验证和存储等应用中得到了广泛使用，如图 5-7 所示，可以进一步理解哈希算法的工作原理及其如何有效地转换输入数据。

在 Python 语言中，哈希算法的实现主要依赖于内置函数 hash()。通过调用该函数并传入相应的数据，Python 能够迅速地返回对应的哈希值。这种实现方式不仅简化了哈希值的计算过程，还提升了程序的执行效率。对于常见的数据类型，例如整数和浮点数，hash() 函数直接返回其哈希值。

但是需要注意的是，Python 语言中的 hash() 函数主要用于数据结构（例如集合、字典等）的快速查找和数据对比，而不是为了提供密码学级别的安全性，因此与密码学中的哈希函数不同，Python 语言中的哈希函数并没有严格的不可逆性和抗碰撞性要求。它的目标是简化计算并尽量减少哈希冲突，从而保证在常规使用场景下的性能表现。例如，对于整数和浮点数输入，hash() 函数会尽量快速返回与输入直接相关的哈希值，而不是进行复杂的加密处理，如图 5-8 所示，展示了 hash() 函数在 Python 语言中对数字类型进行哈希运算的具体效果。

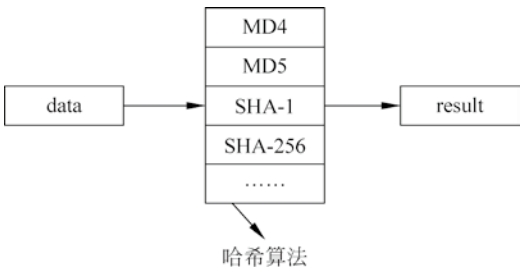


图 5-7 哈希算法示意图

```
nums = 3.44
keys = hash(nums)
print(keys)
```

```
101457092405402521 9
```

图 5-8 哈希函数整数演示

在 Python 语言中，对于整数输入，hash() 函数会直接返回与该数字本身紧密相关的哈希值。这种设计不仅简化了哈希运算的实现，还优化了基于哈希的常见数据结构（例如哈希表）的性能。在哈希表中，由于快速查找操作依赖于哈希计算，因此将整数和浮点数的哈希值保持与原始值直接关联能够显著地提升数据存储和检索的速度。

Python 语言中的哈希实现方式特别有利于处理简单数值类型。由于整数和浮点数在许

多算法中经常作为键值或索引，所以保持哈希值与原始输入的直接关系可使哈希表在进行查找、插入等操作时更加高效。这种高效性对于处理大量数据的场景尤其重要，通过减少不必要的计算步骤，hash( ) 函数保证了常见数值类型的快速映射，如图 5-9 所示。此图进一步地展示了这一原理在具体应用中的效果。hash( ) 函数使程序在处理数值类型时能够以更优的性能表现进行快速查找和数据管理。

对于字符串类型的输入，hash( ) 函数会返回一个相应的哈希值。在 Python 语言中，哈希算法的一个重要特性是对相同的输入总是产生相同的哈希值。这意味着，无论程序何时或在何处调用 hash( ) 函数，只要传入的字符串相同，生成的哈希值就保持一致。这种一致性确保了查找、插入和删除操作在不同时间和环境下都能获得相同的结果，进而可以验证操作。

此外，字符串的哈希值计算虽然涉及更多字符处理操作，但 Python 内部的哈希函数经过优化后能够快速地处理字符串数据。对于哈希表而言，保持字符串哈希值的稳定性和一致性有助于高效地存储和检索数据。这一机制在处理复杂数据集时提供了可靠性和可预测性，尤其是在需要频繁查找相同数据时，如图 5-10 所示。

```
nums = 100
keys = hash(nums)
print(keys)
```

```
100
```

图 5-9　哈希函数浮点数演示

```
data = str([10,20,30])
keys = hash(data)
print(keys)
print(hash(data))
```

```
7009786208762907193
7009786208762907193
```

图 5-10　哈希函数字符串演示

哈希算法具备一些关键特点。首先是其确定性，对于相同的输入，哈希算法总能够生成相同的哈希值，这使它在数据验证中非常可靠。其次，哈希算法的输出是固定长度的，无论输入数据有多长，生成的哈希值始终保持固定长度。这种固定长度的输出不仅方便存储和传输，还能有效地减少存储空间的使用。再次是其计算速度快，哈希算法通常能够在极短的时间内生成哈希值，这使其非常适合用于高效的数据处理场景。最后，哈希算法的不可逆性确保了无法从生成的哈希值反推出原始数据，这增强了其在数据保护中的安全性。

正是由于这些特性，哈希算法被广泛地应用于数据保护中，然而，它并不直接返回原始数据的哈希值，因为哈希算法的设计初衷并不支持这种操作。哈希值更多地用于数据验证，而非数据传输或还原。在数据完整性验证方面，常见的做法是计算原始数据的哈希值并保存在一个安全的地方。当需要验证数据时，再次计算当前数据的哈希值，并与之前保存的哈希值进行对比，如果二者一致，则说明数据未被篡改。

在密码验证中，哈希算法同样有着重要的作用。当用户输入密码时，系统并不会直接保存或比较原始密码，而是将输入的密码经过哈希处理后，与预先存储的哈希值进行比较。这一过程避免了存储或传输明文密码，显著地提高了系统的安全性。

哈希算法的应用范围非常广泛，涵盖从数据完整性验证到密码存储等多个领域。在接下来的两节中，将详细介绍几种常见的哈希算法，并通过实例展示其在实际开发中的应用。

5.3.2 哈希算法代码写作指导

在 Python 语言中，哈希算法的实现方式主要有 3 种方式：自定义哈希算法、hash( ) 函数和 hashlib 模块。自定义哈希算法允许开发者根据特定需求灵活地进行定制，适用于个性化使用场景；hash( ) 函数则适用于高效生成内置对象的哈希值，常见于集合、字典等数据结构的快速查找中；hashlib 模块提供了多种安全哈希算法，被广泛地应用于数据加密和完整性验证等需要高安全性的场景。开发者可以根据不同的需求选择适合的哈希实现方法。

1. 自定义哈希算法

哈希算法被广泛地应用于数据加密领域，而自定义哈希算法则是根据特定需求设计的一种加密方式。一种常见的思路是将每个字符转换为其对应的 ASCII 码值，在计算机中每个字符都可以用一个唯一的 ASCII 码来表示。通过将输入字符串中的每个字符逐个转换为相应的 ASCII 码值，再将所有这些 ASCII 码值进行累加，就可以得到一个代表该字符串的哈希值。

通过 ASCII 码值自定义哈希算法的实现相对简单。然而，需要注意的是，这种方式虽然能快速地生成哈希值，但在面对复杂的应用场景时，其抗碰撞性和安全性并不足够强大，即不同的输入可能得到相同的哈希值，进而影响系统的可靠性和安全性。与标准化的哈希算法（例如 SHA-256）相比，ASCII 码值自定义算法更适合于简单的应用，尤其是在对安全性要求不高的场景中。因此在实际开发中，选择合适的哈希算法应根据具体应用场景的需求来决定。

自定义哈希算法的代码如下：

```
// 第 5 章 / 自定义哈希算法 .py
def custom_hash(text):
    hashed_value = 0
    for char in text:
        hashed_value += ord(char)
    return hashed_value

text = "Hello, World!"
hashed_text = custom_hash(text)
print("Custom hash of '{}' is: {}".format(text, hashed_text))
```

举一个简单的例子，对于字符串 abc，各字符的 ASCII 码值分别为字符 a 的值是 97，字符 b 的值是 98，字符 c 的值是 99。将这些字符的 ASCII 码值相加，得到的总和是 294。然而，另一个字符串 bca 的各字符 ASCII 码值分别为字符 b 的值是 98，字符 c 的值是 99，

字符 a 的值是 97。虽然字符的顺序不同，但它们的 ASCII 码值相加后的结果也是 294，因此在这种简单的哈希算法中，两个不同的字符串 abc 和 bca 产生了相同的哈希值，这就是所谓的碰撞现象。

在实际的哈希算法中，避免碰撞是至关重要的。哈希函数一旦发生碰撞，就可能会引发严重的问题，尤其是在数据完整性和信息安全领域。例如，数据碰撞可能会导致数据库中的数据损坏，或在加密系统中造成潜在的安全漏洞。因此为了降低碰撞的概率，现代的哈希算法（例如 SHA-256 等）采用了更复杂的数学运算和结构设计，以确保即使输入稍有不同，输出的哈希值也会发生显著变化，从而大幅度地降低碰撞的可能性。

2. hash() 函数

在 Python 编程语言中，hash() 函数是一个用于计算哈希值的内置函数，能够直接对传入的对象生成对应的哈希值。在图 5-8~ 图 5-10 中，展示了该函数在处理字符串和数值时的不同表现。通过这些例子可以清楚地看到，Python 语言内置的 hash() 函数可以对多种类型的数据进行哈希运算，包括字符串、整数、浮点数等。这使哈希函数在需要快速比较、查找或者存储数据的场景中具有重要的应用。

仍需再次强调的是，尽管在同一次程序运行中，对于相同的字符串，hash() 函数生成的哈希值是固定的，然而在不同的程序运行中，生成的哈希值可能会有所不同。产生这一现象的原因是，Python 的 hash() 函数在运行时为了增强安全性，引入了一个随机化机制。具体来讲，在 Python 编译器启动时会生成一个随机的前缀或后缀（通常称为 secret prefix/suffix），并将其附加到被哈希的对象上，这种设计有效地防止了某些基于哈希冲突的攻击，例如哈希碰撞攻击。

因此，虽然在开发和调试中，可以使用 hash() 函数快速地获得对象的哈希值，但应牢记这一机制，尤其是在编写需要多次运行并依赖于哈希值一致性的程序时。对于需要跨运行保持哈希值稳定的情况，例如在数据库索引或缓存机制中，可以考虑使用更加稳定的哈希函数，例如 hashlib 模块提供的 md5() 函数或 sha256() 函数等，这些函数能够为输入生成固定的跨运行一致的哈希值。

3. hashlib 模块

Python 语言的内置 hash() 函数在哈希值的生成过程中具有一定的随机性，主要体现在它的哈希值在不同运行之间可能不一致。这种随机性源自 Python 语言的安全机制，而我们无法控制 hash() 函数内部使用的具体算法，因此尽管 hash() 函数在某些场景下足够用，但对于要求精确、一致的哈希值场景，它可能并不合适。

为了解决这一问题，Python 语言提供了 hashlib 模块。与 hash() 函数不同，hashlib 模块允许开发者选择并使用特定的哈希算法，这些算法提供了稳定且一致的输出，特别适用于加密、数据校验和数字签名等场景。hashlib 模块中包含了多种常用的哈希函数，例如 MD5、SHA-1、SHA-224、SHA-256 等。用户可以根据具体需求选择适合的哈希算法，并且这些算法在不同的程序运行中对相同的输入始终会生成相同的哈希值，这使其更加适合在对数据一致性要求高的场景中使用。

无论选择哪种哈希算法，使用 hashlib 模块的流程通常遵循以下步骤。首先，需要导入 hashlib 模块。接着，选择所需的哈希算法，例如 MD5 或 SHA-256。以 MD5 为例，如图 5-11 所示，如果需要对字符串进行 MD5 哈希运算，则第 1 步是导入 hashlib 模块，接下来调用模块中的 md5( ) 函数。调用哈希函数后，将需要哈希的字符串传递给函数并进行运算，即可得到该字符串的哈希值。

```
#导入 hashlib 模块
import hashlib

#输入文本
text = "我希望这本书可以帮到你！"

#选择 md5 函数并对文本进行哈希处理
md5_hash = hashlib.md5()
md5_hash.update(text.encode('utf-8'))

#获取哈希值的十六进制表示
hashed_text = md5_hash.hexdigest()

#输出结果
print("MD5 Hash of '{}' is: {}".format(text, hashed_text))
```

```
MD5 Hash of '我希望这本书可以帮到你！' is: 4dd02e16f16a8217876c35c9cfc1b000
```

图 5-11　哈希算法应用

在使用 hashlib 模块时，选择适合的哈希算法至关重要。MD5 虽然较为常见，但在现代加密领域已不被认为是安全的，因为它在面对碰撞攻击时较为脆弱，因此如果安全性是首要考量，则建议使用更强大的算法，例如 SHA-256 或更高级别的哈希函数。这些算法可以为系统提供更强的抗碰撞能力，并确保数据的安全性。

5.3.3　哈希算法实际应用介绍

哈希算法主要应用在数据验证操作，尤其是在密码验证和文本完整性校验等方面。在这些场景中，哈希算法通过将原始数据转换为固定长度的哈希值来确保数据的保密性和完整性，因此在实际开发中，哈希算法常常被用于密码验证等安全性相关的操作。

密码验证的基本流程通常包括两个阶段：输入和验证。在用户输入用户名和密码后，系统需要对输入的数据进行进一步处理。接下来，根据操作的不同，系统会选择是对输入的密码进行存储，还是对输入的密码进行验证。在实际应用中，存储密码时并不会直接存储明文密码，而是通过哈希算法对密码进行加密后再存储。这样，即使数据库被泄露，攻击者也无法直接得到用户的明文密码，而只能获取经过哈希处理的值，如图 5-12 所示。

在实际开发中，密码的存储和验证通常通过数据库来实现。然而，出于示例目的，本节的例子不涉及数据库的使用，而是通过列表或字典来简单地模拟数据存储。具体来讲，当用户输入用户名和密码时，系统将这些数据存储在列表或字典中，并对密码进行哈希处理以生成哈希值。由于每次程序运行时数据都会被清空，所以这种方式仅适用于演示和

测试。

在用户输入账号和密码时，通常遵循的一项基本惯例是账号以明文方式显示，而密码则采用暗文显示。明文显示账号的做法主要是为了减少用户输入错误的概率，而将密码以暗文（通常为星号或圆点形式）显示，则能够有效地防止旁观者看到密码内容，从而避免潜在的安全风险，如图 5-13 所示。

图 5-12　哈希算法验证密码操作

图 5-13　账号和密码登录演示

在数据存储成功后，系统通常会弹出一个提示框，提醒用户数据已被成功保存。这个提示框不仅提供了操作的反馈，还能让用户确认当前步骤已经完成，接下来可以进行后续操作，例如数据验证等。提示框的设计应简洁明确，避免用户因不确定操作结果而重复提交数据。在 Python 语言的 GUI 开发中，常用的 Tkinter 库可以实现这种功能，而在 Web 应用中则通常通过 JavaScript 提供类似的反馈。

一旦数据存储成功，临时数据库（或内存中的数据结构）就会记录用户输入的信息。用户可以通过验证操作来检查输入的数据是否与系统记录的数据一致，如图 5-14 所示。

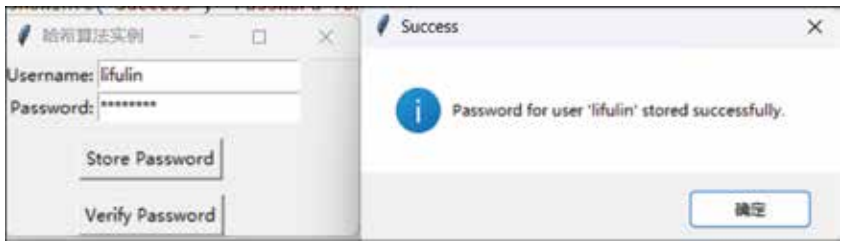


图 5-14　存储成功示意图

在完成数据存储的操作后，系统会进入数据验证阶段。在这一过程中，系统首先会检查临时数据库或存储结构中是否存在用户输入的相关数据。如果找到匹配的用户名，则系统接下来会对用户名对应的密码进行验证，即比对用户输入的密码与存储中记录的哈希值是否一致。

当用户名和密码验证通过后，系统会弹出一个提示框，告知用户验证成功。这一提示框的作用在于向用户提供明确的反馈，确认他们的输入是正确的，并让用户可以继续进行下一步操作。验证过程中的每个步骤，尤其是密码的比对都应保持高效且安全，如图 5-15 所示。

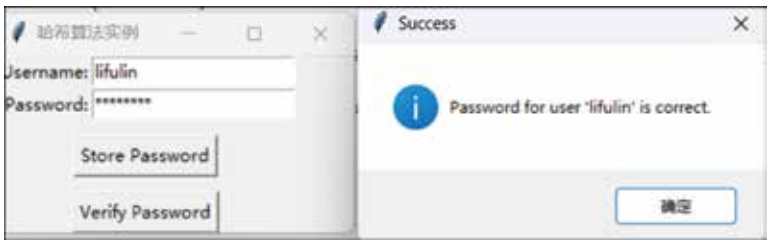


图 5-15　密码验证成功示意图

哈希算法应用于密码验证操作的代码如下：

```
// 第 5 章 / 自定义哈希算法 .py
import hashlib
import tkinter as tk
from tkinter import messagebox

class PasswordManagerUI:
    def __init__(self, master):
        self.master = master
        master.title(" 哈希算法实例 ")

        self.username_label = tk.Label(master, text="Username:")
        self.username_label.grid(row=0, column=0, sticky="e")

        self.username_entry = tk.Entry(master)
        self.username_entry.grid(row=0, column=1)

        self.password_label = tk.Label(master, text="Password:")
        self.password_label.grid(row=1, column=0, sticky="e")

        self.password_entry = tk.Entry(master, show="*")
        self.password_entry.grid(row=1, column=1)

        self.store_button = tk.Button(master, text="Store Password", command=
self.store_password)
        self.store_button.grid(row=2, column=0, columnspan=2, pady=10)

        self.verify_button = tk.Button(master, text="Verify Password", command=
self.verify_password)
        self.verify_button.grid(row=3, column=0, columnspan=2)

        self.password_manager = PasswordManager()

    def hash_password(self, password):
        return hashlib.sha256(password.encode('utf-8')).hexdigest()

    def store_password(self):
        username = self.username_entry.get()
```

```
            password = self.password_entry.get()
            hashed_password = self.hash_password(password)
            self.password_manager.store_password(username, hashed_password)
            messagebox.showinfo("Success", "Password for user '{}' stored
successfully.".format(username))

        def verify_password(self):
            username = self.username_entry.get()
            password = self.password_entry.get()
            hashed_password = self.hash_password(password)
            self.password_manager.verify_password(username, hashed_password)

    class PasswordManager:
        def __init__(self):
            self.passwords = {}

        def store_password(self, username, hashed_password):
            self.passwords[username] = hashed_password

        def verify_password(self, username, hashed_password):
            stored_password = self.passwords.get(username)
            if stored_password:
                if stored_password == hashed_password:
                    messagebox.showinfo("Success", "Password for user '{}' is
correct.".format(username))
                else:
                    messagebox.showerror("Error", "Incorrect password for user
'{}'.".format(username))
            else:
                messagebox.showerror("Error", "User '{}' not found.".format
(username))

    def main():
        root = tk.Tk()
        app = PasswordManagerUI(root)
        root.mainloop()

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

上述代码实现了一个简单的密码管理器用户界面（UI），允许用户通过输入框输入用户名和密码，并提供存储和验证密码的功能。在用户输入用户名和密码后，可以选择通过相应的按钮将密码存储到密码管理器中。

整个 UI 界面由 Python 语言的 Tkinter 库构建，Tkinter 是一个常用的图形用户界面库，它提供了创建窗口、按钮、文本框等组件的便捷方法。通过这种方式，用户可以与系统简单而直观地进行交互。

5.4 数值分析算法

数值分析是一门应用数学领域，专注于开发和分析解决数学问题的数值方法，其主要目标是通过计算机来近似解决数学问题，这些问题可能无法或很难通过解析方法求解。数值分析方法被广泛地应用于科学、工程、经济学和许多其他领域，涉及从微分方程求解到优化问题等各种数学问题的近似解决方案。

5.4.1 插值和拟合

插值与拟合是数值分析和数据处理中的重要工具，它们通过不同的数学方法，在已知数据点之间或数据点的趋势上构造函数模型。插值旨在找到一个精确通过所有已知数据点的函数，用于预测这些点之间的值，而拟合则侧重于通过一个模型来近似数据，允许有一定的误差，以捕捉数据的整体趋势。

1. 插值算法

插值是指在已知数据点之间的数据点上构造函数的过程，其目标是在这些数据点之间寻找一个函数，使该函数通过所有已知数据点。插值方法适用于希望在已知数据点之间获得精确的函数值时，例如在实验测量值之间进行预测。常用的插值算法主要包含拉格朗日插值法、牛顿插值法和样条插值。

1）拉格朗日插值法

拉格朗日插值法用于在给定一组离散数据点的情况下估计一个函数的值。这种方法的核心在于构建拉格朗日多项式，它将已知的数据点与相应的函数值相结合，形成一个连续的多项式函数，使该函数在每个数据点上都能够精确地匹配给定的函数值。具体而言，拉格朗日插值多项式是以数据点的横坐标为变量，利用多项式的线性组合来实现对函数的插值。

在构建拉格朗日插值多项式时，对于每个数据点都会计算一个对应的基函数，基函数的特性在于它们在特定的数据点处取值为 1，而在其他的数据点处取值为 0。通过将这些基函数与相应的数据点函数值相乘，并将所有的乘积相加，可以得到整体的插值多项式。在 Python 语言中，可以利用 NumPy 和 SciPy 等库实现拉格朗日插值，进而为各种应用场景提供高效的数值计算支持。

拉格朗日多项式公式表达为

$$L(x)=\sum_{i=1}^{n} y_i * l_i(x) \tag{5-1}$$

其中，$L_i(x)$ 是拉格朗日基函数，定义为

$$l_i(x)=\prod_{\substack{j=1\\ j\neq 1}}^{n} \frac{x-y_j}{x_i-x_j} \tag{5-2}$$

这些基函数有一个特性，即当 $x=x_i$ 时，$l_i(x)=1$，而在其他数据点上 $x=x_j$（其中 $j\neq j$）时，$l_i(x)=0$。这确保了拉格朗日插值多项式通过每个数据点，因此在这些点上与给定的数据完全一致。

拉格朗日函数的代码可视化过程首先需要定义一组坐标点，这些点将作为插值的基础。在本例中，选择了 5 个坐标点，分别为（1，2）、（2，3）、（3，5）和（4，7）。同时在代码中自定义一个拉格朗日函数，该函数负责计算并生成对应的拉格朗日插值多项式。多项式将通过已定义的 5 个数据点，并保证在这些点上的插值结果与原始函数值完全一致。随后，根据计算得到的拉格朗日插值多项式，利用可视化库（例如 Matplotlib 库等）绘制出插值曲线，如图 5-16 所示。插值曲线在所选的数据点处与原始函数值重合，而在其他区域则展现了通过多项式插值得到的函数估计。

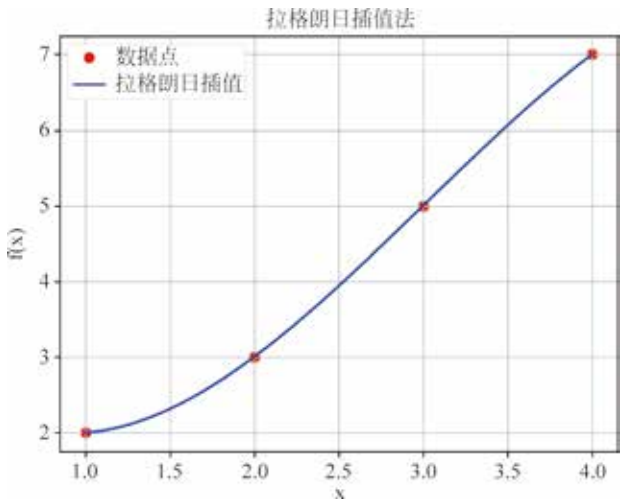


图 5-16　拉格朗日插值法代码可视化

拉格朗日插值法的代码如下：

```
// 第 5 章 / 拉格朗日插值方法 .py
import numpy as np
import matplotlib.pyplot as plt

# 定义拉格朗日函数
def lagrange(x, points):
    result = 0
    n = len(points)
    for j in range(n):
        term = points[j][1]
        for k in range(n):
            if k != j:
                term *= (x - points[k][0]) / (points[j][0] - points[k][0])
```

```
            result += term
        return result

# 生成数据点
points = [(1, 2), (2, 3), (3, 5), (4, 7)]

# 绘制原始数据点
x_vals = [p[0] for p in points]
y_vals = [p[1] for p in points]
plt.scatter(x_vals, y_vals, color='red', label='Data Points')

# 计算并绘制拉格朗日函数
x_range = np.linspace(min(x_vals), max(x_vals), 100)
y_range = [lagrange(x, points) for x in x_range]
plt.plot(x_range, y_range, color='blue', label='Lagrange Polynomial')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Lagrange Interpolation')
plt.legend()
plt.grid(True)
plt.show()
```

2）牛顿插值法

牛顿插值法是一种重要的数值插值方法，该方法的核心思想是利用已知数据点的函数值，通过递归计算差商的方式，逐步构造出高次插值多项式。这种差商的计算不仅能提高插值的效率，还能增强插值的数值稳定性。

在牛顿插值法中，首先需要选择一组数据点，并计算这些点的函数值。接着，通过构造差商表，利用数据点之间的关系逐步获得差商值，差商值反映了数据点的变化趋势。通过差商，构造出牛顿插值多项式，该多项式不仅能在每个已知数据点上精确匹配对应的函数值，还能够在插值区间内提供良好的函数估计。

假设现在有一组数据点，分别是 $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$。其中，$x_i$ 是已知的节点，y_i 是相应节点的函数值。牛顿插值法多项式 $P_n(x)$ 可以通过以下递归函数得到：

$$P_n(x)=f(x_0)+(x-x_0)f[x_0, x_1]+(x-x_0)(x-x_1)f[x_0, x_1, x_2]+\cdots+(x-x_{n-1})f[x_0, x_1, \cdots, x_n] \tag{5-3}$$

其中，$f[x_i, x_{i+1}, \cdots, x_j]$ 表示差商，可以通过以下递归公式计算得到：

$$f[x_i, x_{i+1}, \cdots, x_j]=\frac{f(x_{i+1}, x_{i+2}, \cdots, x_j]-f(x_i, x_{i+1}, \cdots, x_{j-1}]}{x_j-x_i} \tag{5-4}$$

利用随机数生成函数生成 10 组横坐标和纵坐标，这些数据点将作为插值的基础。在 Python 中，常用的库（例如 NumPy 和 Random）可以实现这一过程，以确保生成的点在数

值上具有一定的随机性和分布特性。一旦数据点生成完成，接下来便可利用牛顿插值法构造插值多项式。该方法通过递归计算差商，将已知数据点的函数值转换为多项式的系数。最后，利用可视化工具将构造出的牛顿插值多项式绘制出来，如图 5-17 所示。

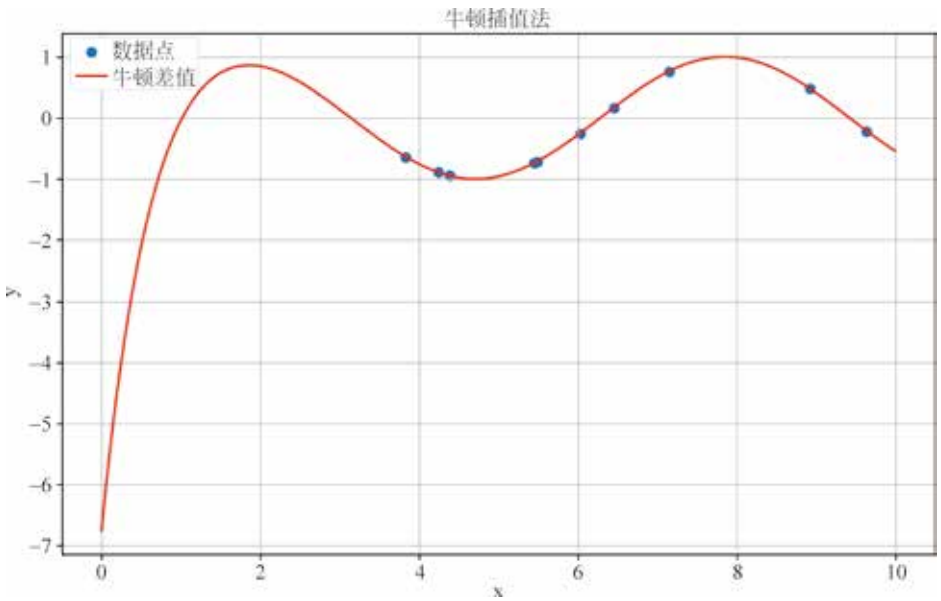


图 5-17　牛顿插值法代码可视化

牛顿插值法的代码如下：

```
// 第 5 章 / 牛顿插值方法 .py
import numpy as np
import matplotlib.pyplot as plt

# 计算差商
def divided_difference(x, y):
    n = len(x)
    F = np.zeros((n, n))
    F[:,0] = y
    for j in range(1, n):
        for i in range(j, n):
            F[i, j] = (F[i, j-1] - F[i-1, j-1]) / (x[i] - x[i-j])
    return F

# 计算插值多项式
def newton_interpolation(x, y, xi):
    n = len(x)
    F = divided_difference(x, y)
    yi = np.zeros(len(xi))
```

```
        for i in range(n):
            term = F[i,i]
            for j in range(i):
                term *= (xi - x[j])
            yi += term
        return yi

    # 生成随机数据点
    np.random.seed(0)
    x = np.sort(np.random.uniform(0, 10, 10))
    y = np.sin(x)

    # 生成插值点
    xi = np.linspace(0, 10, 1000)

    # 计算插值结果
    yi_newton = newton_interpolation(x, y, xi)

    # 绘图
    plt.figure(figsize=(10, 6))
    plt.scatter(x, y, label='Data Points')
    plt.plot(xi, yi_newton, label='Newton Interpolation', color='red')
    plt.title('Newton Interpolation')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid(True)
    plt.show()
```

3）样条插值法

样条插值法是一种插值方法，用于在已知数据点之间构造光滑曲线的插值函数。它通过将数据点之间的区间划分为小段，并在每个小段内使用低次多项式进行插值，以保证插值函数的光滑性。

给定一组数据点 $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$，其中 x_i 是已知节点，y_i 是相应节点的函数值。样条插值将数据点之间的区间划分为小段，每个小段内使用一个低次多项式来近似描述数据的整体行为。为了保证插值函数的光滑性，通常要求插值函数的一阶导数和二阶导数在节点处连续。

在本例中，选择使用三次样条插值法，不仅能在每个已知数据点上精确地匹配函数值，还能保证在连接点处的一阶导数和二阶导数的连续性。样条插值的关键在于在每个数据点之间建立一个三次多项式，使这些多项式在数据点处平滑连接。

一旦数据点生成并建立了样条插值，便可使用可视化工具绘制插值结果，如图 5-18 所示。此图展示了随机生成的 10 个数据点及通过样条插值法得到的平滑曲线。这条曲线不仅通过每个数据点，同时还展现了插值方法在数据集中的光滑过渡特性。

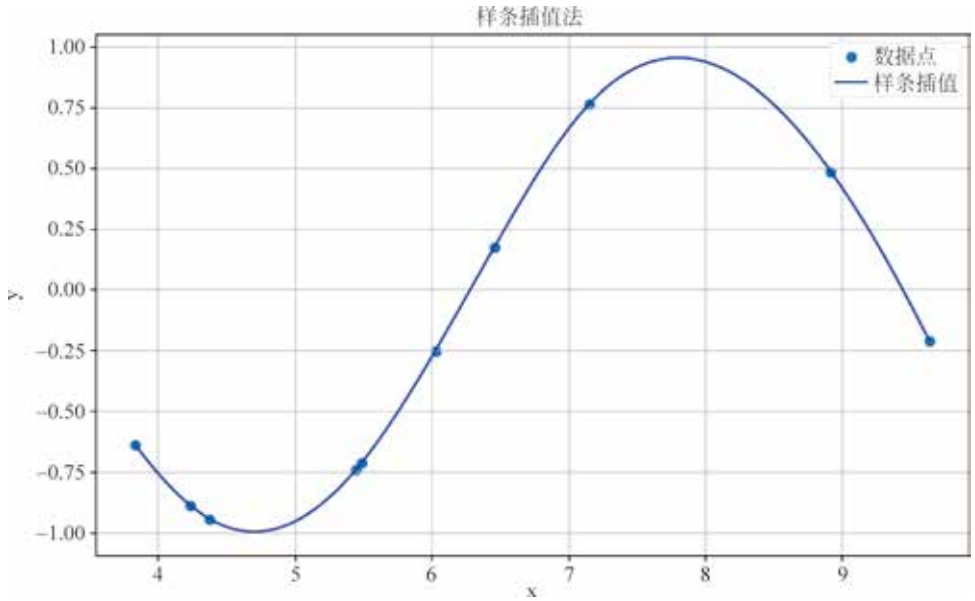


图 5-18　样条插值法代码可视化

样条插值法的代码如下：

```
// 第 5 章 / 样条插值方法 .py
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# 生成随机数据点
np.random.seed(0)
x = np.sort(np.random.uniform(0, 10, 10))
y = np.sin(x)

# 样条插值
def spline_interpolation(x, y, xi):
    spline = interp1d(x, y, kind='cubic')
    return spline(xi)

# 生成插值点
xi = np.linspace(min(x), max(x), 1000)

# 计算插值结果
yi_spline = spline_interpolation(x, y, xi)

# 绘图
```

```
plt.figure(figsize=(10, 6))
plt.scatter(x, y, label='Data Points')
plt.plot(xi, yi_spline, label='Spline Interpolation', color='blue')
plt.title('Spline Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

2. 拟合算法

拟合算法是一种数据分析技术，用于找到一组数据点之间的关系，并建立一个数学模型来描述这种关系。这种关系可以是线性、非线性、多项式或其他形式的。拟合算法的目标是通过调整模型的参数，使模型能够尽可能地逼近或拟合数据点，以便对未知数据进行预测或估计。常见的拟合函数大致有 4 种，分别是线性回归、多项式回归、非线性最小二乘拟合、核回归等。

1）如何拟合

拟合函数的多项式是开始准备拟合时首先确定下来的。可能有的读者会想到在 5.4.1 节中提到的插值方法，插值方法可以使函数曲线过直角坐标系中的每个点，是否可以用插值的方法得到函数，进而作为拟合的函数呢?

答案是不可以的，插值要求设计的曲线过每个函数点，适用于对现有数据点的精确计算，而拟合则是用函数表达现有数据点的一种趋势，对未来可能存在的函数点进行一些趋势性的预测。

如果在拟合的过程中使用插值函数，则会出现过度拟合的现象，即拟合函数作为预测作用的功能消失了。

拟合函数的选取大致分为两类：第 1 类用于观察散点图的形状和趋势，根据数据的分布情况，可以初步判断采用何种类型的函数进行拟合，例如线性、多项式、指数等。第 2 类用于了解数据所属领域的相关知识，有助于确定拟合函数的类型。例如，在物理学领域，如果已知数据应该遵循某种特定的物理定律，则可以根据这些定律来选择拟合函数。

而判断函数拟合效果的好坏，可以使用均方误差、决定系数、平均绝对误差、残差分析等，当然还有很多种方法可供选择，每次进行验证时，应尽量多选取几种方法联合进行验证，而不要单单只选取一种方法。

2）线性回归

线性回归是一种用于建立变量之间线性关系的拟合算法。假设现在有一辆汽车，它的速度是 30km/h，现在这辆汽车在一条道路上匀速前行，绘制一个时间和距离相关关系的直角坐标系，并在该坐标系上随机生成 20 个点，如图 5-19 所示。

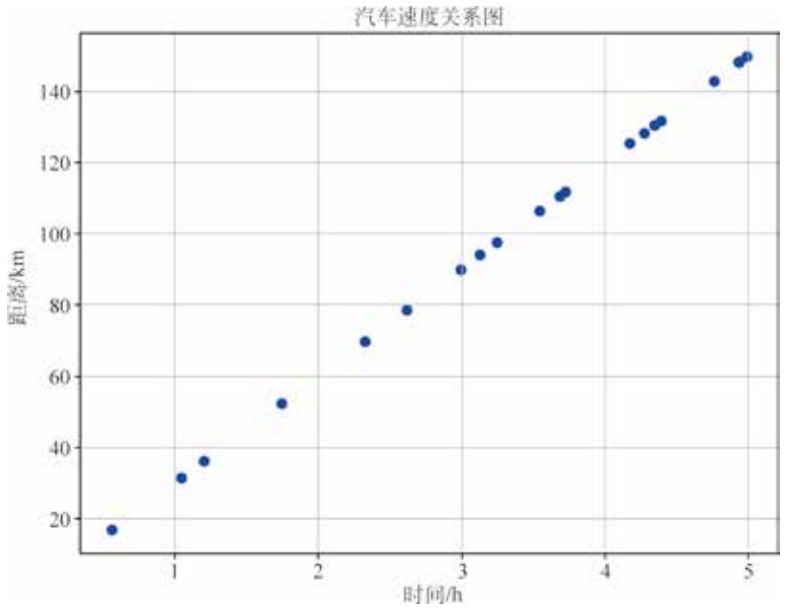


图 5-19　时间和速度关系图

实现汽车以匀速 30km/h 行驶时的时间和距离关系散点图的代码如下：

```
// 第 5 章 / 汽车速度关系图 .py
import matplotlib.pyplot as plt
import numpy as np

# 生成时间数据（小时）
time = np.random.uniform(0, 5, 20)    # 生成 20 个随机时间数据点

# 根据线性关系生成距离数据（千米）
speed = 30                            # 时速 30km/h
distance = speed * time

# 绘制散点图
plt.figure(figsize=(8, 6))
plt.scatter(time, distance, color='blue')
plt.title('Scatter Plot of Time vs Distance')
plt.xlabel('Time (hours)')
plt.ylabel('Distance (kilometers)')
plt.grid(True)
plt.show()
```

在该散点图的基础上，对 20 个数据点进行线性拟合，并绘制拟合图形，输出拟合函数。由图 5-19，同时结合题干，当前是匀速直线前行，当前散点图是一次函数，即线性拟合，如图 5-20 所示。

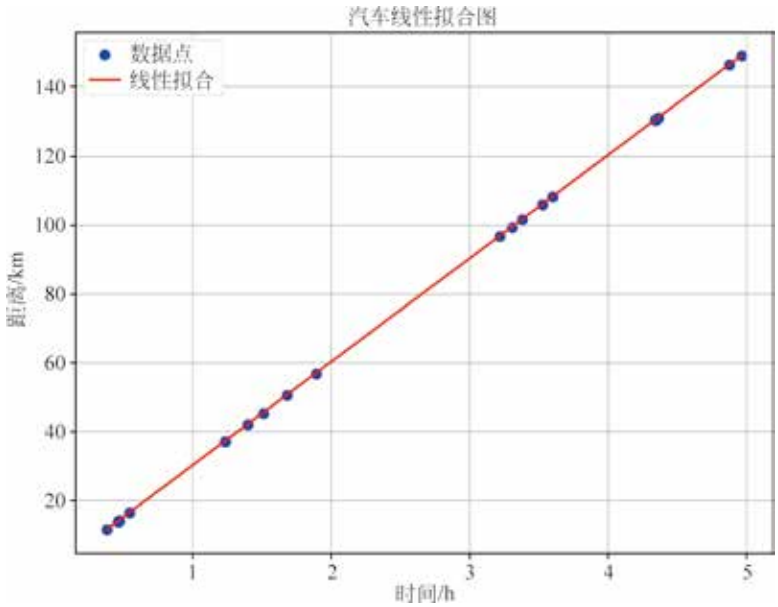


图 5-20　线性拟合图

输出当前直线函数，如图 5-21 所示。

拟合直线公式：y = 30.00x + - 0.00

图 5-21　线性直线关系式

实现汽车以匀速 30km/h 行驶时的时间和距离关系散点拟合图的代码如下：

```
// 第 5 章 / 汽车速度线性拟合 .py
import matplotlib.pyplot as plt
import numpy as np

# 生成时间数据（小时）
time = np.random.uniform(0, 5, 20)      # 生成 20 个随机时间数据点

# 根据线性关系生成距离数据（千米）
speed = 30                              # 时速 30km/h
distance = speed * time

# 进行线性拟合
fit = np.polyfit(time, distance, 1)
slope = fit[0]                          # 斜率
intercept = fit[1]                      # 截距

# 输出拟合直线的公式
print("拟合直线公式：y = {:.2f}x + {:.2f}".format(slope, intercept))

# 计算拟合直线上的点
fit_line_x = np.linspace(min(time), max(time), 100)
fit_line_y = slope * fit_line_x + intercept
```

```
# 绘制散点图和拟合直线
plt.figure(figsize=(8, 6))
plt.scatter(time, distance, color='blue', label='Data')
plt.plot(fit_line_x, fit_line_y, color='red', label='Linear Fit')
plt.title('Scatter Plot of Time vs Distance with Linear Fit')
plt.xlabel('Time (hours)')
plt.ylabel('Distance (kilometers)')
plt.legend()
plt.grid(True)
plt.show()
```

3）多项式回归

在线性回归中，主要进行探讨的是自变量和因变量成线性相关关系的数据，但是现实中很多情况往往并不是线性相关的，例如复习时间和考试成绩等。如果想要探讨这些问题，则需要考虑的因素便会增多。研究一个因变量与一个或多个自变量间多项式的回归分析方法便是多项式回归。

一次函数表达式通常可以用 $y=kx+b$ 来表示，函数绘制出来通常是一条直线，而多项式回归，函数的指数会升高，通常采用二次函数 $y=ax^2+bx+c$，或者三次函数 $y=ax^3+bx^2+cx+d$ 及多次函数等来拟合数据。

多项式回归首先需要生成一些模拟的数据点，这些数据点之间应该具有某些多项式关系。之后需要将一些随机噪声添加到多项式之中，以便更好地模拟真实的数据。随后将所有数据集一分为二，其中 70% 为训练集，30% 为测试集。用训练集训练回归函数，用测试集对训练集得出的函数进行测试。

计算机模拟测试，随机生成多个数据点，其中横轴的范围为 0~5，纵轴的范围为 −1~1，拟合函数的指数为 3 次，最终函数点与函数拟合曲线如 5-22 所示。

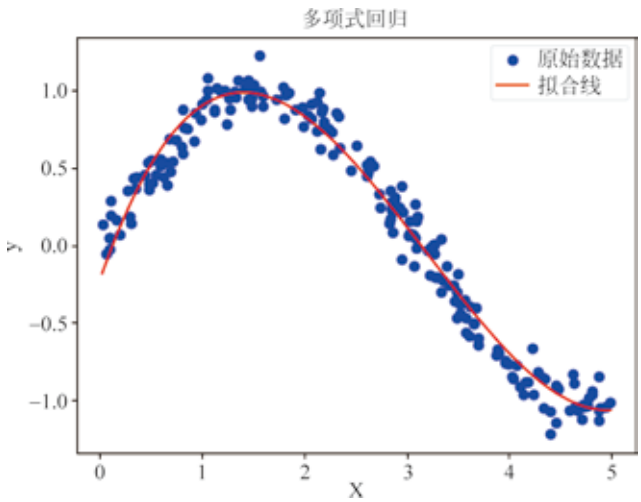


图 5-22　多项式回归拟合函数

多项式回归拟合曲线最终会有一个评估标准，在此次示例中采用均方根误差的方式进行评估，表示观测值与模型预测值之间的差异的平方的平均值的平方根。它的值越小证明模拟预测的效果越好，均方根误差的结果如图 5-23 所示。

图 5-23　均方根误差结果图

多项式归回拟合函数的代码如下：

```
// 第 5 章 / 多项式回归 .py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# 生成模拟数据
np.random.seed(0)
X = np.sort(5 * np.random.rand(200, 1), axis=0)
y = np.sin(X).ravel() + np.random.normal(0, 0.1, X.shape[0])

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 使用多项式特征转换器将特征转换成多项式特征
degree = 3  # 设置多项式的阶数
poly_features = PolynomialFeatures(degree=degree)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

# 训练线性回归模型
model = LinearRegression()
model.fit(X_train_poly, y_train)

# 在训练集和测试集上评估模型
y_train_pred = model.predict(X_train_poly)
train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))

y_test_pred = model.predict(X_test_poly)
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

print(f"训练集 RMSE: {train_rmse}")
print(f"测试集 RMSE: {test_rmse}")

# 可视化拟合结果
plt.scatter(X, y, color='blue', label='Original data')
```

```
    plt.plot(X, model.predict(poly_features.transform(X)), color='red',
    label='Fitted line')
    plt.title('Polynomial Regression')
    plt.xlabel('X')
    plt.ylabel('y')
    plt.legend()
    plt.show()
```

4）非线性最小二乘拟合

非线性最小二乘拟合是一种通过拟合一个非线性模型来逼近数据的方法。与线性最小二乘拟合不同，由于非线性最小二乘拟合的模型函数是非线性的，因此需要使用迭代算法来最小化损失函数，通常采用的是梯度下降法或者其他优化算法。

关于线性和非线性比较简单的判断标准便是看形成的函数曲线是否为一条直线或者线段：如果是，则为线性；如果不是，则为非线性。

非线性最小二乘拟合在给定数据集的情况下，首先需要根据数据点的分布情况选择合适的函数进行拟合，常见的非线性函数包括指数函数、对数函数、多项式函数等。在选择函数模型之后，便需要构建一个描述模型与实际数据之间差异的损失函数，之后利用优化算法来减小损失函数，最后对模型进行评估，如果评估结果可以接受，则可用于数据的预测。

在非线性最小二乘拟合函数中，一个比较常见的案例便是拟合生长曲线，生长曲线描述的是生物体的生长过程。生物体的生长过程一般是非线性的，刚开始生长迅速，随后生长速度渐趋减缓，最终生长速度趋于稳定。

为了拟合生长曲线，可以选定一个生物生长模型来描述生长的过程，其中常见的便是 Gompertz 生长模型。Gompertz 生长模型用于预测生长曲线的回归预测，常见的应用有代谢预测、肿瘤生长预测、有限区域内生物种群数量预测、工业产品的市场预测等。Gompertz 生长模型公式如下：

$$f(x)=K*e^{-e^{-r(x-t)}} \tag{5-5}$$

其中，$f(x)$ 表示生物体在时间 x 的体量；K 表示生长上限，生物在经过无限时间生长后可以达到的最大体量；r 是生长速率，表示生物体的生长速度；t 表示生长延迟时间，表示生物体从开始生长到真正开始迅速增长的时间。

在绘制函数曲线的过程中，首先需要定义函数体，之后加入一定的数据点，并加入一点噪声，使数据集描述生长过程更加真实，最终将数据点分为训练集和测试集，如图 5-24 所示。

其中，蓝色的点为观测点；红色的曲线为理想中模型的函数曲线，因为在函数点中增加了一些噪声，因此蓝色的点并非与红色的模型曲线完全拟合；绿色的曲线则为关于蓝色的点的模拟曲线，可以看出大体上绿色的曲线和红色的模型曲线还是重合的，稍微的细微差距为真实情况下的差异。绿色拟合曲线的各个参数值，如图 5-25 所示。

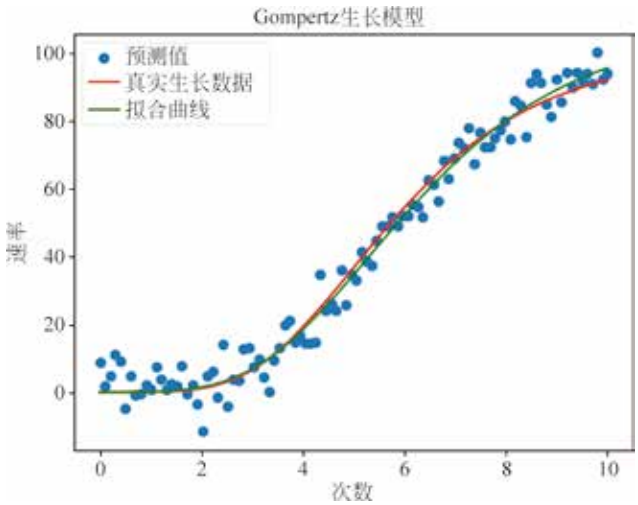


图 5-24　生长曲线示意图

```
拟合参数:
K = 108.43
r = 0.44
t = 5.29
```

图 5-25　拟合参数结果

非线性最小二乘拟合的代码如下：

```
// 第 5 章 / 非线性最小二乘拟合 .py
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# 定义 Gompertz 生长模型
def gompertz_growth(x, K, r, t):
    return K * np.exp(-np.exp(-r * (x - t)))

# 生成模拟数据
np.random.seed(0)
x_data = np.linspace(0, 10, 100)
y_true = 100 * np.exp(-np.exp(-0.5 * (x_data - 5)))  # 真实的生长曲线
y_data = y_true + np.random.normal(0, 5, size=len(x_data))
 # 加入噪声的观测数据

# 利用 curve_fit 进行拟合
popt, pcov = curve_fit(gompertz_growth, x_data, y_data, p0=[100, 0.1, 5])

# 提取拟合的参数
K_fit, r_fit, t_fit = popt

# 绘制拟合结果
plt.scatter(x_data, y_data, label='Observations')
plt.plot(x_data, y_true, color='red', label='True Growth Curve')
plt.plot(x_data, gompertz_growth(x_data, *popt), color='green', label='Fitted
Growth Curve')
plt.title('Gompertz Growth Curve Fitting')
```

```
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend()
plt.show()

print("拟合参数:")
print(f"K = {K_fit:.2f}")
print(f"r = {r_fit:.2f}")
print(f"t = {t_fit:.2f}")
```

5.4.2　高斯消元法

高斯消元法是求解线性方程组的一种算法，常用于行列式计算、求矩阵的逆，以及其他计算机和工程方面。

消元的过程通过一系列的代数运算，将方程组中的某个方程的未知数表示为其他方程中未知数的线性组合。这样做的目的是逐步消除方程组中的未知数，从而简化解方程的过程。

假设现在存在二元一次线性方程组：

$$\begin{cases} 3x-y=100 \\ x-y=100 \end{cases} \tag{5-6}$$

将方程组中的 1 式减去 2 式，消去方程组中的未知数 y，得到 $x=0$。

将 $x=0$ 再代回 2 式中，得到 $0-y=100$，解得 $y=-100$。

在消元过程中主要有两个基本的原理：①一个方程各项式同时乘以一个系数 k，解不变；②两方程相加或者相减，解不变。

高斯在消元的基础上进行改进，提出了高斯消元法，共分为 4 个步骤：构建增广矩阵、消元、回代、检查结果唯一性。

已知线性方程组如下：

$$\begin{cases} 2x+y-z=8 \\ -3x-y+2z=-11 \\ -2x+y+2z=-3 \end{cases} \tag{5-7}$$

构建增广矩阵：

$$\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right) \tag{5-8}$$

对其进行求解，最终解得如下方程组：

$$\begin{cases} 2x+y-z=8 \\ \dfrac{1}{2}y+\dfrac{1}{2}z=1 \\ -2z=1 \end{cases} \tag{5-9}$$

解得 $z=-1$，将其代入 2 式中解得 $y=3$，将两个未知数的值代入 1 式中解得 $x=2$。

高斯消元法实现过程的代码如下：

```
// 第 5 章 / 高斯消元法 .py
def gauss_elimination(A, b):
    n = len(A)

    #Elimination phase
    for i in range(n):
        #Partial pivoting
        max_row = i
        for k in range(i+1, n):
            if abs(A[k][i]) > abs(A[max_row][i]):
                max_row = k
        A[i], A[max_row] = A[max_row], A[i]
        b[i], b[max_row] = b[max_row], b[i]

        #Zeroing out below the pivot
        for j in range(i+1, n):
            factor = A[j][i] / A[i][i]
            for k in range(i, n):
                A[j][k] -= factor * A[i][k]
            b[j] -= factor * b[i]

    #Back substitution
    x = [0] * n
    for i in range(n - 1, -1, -1):
        x[i] = b[i] / A[i][i]
        for j in range(i):
            b[j] -= A[j][i] * x[i]

    return x

#Example usage
A = [[2, 1, -1],
     [-3, -1, 2],
     [-2, 1, 2]]

b = [8, -11, -3]

solution = gauss_elimination(A, b)
formatted_solution = [round(sol, 1) for sol in solution]
print("Solution:", formatted_solution)
```

5.4.3 牛顿法

牛顿法是一种基于切线逼近的迭代算法，用于快速地找到实值函数的零点，通过当前点的导数信息来更新下一步的近似解。

1. 概念解析

牛顿法是在实数和复数域上近似求解方程的方法，其利用函数$f(x)$的泰勒级数计算方程$f(y)=0$的根。该方法的基本思想是先从一个初始近似值开始，然后使用方程的导数信息来不断地改进这个近似值，直到得到满足精度要求的根的近似值。通常情况下，牛顿法具有快速收敛的特点，尤其是在初始值选取得当且方程光滑的情况下。

牛顿法的优点是速度相对较快，并且高度逼近最优值。其迭代公式如下：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)} \tag{5-10}$$

其中，x_n是第n次迭代的近似根，$f(x_n)$是方程，$f'(x_n)$是$f(x)$的导数。牛顿法的主要优点是其收敛速度通常很快，尤其是在靠近根的地方。然而，它也有一些局限性，例如可能会收敛到非根的点上，或者在某些情况下可能出现发散。此外，牛顿法对初始近似值的选取比较敏感。

2. 案例分析

现在存在一个方程$f(x)=x^2-4$，利用牛顿法来求解方程的根。

首先需要定义函数和它的导数，已知函数$f(x)=x^2-4$，那么它的导数$f'(x)=2x$。

之后选择一个初始近似值x_0，假设其等于 2，那么初始近似值的函数值$f(x_0)=0$。

注意： 初始近似值是操作者自己进行选取的，这个初始值可以是任意合理的数值，通常是通过对问题的先验知识或者对方程特性的了解来选择的。初始近似值的选择对于牛顿法的收敛性和求解效率都有重要影响。如果初始值距离方程根比较远，则可能会导致牛顿法迭代过程中跳跃性地振荡或者收敛到错误的根，而如果初始值选取得比较接近方程根，则通常会加速收敛过程并提高求解的准确性。

之后使用牛顿迭代法进行迭代计算：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)} \tag{5-11}$$

将$x_0=2$和$f(x)=x^2-4$及$f'(x)=2x$代入，可以得到：

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)}=2-\frac{2^2-4}{2*2}=2-0=2 \tag{5-12}$$

如今得到$x_1=2$，继续迭代运算，直到满足某个停止条件，最终得到方程的根等于 2。

牛顿法求解近似值的代码如下：

```
// 第 5 章 / 牛顿法 .py
def f(x):
    return x**2 - 4

def f_prime(x):
```

```
        return 2*x

    def newton_method(initial_guess, tol=1e-6, max_iter=100):
        x = initial_guess
        iterations = 0

        while abs(f(x)) > tol and iterations < max_iter:
            x = x - f(x) / f_prime(x)
            iterations += 1

        if iterations == max_iter:
            print("Maximum iterations reached without convergence.")
        else:
            print(f"Root found: {x} after {iterations} iterations.")

        return x

    # 初始近似值
    initial_guess = 2
    # 调用牛顿法函数
    root = newton_method(initial_guess)
```

5.5 机器学习算法

机器学习是人工智能的一个重要分支，通过开发算法使计算机能够从数据中自动学习。主要算法包括监督学习和无监督学习。前者通过标注数据进行预测或分类，常见的有线性回归和逻辑回归；后者则通过发现数据的内在结构进行聚类，典型算法如 k 均值算法。决策树既可用于监督学习，也可用于无监督学习。由于机器学习与数学联系紧密，所以许多算法背后依赖概率论、统计学和线性代数知识。

5.5.1 逻辑回归

逻辑回归是一种用于二分类问题的机器学习算法，尽管名称中有“回归”，但是它实际上用于分类任务。通过对输入特征的线性组合，逻辑回归计算出样本属于某类别的概率。由于其简洁性、易实现性和可解释性，它被广泛地应用于垃圾邮件分类、疾病预测等场景。Python 中常用 scikit-learn 库及 NumPy 库等来实现逻辑回归，提供灵活的模型训练和正则化选项，帮助开发者有效地解决分类问题。

1. 逻辑回归概念解析

逻辑回归是一种经典的统计方法，应用于解决二分类问题。它的核心任务是估计某个事件发生的概率，并利用这一概率来对数据进行分类。逻辑回归模型的输出是一个介于 0 到 1 之间的概率值，这个值表示某个样本属于特定类别的可能性。在实际应用中，通常通过设置

一个阈值，将输出的概率与阈值进行比较，以决定样本属于哪个类别。常见的做法是将概率大于阈值的样本归为正类（类别 1），将小于或等于阈值的样本归为负类（类别 0）。

二分类问题也是机器学习中的常见任务类型。在二分类问题中，每个样本都必须被明确地归入两个互斥的类别。这类任务在日常生活和实际应用中非常普遍。例如，在垃圾邮件检测中，邮件被分类为“垃圾邮件”或“非垃圾邮件”；在图像分类中，可能需要判断某张图片中的物体是“猫”还是“狗”；在文本情感分析中，系统会将用户的评论或反馈划分为“积极”或“消极”等情感类别。这些场景都要求模型能够精确地判断每个样本所属的类别。

在完成逻辑回归模型的训练后，通常需要对其性能进行评估。这一过程包括对模型的准确性、精度、召回率等指标的分析，以确定模型在预测新数据时的表现是否可靠。如果模型的性能未达到预期标准，则开发者可以通过调整模型的参数、选择不同的特征或改进数据预处理步骤等手段进行修正和优化。在实际项目中，优化后的逻辑回归模型常被用于各种分类预测任务，以便帮助解决一系列实际问题。

为了实现逻辑回归，引入了 Sigmoid 函数，它能够将线性回归的输出值限制在［0，1］的区间内。具体而言，当线性回归的输出值趋近负无穷时，Sigmoid 函数的输出接近 0；当输出值趋近正无穷时，函数值接近 1，如图 5-26 所示。图中展示了 Sigmoid 函数的曲线，它形似 S 形，正是这种曲线使逻辑回归能够在二分类问题中有效工作。

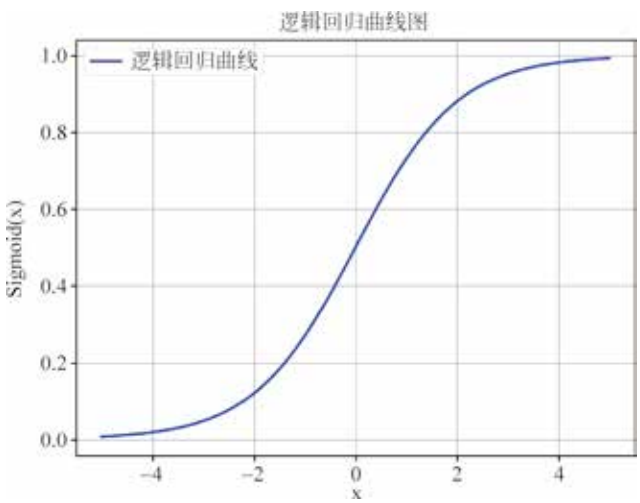


图 5-26　逻辑回归曲线图

构建逻辑回归曲线函数的代码如下：

```
// 第 5 章 / 逻辑回归曲线图 .py
import numpy as np
import matplotlib.pyplot as plt

# 定义逻辑函数（S 形函数）
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 生成一些示例数据
x = np.linspace(-5, 5, 100)
y = sigmoid(x)

# 绘制曲线
plt.plot(x, y, label='Sigmoid Curve', color='blue')
plt.xlabel('x')
plt.ylabel('Sigmoid(x)')
plt.title('Logistic Regression Sigmoid Curve')
plt.legend()
plt.grid(True)
plt.show()
```

2. 逻辑回归算法实现

逻辑回归的过程通常分为 4 个基本步骤：初始化、使用 Sigmoid 函数、模型拟合和进行预测。

首先，初始化步骤通常在代码中通过 __init__() 方法完成。初始化包括设置学习率、迭代次数，并将模型的权重和偏置参数初始化为 0。这是逻辑回归模型的起点，其中学习率决定了每次参数更新的步长。学习率的选择对模型的训练过程至关重要。如果学习率过小，则模型收敛速度会很慢，意味着需要更多的迭代次数才能达到收敛；反之，如果学习率过大，则可能会导致模型跳过最优值，无法收敛。梯度下降优化算法通过多次迭代，逐步更新模型参数，以找到最优解。学习率的大小直接影响了梯度下降的步幅大小，如图 5-27 所示，当步幅过小时，模型的收敛过程会显得缓慢，而当步幅过大时，模型可能会直接跳过最优解，导致不稳定或无法收敛。

迭代次数是指在参数更新过程中，模型对训练数据执行更新的次数。它与学习率共同影响着模型的优化过程，但二者的作用有所不同。学习率决定了每次参数更新时的步幅大小，即每次迭代中模型参数的调整幅度，而迭代次数则表示参数更新的总次数。在梯度下降优化算法中，每次迭代都会先基于当前的参数计算损失函数的梯度，然后利用该梯度更新参数，以减小损失函数的值，从而使模型逐渐逼近最优解。

理论上来讲，迭代次数越多，模型的训练越充分，它有更多机会调整参数，使其更好地拟合训练数据，然而，现实中的条件是多变的，并且资源利用却是非常有限的。过多的迭代次数可能会导致模型过拟合，即模型对训练数据的拟合过度，表现出较高的训练精度，但在新数据上的泛化能力下降。另一方面，过多的迭代次数也会显著地增加模型的训练时间，尤其是在处理大规模数据集时，可能会带来不必要的计算开销。

因此，在实际应用中，选择合适的迭代次数至关重要。设置较少的迭代次数可能会导致模型未能充分训练，距离最优解较远，而过多的迭代次数则可能陷入过拟合或增加训练成本，如图 5-28 所示。

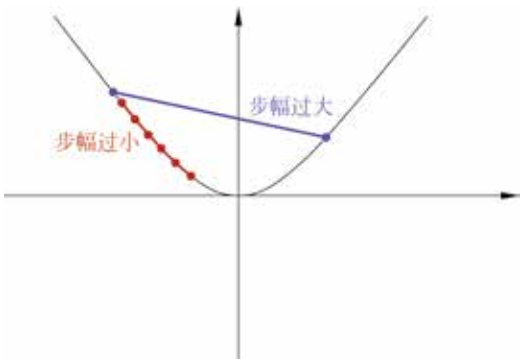


图 5-27　学习率设置图片

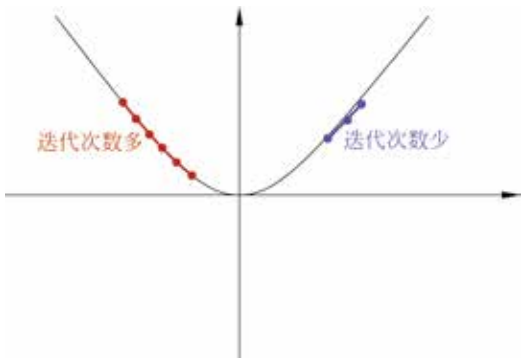


图 5-28　迭代次数设置图

权重是逻辑回归模型中的关键参数，它代表输入特征的系数。模型通过对各输入特征进行加权，计算出线性组合值，从而预测输出。每个输入特征的权重反映了该特征在最终预测中的重要性，权重的大小和符号会直接影响模型的预测结果。如果某个特征的权重较大，则说明它对输出结果的影响更显著；反之，权重较小的特征对模型的预测影响较弱。此外，权重的正负号也决定了该特征对输出是正向还是负向的影响。

偏置是逻辑回归模型中的常数项，它的作用是调整预测值，使模型的预测更加贴近实际情况。偏置可以看作模型的基础输出值，即在所有输入特征都为 0 的情况下，模型的预测值。通过引入偏置项，模型能够对预测值与实际值之间的偏差进行校正，以更好地捕捉数据中的规律，尤其在特征值较小或者线性可分性较差的情况下，偏置项的引入显得尤为重要。

在逻辑回归中，输入数据点往往属于两种不同的类别，例如红色点和蓝色点代表两类不同的数据样本。在分类任务中，模型的目标是通过找到一个最佳决策边界，将这两类数据准确地区分开。为了实现这一目标，逻辑回归使用了 Sigmoid 函数，它将线性组合后的输入值映射为一个概率值，并根据该概率来预测每个样本属于某个类别的可能性。具体而言，Sigmoid 函数将模型的输出压缩到［0，1］区间，通过对这些概率值设定阈值，模型即可将输入数据点划分为不同的类别，如图 5-29 所示。通常，若 Sigmoid 函数输出的概率值大于 0.5，则该数据点被分类为正类；反之，则被分类为负类。

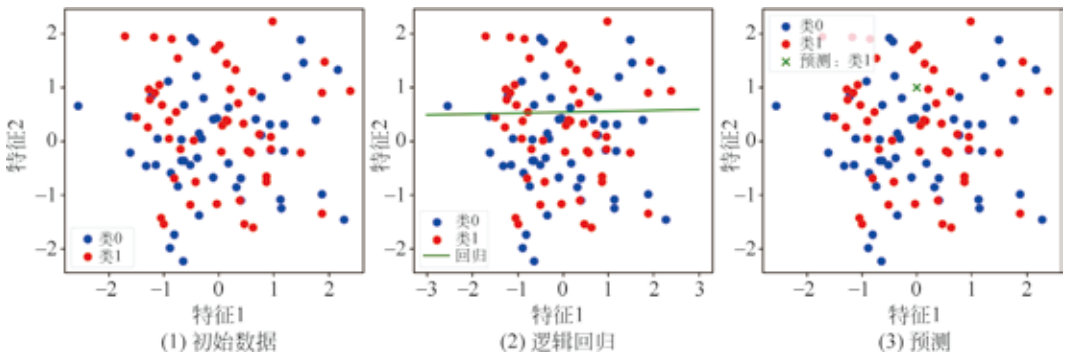


图 5-29　逻辑回归函数图

逻辑回归算法实现过程的代码如下：

```
// 第 5 章 / 逻辑回归曲线图 .py
import numpy as np
import matplotlib.pyplot as plt

#Sigmoid 函数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 生成一些示例数据
np.random.seed(0)
x = np.random.randn(100, 2)
y = np.random.randint(0, 2, size=100)

# 训练逻辑回归模型
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(x, y)

# 提取模型参数
theta_0 = model.intercept_[0]
theta_1, theta_2 = model.coef_[0]

# 将数据分为两个类别
class_0 = x[y == 0]
class_1 = x[y == 1]

# 绘制原始数据的散点图
plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.scatter(class_0[:, 0], class_0[:, 1], color='blue', label='Class 0')
plt.scatter(class_1[:, 0], class_1[:, 1], color='red', label='Class 1')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Original Data')
plt.legend()

# 绘制拟合到 Sigmoid 函数的图
plt.subplot(1, 3, 2)
plt.scatter(class_0[:, 0], class_0[:, 1], color='blue', label='Class 0')
plt.scatter(class_1[:, 0], class_1[:, 1], color='red', label='Class 1')

# 计算拟合直线的斜率和截距
x_values = np.linspace(-3, 3, 100)
y_values = -(theta_1 * x_values + theta_0) / theta_2

# 使用 Sigmoid 函数将拟合直线限制在 0 和 1 之间
```

```
y_values = sigmoid(y_values)

plt.plot(x_values, y_values, color='green', label='Decision Boundary')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Logistic Regression Fit')
plt.legend()

# 绘制预测图
plt.subplot(1, 3, 3)
plt.scatter(class_0[:, 0], class_0[:, 1], color='blue', label='Class 0')
plt.scatter(class_1[:, 0], class_1[:, 1], color='red', label='Class 1')

# 新的测试数据点
x_new = np.array([[0, 1]])
y_new = model.predict(x_new)
plt.scatter(x_new[:, 0], x_new[:, 1], color='green', marker='x', label='New
Prediction: Class {}'.format(y_new[0]))

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Prediction')
plt.legend()

plt.tight_layout()
plt.show()
```

3. 逻辑回归实际应用介绍

逻辑回归在实际应用中非常广泛，其中一个常见的应用场景便是垃圾邮件检测。以搭建一个垃圾邮箱检测系统为例，一共需要分成比 7 部分。

（1）系统需要一个经过标记的数据集，该数据集中的邮件样本被分为两类：垃圾邮件和非垃圾邮件。获取高质量的标记数据集是构建模型的基础，因为只有这样，模型才能学习到有效的特征，进而准确地进行分类。

（2）在获取数据集后，便要进行数据预处理操作。数据预处理的目的是清理和标准化数据，以便为后续的特征提取和模型训练做好准备。具体的预处理操作可能包括删除特殊字符、将所有文本转换为小写、去除停用词及进行拼写校正等。

（3）进行特征提取。特征提取是从邮件内容中识别和提取有助于分类的关键信息。在垃圾邮件检测中，常用的特征提取方法包括词频（Term Frequency）、TF-IDF（Term Frequency-Inverse Document Frequency）等。这些特征可以有效地反映邮件中某些词语的重要性和出现的频率。特征提取可以通过对邮件进行分词操作、统计每个词的出现次数、或者计算某个词在整个数据集中出现的频率等实现。

在特征提取的过程中，如果邮件内容中包含明显的广告信息（需要开发者自行加入判断，一般来讲需要付费等内容），则可以将其判断为垃圾邮件，而如果邮件内容涉及会议

安排或其他重要事项，则可以将其归为非垃圾邮件。系统会对每封邮件进行遍历，提取特征并进行分类，最终确定每封邮件的类别。

（4）完成特征提取后，数据集需要进行分割，将其分为训练集和测试集。训练集用于模型的训练，测试集则用于评估模型的性能。通常，数据集的划分比例为 70%~80% 用于训练，剩余的用于测试。

（5）训练模型。训练过程将通过逻辑回归模型对训练集进行拟合，以求得最佳的权重和偏置参数。

（6）在模型训练完成后，使用测试集对模型进行评估，以验证模型的效果。评估指标包括准确率（Accuracy）、精确率（Precision）、召回率（Recall）等。这些指标可以全面地反映模型的分类性能，帮助判断模型在实际应用中的有效性。

（7）在评估和预测工作完成后，可能会发现模型在某些特征或分类任务上表现不佳。此时，需要对模型进行优化。优化功能涉及调整特征选择、重新定义特征提取的方法，或者修改模型参数等。

实现垃圾邮件检测功能的代码如下：

```
// 第 5 章 / 垃圾邮件检测功能 .py
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# 示例数据集，包含邮件内容和标签（0 为非垃圾邮件，1 为垃圾邮件）
emails = [
    ('Get free money', 1),
    ('Claim your prize', 1),
    ('Send money now', 1),
    ('Get free gift', 1),
    ('Congratulations, you have won', 1),
    ('Meeting agenda', 0),
    ('Reminder: project deadline', 0),
    ('Launch menu for today', 0),
    ('Meeting minutes', 0),
    ('Thank you for your email', 0)
]

# 将邮件内容和标签分离
x = [email[0] for email in emails]
y = [email[1] for email in emails]

# 将文本数据转换为特征向量
vectorizer = CountVectorizer()
x_vectorized = vectorizer.fit_transform(x)
```

```
# 将数据集划分为训练集和测试集
x_train, x_test, y_train, y_test = train_test_split(x_vectorized, y, test_
size=0.2, random_state=42)

# 训练逻辑回归模型
model = LogisticRegression()
model.fit(x_train, y_train)

# 在测试集上评估模型性能
y_pred = model.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_report(y_test, y_pred))

# 使用训练好的模型进行预测
new_emails = [
    'Get your free gift now',
    'Important project update',
    'You have won a prize'
]
new_emails_vectorized = vectorizer.transform(new_emails)
predictions = model.predict(new_emails_vectorized)
for email, prediction in zip(new_emails, predictions):
    print("Email:", email)
print("Prediction (1 for spam, 0 for non-spam):", prediction)
```

5.5.2　梯度下降法

梯度下降是一种优化算法，通常应用于机器学习和深度学习中。梯度下降法，使迭代参数不断更新，每次参数更新都是在提升模型的准确率。该模型持续调整其参数，直至该函数接近或等于 0，以使产生的误差尽可能最小。机器学习的模型通过优化后，就可以投入到实际应用中。

在梯度下降法中，整个过程大致分为三类：初始点的选择、迭代过程和收敛条件。首先是初始点的选择，初始点的选择一般来讲是随机进行挑选的，方便后续训练模型的准确性，如果是精挑细选的一个点，则反倒可能会遗漏掉一些情况；其次便是迭代过程，梯度下降是一个迭代参数的过程，通过多次迭代来使损失函数最小，在每次迭代中，使用当前参数值计算梯度，并根据学习率更新参数值；最后便是收敛条件，收敛条件一般称为终止条件，例如达到最大迭代次数或者损失函数变化很小时。

在迭代过程中，主要涉及 3 个比较主要的名词：梯度、学习率和更新规则。在梯度下降中，主要进行计算的便是其梯度，梯度是指在当前点上函数增长最快的方向。在二维空间中，梯度是一个向量，指向函数值增长最快的方向。学习率是梯度下降中的一个超参数，用于控制每次迭代中参数更新的步长。它决定了在梯度方向上移动的大小：如果学习率太小，则收敛速度会很慢；如果学习率太大，则可能会导致错过最小值。一般来讲，更

新规则参数值等于当前值减去学习率乘以梯度值，这个更新规则使参数朝着减小损失函数的方向移动。

假设现在存在一个简单函数 $f(x)=x^2$ 的平面图，如图 5-30 所示。目标是找到这个函数图像上的最小值，当然我们都知道最小值在 $x=0$ 的地方，假设现在还不知道，该如何使用梯度下降的方法去找到最小值呢?

首先可以计算当前函数的导数，当前函数的导数即当前的梯度：

$$\begin{cases} f(x)=x^2 \\ f'(x)=2x \end{cases} \tag{5-13}$$

之后选择初始值，假设当前函数的初始值在 $x=2$ 的位置，假设初始值的形参为 x_0，接下来反方向调整 x 的值，即朝着导数的负方向移动一小步。可以使用以下更新规则对函数梯度进行更新：

$$x_1=x_0-\alpha f'(x) \tag{5-14}$$

其中，α 是学习率控制着每次迭代的步长。假设规定 $\alpha=0.1$，现在计算：

$$x_1=2-0.1*(2*2)=2-0.4=1.6 \tag{5-15}$$

现在得到一个新的 x 的值，重复以上过程，直到找到一个满足需求的值，或者达到预期规定的迭代次数，如图 5-31 所示。

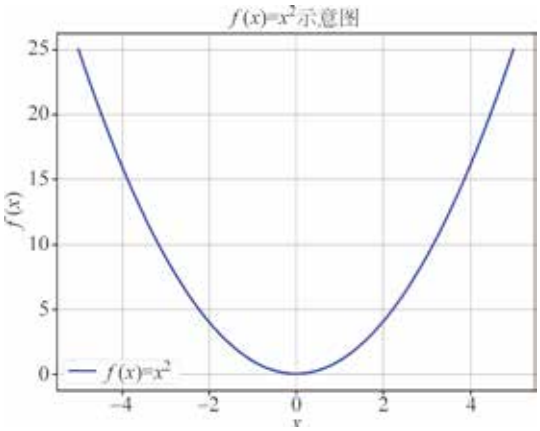


图 5-30　函数示意图

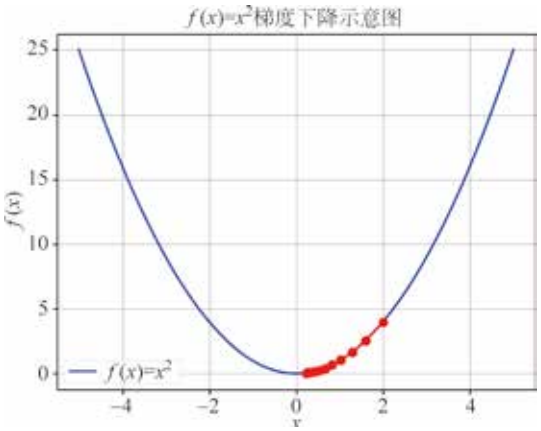


图 5-31　梯度下降迭代过程图

梯度下降方法实现过程的代码如下：

```
// 第 5 章 / 梯度下降迭代过程 .py
import numpy as np
import matplotlib.pyplot as plt

# 定义函数
def f(x):
    return x**2
```

```
# 定义函数的导数
def df(x):
    return 2 * x

# 梯度下降算法
def gradient_descent(learning_rate, num_iterations, initial_x):
    x = initial_x
    history = [x]
    for i in range(num_iterations):
        gradient = df(x)
        x = x - learning_rate * gradient
        history.append(x)
    return history

# 设置学习率和迭代次数
learning_rate = 0.1
num_iterations = 10
initial_x = 2  # 初始点的选择可能会影响最终的收敛结果

# 执行梯度下降算法
history = gradient_descent(learning_rate, num_iterations, initial_x)

# 绘制函数图像
x_values = np.linspace(-5, 5, 100)
y_values = f(x_values)
plt.plot(x_values, y_values, label='f(x) = x^2', color='blue')

# 绘制梯度下降路径
for i in range(len(history)-1):
    plt.plot([history[i], history[i+1]], [f(history[i]), f(history[i+1])],
marker='o', color='red')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Gradient Descent on f(x) = x^2')
plt.grid(True)
plt.legend()
plt.show()
```

5.5.3　决策树

决策树是一种广泛应用于分类和回归分析的机器学习算法，其独特之处在于它以树的结构形式展现决策过程。这种结构不仅直观易懂，而且能够有效地对复杂的数据进行处理。决策树既可以呈现为二叉树形式，即每个节点最多有两个子节点，也可以是非二叉树形式，允许每个节点有多个子节点，具体形式的选择通常取决于所处理问题的复杂性和特征的数量。

在决策树中，树状数据结构包含几个重要的概念。首先是根节点，它代表整个样本数

据集的起点。在树的构建过程中，根节点是通过对输入特征进行分裂而形成的，分裂的依据通常是特征值对输出结果的影响程度。

其次是叶节点，叶节点代表决策树的最终输出结果。在分类任务中，叶节点通常对应于特定的类别标签，而在回归任务中，叶节点则表示对目标值的预测。每当输入数据通过决策树的各个分支到达叶节点时，模型便根据所经过的路径做出决策。

分支是决策树中连接不同节点的重要部分，代表节点之间的连接关系。在每个节点上，决策树通过特定的条件将样本数据划分为不同的子集。这样的划分依据通常是特征的值，例如某个特征是否大于某个阈值。分支不仅可以帮助构建树的结构，也为后续的决策提供依据。

决策树的构建过程一般包括数据的分裂、节点的选择和树的修剪。数据的分裂通常使用一些评估指标（例如信息增益、基尼指数或均方误差等）来衡量每次分裂的效果。节点的选择则决定了哪些特征在当前节点上进行分裂。为了避免模型的过拟合，决策树构建完成后，通常需要对树进行修剪，通过删除某些叶节点或分支，简化模型结构，如图 5-32 所示。

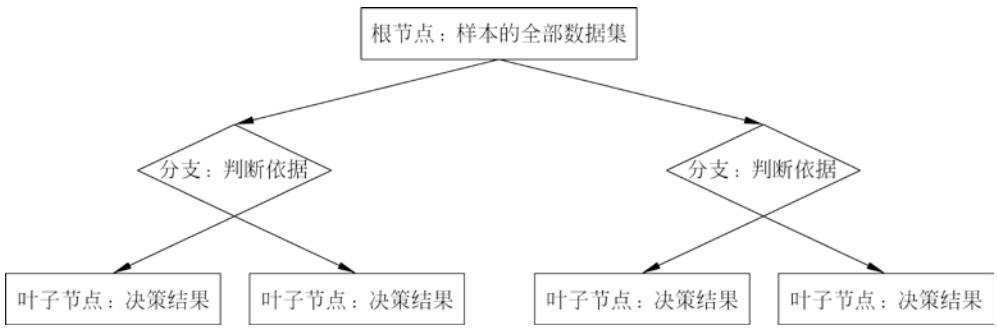


图 5-32　决策树示意图

决策树是一种典型的有监督学习算法，其基本原理是通过一系列的 if-else-then 判断规则来对数据进行分类或回归。构建决策树的过程主要包括选择最佳属性进行数据分裂，并依据特定的分裂标准（例如信息增益、基尼系数或均方误差等）来优化树的结构。每次分裂后，数据集被划分为不同的子集，并通过这种递归方式，最终形成树状结构。在树的每个节点上，根据特征的值进行判断，决定是继续向下分裂还是输出最终的分类结果。若判断结果为“是”，则输出相应的结果；若为“否”，则继续进入下一个判断阶段，直至达到叶节点。

考虑一个具体的例子：假设在一个周末，全家决定是否出去野餐。在这个过程中，可以使用决策树的结构来进行判断。首先，需要考虑的首要因素是天气情况。如果为阴雨天而非晴天，则不适合外出野餐，这构成了树的第 1 层判断。如果判断结果为晴天，则接下来需评估温度的影响。若温度超过 28℃，则判断结果为“不适合”，因为高温可能使外出活动变得不愉快。

如果天气晴朗且温度在可接受范围内，则可以进一步进入下一个判断层次。此时，紫

外线强度成为新的判断依据。紫外线强度的高低对人体皮肤健康有直接影响，因此也是影响外出郊游的重要因素。如果紫外线强度大于或等于 3 级，则判断结果为“不适合郊游”，因为强紫外线可能造成皮肤损伤；如果紫外线强度低于 3 级，则最终判断结果为“适合进行郊游”，此时全家可以安心外出享受野餐。

决策树如何通过一系列的条件判断来得出最终决策。它通过层层筛选，最终确定了满足自己的条件，进而进行决策，如图 5-33 所示。

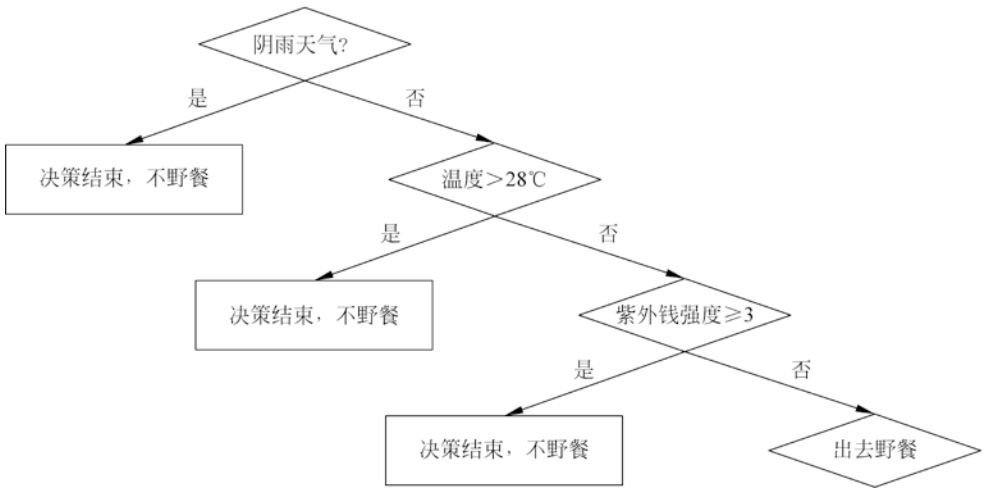


图 5-33　野餐决策示意图

在构建决策树的过程中，首先需要对判断条件进行数值化，以便模型能够理解和处理这些数据。在这种数值化的表示中，通常采用二进制编码来简化分类问题。例如，对于一个肯定的回答，可以用数值 0 来表示，而对于一个否定的回答，则使用数值 1 来表示。

具体而言，针对特定的判断条件，例如天气、温度和紫外线强度都可以被转化成数值的形式，在该示例中，可以用 0 表示阴天，用 1 表示晴天；对于温度，可以设定一个阈值，当温度超过 28℃时，表示为 1，而不超过 28℃则表示为 0；对于紫外线强度，若强度大于或等于 3，则用 1 表示，反之则用 0 表示。

在完成数值化的准备工作后，接下来可以利用 sklearn 库中的 DecisionTreeClassifier 类来进行决策树的构建和训练。首先，创建一个 DecisionTreeClassifier 对象，用户可以自定义一些参数，树的最大深度、最小样本分裂数等。之后，使用 fit( ) 函数将处理后的数据集传入该模型进行拟合训练。

训练完成后，可以使用 predict( ) 函数对新样本进行预测。通过输入新的特征数据，决策树模型将根据已学习到的规则进行分类并输出预测结果。如果预测结果符合预期，则说明模型的构建与训练过程是有效的。在实际应用中，可以通过不断调整模型参数和训练数据来进一步地提高模型的准确性和泛化能力。

决策树构建过程的代码如下：

```
// 第 5 章 / 决策树 .py
from sklearn.tree import DecisionTreeClassifier

# 创建样本数据
x = [[0, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
#0 表示阴天、28℃以上、紫外线强度大于或等于 3
y = [0, 0, 0, 1]  #0 表示不适合野餐，1 表示适合野餐

# 创建决策树模型
clf = DecisionTreeClassifier(random_state=42)

# 拟合模型
clf.fit(x, y)

# 输入新的天气、温度和紫外线条件
new_data = [[1, 1, 1]]  # 例如天气是晴天、温度不超过 28℃、紫外线强度大于或等于 3

# 进行预测
prediction = clf.predict(new_data)

# 根据预测结果输出判断
if prediction[0] == 1:
    print(" 适合野餐 !")
else:
    print(" 不适合野餐 !")
```

决策树主要包括 3 种实现方法：ID3 算法、C4.5 算法及 CART 算法。这些算法各有其特点和适用场景，但在 Python 编程语言中，使用 sklearn 库进行决策树的实现时，通常采用的是 CART 算法。CART 算法的设计旨在处理分类和回归问题，并具有良好的可解释性和灵活性。

CART 算法的基本思想是通过递归将数据集划分为越来越小的子集。在每次划分过程中，算法会选择最佳的特征及其分割点，以确保划分后的子集中的样本尽可能地属于同一类别（在分类问题中），或使样本的数值尽可能相似（在回归问题中）。这个过程是通过评估某种准则来实现的。这些准则帮助算法决定如何进行划分，以最大化划分后的纯度或最小化划分后的不纯度。

当然仅仅进行模拟回归和预测还是远远不够的，真正想要将决策树运用到实际当中，还需要调整一些参数来修正决策树的复杂度、剪枝策略等，以便用户能够根据实际情况进行调节和优化。

5.5.4 *K* 近邻算法

K 近邻（*K*-Nearest Neighbors，KNN）算法是一种简单而有效的监督学习算法，通过计算样本之间的距离，将待分类样本归类为其 *K* 个最近邻居中最常见的类别，被广泛地应用于分类和回归问题。

1. *K* 近邻算法与 *K* 均值聚类算法

K 近邻算法和 *K* 均值（*K*-Means）聚类算法是机器学习中的两种算法。名称相近，但是它们在原理及应用上有明显的差异。选择将它们放在一块进行介绍，可以帮助读者厘清这两个概念之间的关系。

1）*K* 近邻算法

K 近邻算法是有监督学习学习中的分类算法，可以用于分类及回归等。*K* 近邻算法，首先需要对邻居的数量进行限定，即对 *K* 值进行限定。对于每个测试样本，计算它与训练集中每个样本之间的距离。常用的距离度量包括欧氏距离、曼哈顿距离、闵可夫斯基距离等。根据计算出的距离，从训练集中选择距离测试样本最近的 *K* 个样本。

2）*K* 均值聚类方法

K 均值聚类算法是一种无监督的学习方法。其因为是无监督的学习方法，因此事先并不需要对数据集内的数据进行标注，而是由机器自行进行，而由于是聚类算法，因此最终整个数据会被分割为 *K* 个团簇（*K* 无特殊含义，只是一个预设的团簇数量）。使用 *K* 均值聚类方法，首先需要在数据进行聚类伊始，便要设定好 *K* 值，即要将目前的数据集分割为多少个团簇。后续操作，便由机器自行进行，按照机器对于数据的理解，将数据按照规定（设定的 *K* 值）进行分类，最终会得出结果：出现 *K* 个中心点，以及每个数据点属于哪个 *K* 中心点。

3）案例分析

现代智能手机的功能日益强大，不仅能拍摄高质量的图像，还能在拍摄过程中记录大量附加信息。这些附加信息通常包括拍摄的时间、地点及其他与图像相关的元数据。当用户拍摄多张图片时，这些信息会被整合成一个数据集，其中每张图片都携带着特定的地理信息。

例如，苹果手机在拍摄图像时会自动地将地理位置信息嵌入图像文件中。这些信息可以通过分析，形成一个可视化的图片地图，如图 5-34 所示。此时，需要探讨的问题是，生成这一图片地图所使用的算法是 *K* 近邻算法还是 *K* 均值聚类算法。要回答这一问题，关键在于分析图片地图中数据点的数量是否固定。

如果图片地图中数据点的数量是固定的，则可以推断出使用的是 *K* 近邻算法。在 *K* 近邻算法中会对待分类的样本与已有样本进行比较，并选择 *K* 个最近邻居进行汇总，从而实现分类。此时，数量的固定性确保了每次分类都基于相同数量的样本，使分类结果更为稳定。

图 5-34　图片地图

另一方面，如果数据点的数量不是固定的，则更可能采用 *K* 均值聚类算法。*K* 均值聚类是一种无监督学习方法，它通过选择 *K* 个中心点来将附近的数据点分组。该算法的核心思想是最小化组内的平方误差，从而将数据划分为不同的聚类。在此情况下，数据点的数量并不影响算法的执行，算法的目标是根据数据的分布情况动态地调整聚类中心。

2. 实现步骤

K 近邻算法的实现过程首先需要准备一个包含训练数据的标准数据集。数据集中的每个数据点都应该包含一个标签，以便于区分其所属的类别。这些标签通常代表了数据点的类别信息，是模型在训练过程中不可或缺的部分。

在数据集准备好之后，需要设定 *K* 值。*K* 值是算法的一个重要参数，通常由用户在实验开始之前指定。*K* 值的定义是相邻的数据点数量，也就是说，*K* 值决定了将多少个数据点视为一个簇。如果在图 5-34 中将 *K* 设定为 3，则意味着在分类过程中，相邻的 3 张图片将被视为同一类别。在这种情况下，生成的类别数量会相对较多，并且每个类别的规模较小。相反，如果将 *K* 值设定为 30，则意味着将 30 张图片归为同一类别，那么所形成的类别数量将相对较少，并且每个类别的规模较大，因此 *K* 值的选择对分类的精细度和范围具有直接影响。

注意： 一般来讲，较小的 *K* 值会使模型对噪声更敏感，可能会导致过拟合，而较大的 *K* 值会使模型更稳定，但可能会导致模型偏差增加，因此选择合适的 *K* 值对于 K 近邻算法的性能至关重要。

在确定了数据集和 *K* 值之后，*K* 近邻算法的数据模型构建基本上已完成。当出现一个未知的数据点时，算法需要计算该点与训练集中每个数据点之间的距离。这一过程通常使用多种距离计算方法，例如欧氏距离、曼哈顿距离和闵可夫斯基距离等。每种距离计算方法在不同的应用场景中具有各自的优势，因此根据具体情况选择合适的距离度量是非常重要的。

通过计算距离，*K* 近邻算法能够识别出与未知数据点最近的 *K* 个邻居。在分类问题中，算法会对这 *K* 个最近邻居的标签进行投票，将票数最多的标签作为未知数据点的预测类别。这种投票机制确保了分类结果的稳健性和准确性。在回归问题中，*K* 个最近邻居的标签值会被平均计算，得出的平均值则被作为未知数据点的预测值。

根据上述操作，最终输出预测结果，代码如下：

```
// 第 5 章 /KNN 算法示例 .py
import numpy as np
from collections import Counter

class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, x_train, y_train):
```

```
            self.x_train = x_train
            self.y_train = y_train

        def predict(self, x_test):
            predictions = []
            for x in x_test:
                distances = [np.linalg.norm(x - x_train) for x_train in self.x_
    train]
                k_indices = np.argsort(distances)[:self.k]
                k_nearest_labels = [self.y_train[i] for i in k_indices]
                most_common = Counter(k_nearest_labels).most_common(1)
                predictions.append(most_common[0][0])
            return predictions

    # 示例用法
    x_train = np.array([[1, 2], [3, 4], [5, 6]])
    y_train = np.array([0, 1, 0])
    x_test = np.array([[4, 5], [2, 3]])

    knn = KNN(k=2)
    knn.fit(x_train, y_train)
    predictions = knn.predict(x_test)
    print(predictions)  # 输出预测结果
```

3. 案例分析

淘宝首页的商品推广功能依赖于先进的推荐算法来实现个性化的用户体验，如图 5-35 所示。这些推荐算法通过分析用户的历史行为和偏好，生成针对性的商品推荐列表。具体而言，淘宝会综合考虑多个因素，包括用户的搜索记录、浏览历史及对产品的停留时间等。

首先，用户在淘宝平台上的搜索记录提供了大量有关用户偏好的信息。当用户在搜索框中输入关键词时，系统会记录下这些关键词，以了解用户的购物意图和兴趣。这些数据不仅反映了用户当前的需求，还可以揭示出潜在的消费趋势和习惯。

其次，用户在浏览商品时所停留的时间也是一个重要的指标。停留时间长的商品通常意味着用户对该商品的感兴趣。通过对停留时间的分析，推荐算法可以更好地判断哪些商品可能会吸引用户，从而提升推荐的准确性。此数据与搜索记录结合使用，可以为算法提供更全面的用户行为画像。

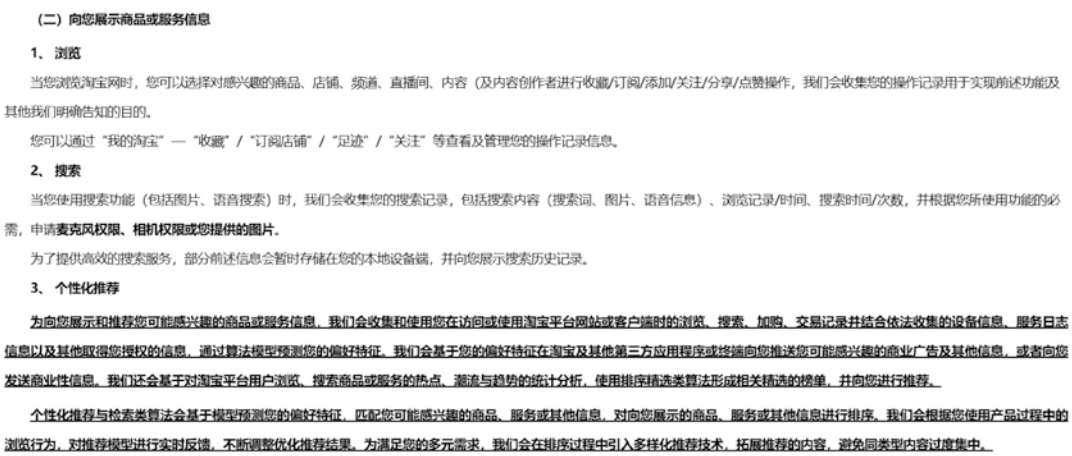
此外，淘宝还会考虑其他因素，例如用户的购买历史、评价反馈及相似用户的行为模式。这些信息不仅能够

图 5-35　淘宝首页推荐图

帮助算法优化推荐效果，还能够使推荐系统具备更强的自适应能力，能够根据用户不断变化的需求和偏好进行实时调整。

在实施 K 近邻算法以实现淘宝平台的推荐规则时，数据的收集是首要步骤。首先，推荐系统的有效性直接依赖于数据的质量和数量。没有足够的用户数据，推荐算法便无法进行有效训练和优化，因此数据的获取过程至关重要，必须遵循相关法律法规，并确保用户的隐私得到保护。

为了合法收集用户数据，淘宝平台需要用户的明确授权。在用户注册或使用平台服务之前，淘宝通常会提供一份隐私权政策，其中详细说明了数据收集的范围、目的和使用方式。用户在阅读并同意隐私权政策后，便可以使用平台的各项功能。这一过程不仅为用户提供了知情权，也为平台的推荐系统奠定了数据基础，如图 5-36 所示。



（二）向您展示商品或服务信息

1、浏览

当您浏览淘宝网时，您可以选择对感兴趣的商品、店铺、频道、直播间、内容（及内容创作者进行收藏/订阅/添加/关注/分享/点赞操作，我们会收集您的操作记录用于实现前述功能及其他我们明确告知的目的。

您可以通过“我的淘宝”—“收藏”/“订阅店铺”/“足迹”/“关注”等查看及管理您的操作记录信息。

2、搜索

当您使用搜索功能（包括图片、语音搜索）时，我们会收集您的搜索记录，包括搜索内容（搜索词、图片、语音信息）、浏览记录/时间、搜索时间/次数，并根据您所使用功能的必需，申请**麦克风权限、相机权限或您提供的图片。**

为了提供高效的搜索服务，部分前述信息会暂时存储在您的本地设备端，并向您展示搜索历史记录。

3、个性化推荐

为向您展示和推荐您可能感兴趣的商品或服务信息，我们会收集和使用您在访问或使用淘宝平台网站或客户端时的浏览、搜索、加购、交易记录并结合依法收集的设备信息、服务日志信息以及其他取得您授权的信息，通过算法模型预测您的偏好特征。我们会基于您的偏好特征在淘宝及其他第三方应用程序或终端向您推送您可能感兴趣的商业广告及其他信息，或者向您发送商业性信息。我们还会基于对淘宝平台用户浏览、搜索商品或服务的热点、潮流与趋势的统计分析，使用排序精选类算法形成相关精选的榜单，并向您进行推荐。

个性化推荐与检索类算法会基于模型预测您的偏好特征，匹配您可能感兴趣的商品、服务或其他信息，对向您展示的商品、服务或其他信息进行排序。我们会根据您使用产品过程中的浏览行为，对推荐模型进行实时反馈，不断调整优化推荐结果。为满足您的多元需求，我们会在排序过程中引入多样化推荐技术，拓展推荐的内容，避免同类型内容过度集中。

图 5-36　淘宝隐私权政策条款

在用户浏览商品的过程中，平台会自动收集多种与用户行为相关的信息，包括用户对商品的收藏行为、订阅信息及在特定商品页面的停留时间等。这些数据可以帮助推荐系统建立用户的兴趣模型，了解用户偏好的商品类型。同时，当用户通过搜索功能找到商品时，系统也会记录下与搜索相关的行为数据，包括搜索关键词、用户单击的商品及这些商品的收藏和订阅情况。这些信息的全面收集，为后续的推荐算法提供了丰富的训练数据。

因为淘宝的项目源代码是不公开的，因此针对淘宝推荐算法的模拟只是根据常识进行预测。在项目开发的过程中，用户在浏览淘宝时，查看系统推荐的商品，其在推荐算法中，权重相对来讲会高一点，但是用户通过搜索获取的商品，其停留时间所获得的权重，一般分配在产品类别中，即用户因为搜索某个产品的次数很多、停留时间很长，因此判断用户对这个类别的产品比较感兴趣。

综合来看，浏览停留时间和搜索停留时间各有优势，在不同的场景下都能提供有价值的推荐信息。搜索产品会让淘宝系统知道用户喜欢这类产品，推荐产品则用于强化淘宝现有的推荐权重，优先推荐用户喜欢的产品。如果用户单击了推荐的产品，则相应推荐值便

会增加，以增强淘宝推荐系统，加强下一次推荐成功的概率。通过综合利用这两类数据，并根据具体应用场景调整权重，可以构建一个更精确和全面的推荐系统。

在书写代码的过程中，首先需要进行数据预处理工作，通过 TF-IDF 算法将文本转换为数值，便于进行相似度计算，这样便可获得在整个文本中相对较为重要的文本内容。将文本转换为数值后，进行归一化操作，将所有数值局限在同一个范围内，便于计算和比较。

然后可以用于计算用户搜索历史与商品描述之间的相似度。这一步是非常重要的。比方说，用户想要购买一台“笔记本电脑”，但是仅仅在搜索栏中输入“笔记本”，这时，用户搜索的是“笔记本”，计算机可不可以识别到用户需要的是一台“笔记本电脑”，而不是单纯的本子，这便需要考虑到相似度。

注意： 余弦相似度是通过计算两个向量之间夹角的余弦值来衡量相似度的一种方法，值域为［-1，1］。1 表示两个向量完全相同，0 表示它们正交（没有相似性），-1 表示它们完全相反。

接着便可进行分配权重，并计算综合得分。

最后，在获取综合得分之后推荐商品。

推荐算法代码如下：

```
// 第 5 章 / 推荐算法 .py
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel

# 示例数据
search_histories = ["laptop", "gaming", "16GB RAM"]
product_descriptions = [
    "gaming laptop with 16GB RAM",
    "office laptop with 8GB RAM",
    "gaming desktop with 32GB RAM",
    "ultrabook with 8GB RAM",
    "laptop with 16GB RAM and 1TB SSD"
]
search_times = np.array([120, 80, 150])      # 用户在搜索结果页面的停留时间（秒）
browse_times = np.array([200, 180, 220])     # 用户在浏览商品页面的停留时间（秒）

#TF-IDF 特征提取
tfidf_vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf_vectorizer.fit_transform(product_descriptions)

# 计算用户搜索历史的 TF-IDF 向量
user_search_vector = tfidf_vectorizer.transform(search_histories)

# 计算商品之间的余弦相似度
```

```
cosine_similarities = linear_kernel(user_search_vector, tfidf_matrix)

# 计算推荐分数
search_weight = 0.4
browse_weight = 0.6

# 计算综合推荐分数
# 假设搜索分数和浏览分数加权平均
combined_scores = (cosine_similarities.max(axis=0) * search_weight +
                   np.mean(browse_times) * browse_weight)
                                          # 使用浏览时间的均值作为浏览分数

# 输出推荐分数最高的商品
recommended_indices = np.argsort(combined_scores)[::-1]
recommended_products = [product_descriptions[i] for i in recommended_indices]

print("推荐商品：")
for idx, product in enumerate(recommended_products):
print(f"{idx + 1}. {product} (综合分数：{combined_scores[recommended_indices
[idx]]:.2f})")
```

5.5.5 神经网络

神经网络是人工智能领域的重要分支，其基本构思源于人类大脑的神经元结构。通过模拟神经元的连接和互动，神经网络能够学习和提取数据中的复杂模式。近年来，神经网络的应用范围不断扩大，涵盖了从图像识别、自然语言处理到推荐系统等多个领域。尤其是近年来，基于神经网络的技术迅速发展，诸如 ChatGPT 这样的应用已经在社会生活的各方面产生了深远影响。

1. 神经网络概念

神经网络是一种模仿人类大脑神经元结构的计算模型，通过层级连接和权重调整来学习和识别数据中的复杂模式。

1）仿真

从字面意思来理解，仿真就是模仿真实环境，通过数学建模方式，对真实情况进行数理模拟，达到理论上和现实一样复杂的情况，进行测试。

为什么需要仿真？这是因为试错成本太高。比方说，我们现在要测试一辆汽车的安全性能，通常会进行撞击实验，而每次撞击，我们的消耗成本都是一辆汽车，如图 5-37 所示。如果这笔钱可以节省下来，就可以投入到其他开销当中。那么如何进行仿真？工程师会对汽车的各项性能、车速及撞击目标做一个相关系数的分析，并建立一个数学模型。在理论层面，将撞击造成的创伤控制在一个合理的范围之内，之后再进行实景测试。由此，再进行测试，便可以降低成本。但是，理论上满足碰撞要求，实际上风险仍然是很大的，因此会在驾驶舱放入假人，再进行测试，观察假人的受伤程度，全部满足标准后，方可进行大规模生产。

图 5-37　汽车碰撞测试图

2）神经网络

神经网络是深度学习中的概念。此概念来源于人类对生物中枢神经系统的观察，而后，做出的实验研究，并通过统计学、数学公式计算，对人类大脑的分析模型所做的相关分析。在一定程度上，可以说是人类对大脑的一种仿真技术。生物体的神经系统和人工神经网络都是信息处理和传递的系统。生物神经系统通过神经元和突触网络传递电化学信号，在大脑和身体各部分之间调节感知、运动和认知功能。类似地，神经网络利用人工神经元和连接，通过加权和激活函数处理输入数据，从而学习模式、预测结果或执行决策。神经网络出现后，在图像识别、语音识别、自然语言处理等诸多领域取得了巨大的成就。

2. 神经网络各类概念解析

神经网络是一种高度结构化的计算模型，通常由 3 种基本层次构成：输入层、隐藏层和输出层。输入层负责接收外部数据，将这些数据传递给网络的下一层。每个输入节点对应一个特征，以便网络能够有效地处理多维数据。隐藏层是网络内部的计算层，通常由多个神经元组成，这些神经元通过权重连接在一起。隐藏层的数量和每层神经元的数量可以根据具体应用的复杂性进行调整，以便模型能够捕捉到数据中的深层次模式。

1）输入层

输入层是神经网络中最初的层次，负责接收外部输入数据并将其传递至下一层，即隐藏层。每个输入节点对应于一个输入特征，因此输入层的节点数必须与输入数据的特征数相等。通过这种方式，输入层将原始数据转换为神经网络可以处理的格式，使后续的计算能够顺利地进行。

以对灰度图像进行分析为例，假设有一幅 28×28 像素的图像。这种图像可以被视为一个二维数据结构，其中包含了 784 像素。由于神经网络在处理数据时通常更倾向于使用

一维向量形式，而不是直接处理二维数据结构，因此必须对这个 28 × 28 像素的图像数据进行转换。这一过程涉及将图像的每行像素依次展平，从而生成一个长为 784 的一维数组。这种转换不仅有助于简化后续的运算，还使数据在神经网络中的传播更加高效。

因此，最终的输入层节点数为 784，这意味着输入层中将有 784 个神经元，每个神经元将负责接收对应的像素值。这些神经元将把各自接收的输入特征传递给隐藏层，以便神经网络能够进一步地对图像进行处理和分析。

将二维灰度图像转换为一维，并计算其节点个数的代码如下：

```
// 第 5 章 / 灰度图像分析 .py
import numpy as np

# 示例的 28x28 像素的灰度图像数据
# 假设这是一个简化的灰度图像数据，用随机数代替实际像素值
image_data = [
    [34, 0, 67, ..., 123],    # 第 1 行像素
    [255, 45, 76, ..., 89],   # 第 2 行像素
    # 其他行像素
    [12, 200, 34, ..., 255]   # 第 28 行像素
]

# 将二维图像数据展开为一维数组
flattened_image = np.array(image_data).flatten()

# 打印展开后的一维数组
print("展开后的一维数组：", flattened_image)
print("一维数组的长度：", len(flattened_image))

# 计算展开后的一维数组的节点数，即神经网络的输入层节点数
input_layer_nodes = len(flattened_image)

print("输入层节点数：", input_layer_nodes)
```

在计算机视觉和图像处理领域，图像的表示形式是至关重要的。常见的图像表示方式有位图和灰度图，这两者各自适用于不同类型的图像和应用场景。位图通常用于表示彩色图像，而灰度图则主要用于表示黑白图像。

位图图像的表示依赖于 3 种基本颜色，即红色（R）、绿色（G）和蓝色（B），这 3 种颜色经组合后形成了人眼所看到的各种色彩。每种颜色通道的值通常在 0 到 255 之间，这意味着每个通道可以提供 256 个不同的强度级别。通过组合这 3 种颜色的不同强度，位图能够准确地呈现复杂的图像细节，包括丰富的颜色层次和精细的纹理。

与位图像比，灰度图仅使用单一的颜色通道来表示图像，其强度值同样在 0 到 255 之间。0 表示完全黑色，255 表示完全白色，而中间的值则表示不同程度的灰色。由于只有一个通道，所以灰度图在存储空间上相对更加节省，并且在处理过程中也降低了计算复杂度。

在 Python 语言中用位图的形式表示红色的代码如下：

```
// 第 5 章 / 位图红色 .py
from PIL import Image

# 创建一个红色的位图（100x100 像素）
bitmap_image = Image.new('RGB', (100, 100), color=(255, 0, 0))

# 保存位图
bitmap_image.save('red_bitmap.jpg')

# 显示位图
bitmap_image.show()
```

在 Python 语言中用灰度图的形式表示灰色的代码如下：

```
// 第 5 章 / 位图红色 .py
from PIL import Image

# 创建一个灰度图（100x100 像素）
gray_image = Image.new('L', (100, 100), color=127)

# 保存灰度图
gray_image.save('gray_image.jpg')

# 显示灰度图
gray_image.show()
```

2）隐藏层

隐藏层位于输入层和输出层之间，是实现复杂数据学习和建模的关键。隐藏层的存在使神经网络能够捕捉数据中的非线性关系，从而提升模型的表现力和泛化能力。

隐藏层由多个节点组成，每个节点实际上是一个人工神经元。每个神经元用于对输入信号进行处理，并将其传递给下一层的神经元。在神经网络中，节点之间的连接意味着信息从一层传播到另一层，这些连接形成了神经元之间的网络结构。具体而言，连接不仅包括前一层和当前层之间的关系，还涵盖了当前层与下一层之间的相互作用。

每个连接都有一个权重值，这个权重值在数值上通常用阿拉伯数字表示。权重反映了连接强度的大小，决定了从一个神经元传递到另一个神经元时信号的影响程度。权重的调整是神经网络学习的核心，网络通过反向传播算法不断地更新权重，以最小化预测输出与实际目标之间的差异。这一过程使网络能够逐渐学习输入数据与输出数据之间的复杂映射关系。

以 28×28 像素的灰度图为例，这种图像可以通过输入层的 784 个节点进行表示。在当前案例中，输入层将每个像素值转换为一个输入特征，而隐藏层的多个节点则对这些输入特征进行更高层次的抽象。隐藏层的设计和结构对神经网络的性能至关重要，通常通过实验和交叉验证来优化隐藏层的节点数和层数。通过适当的设计，隐藏层能够有效地提取图像中的关键特征，为后续的输出层提供有价值的信息，从而实现更准确的分类或预测。

通过矩阵的方式来对隐藏层路径权重进行表示的代码如下：

```
// 第 5 章 / 隐藏层矩阵 .py
import numpy as np

# 输入层节点数
input_layer_size = 28 * 28  # 对应于 784 个输入特征

# 隐藏层节点数
hidden_layer_size = 128

# 初始化权重矩阵 W1，从输入层到隐藏层
# 形状为 （隐藏层节点数，输入层节点数）
W1 = np.random.rand(hidden_layer_size, input_layer_size)

# 打印权重矩阵的形状和一个示例
print("权重矩阵 W1 的形状：", W1.shape)
print("权重矩阵 W1 的示例值：", W1)

# 示例输入数据（28×28 像素的灰度图像展平为一维向量）
input_data = np.random.rand(input_layer_size)

# 计算隐藏层输入：加权和
# 矩阵乘法：隐藏层输入 = 权重矩阵 W1 × 输入数据
hidden_layer_input = np.dot(W1, input_data)

# 偏置向量
b1 = np.random.rand(hidden_layer_size)

# 加上偏置
hidden_layer_input += b1

# 激活函数（例如 ReLU）
def relu(x):
    return np.maximum(0, x)

# 计算隐藏层输出
hidden_layer_output = relu(hidden_layer_input)

# 打印隐藏层输入和输出的示例值
print("隐藏层输入的示例值：", hidden_layer_input)
print("隐藏层输出的示例值：", hidden_layer_output)
```

隐藏层矩阵输出结果示意图如图 5-38 所示。

在神经网络中，权重矩阵 $\boldsymbol{W}_1$ 是一个核心参数，是一个二维矩阵，表示从输出层到隐藏层所有连接的权重。每个元素 w_{ij} 代表从输入层第 i 个节点到隐藏层第 j 个节点的连接权重。$\boldsymbol{W}_1$ 的参数有两个，分别为 hidden_layer_size 和 input_layer_size，其中，hidden_layer_size 是隐藏层的节点数，input_layer_size 是输入层的节点数。因此观察上述例子，开始时，我们规定了隐藏层的节点数，即 128。同时，我们规定了输出层的节点数，由于这是一个 28×28 像素的灰度图像，其节点数为 784，因此最终权重矩阵 $\boldsymbol{W}_1$ 的形状便是 128×784。

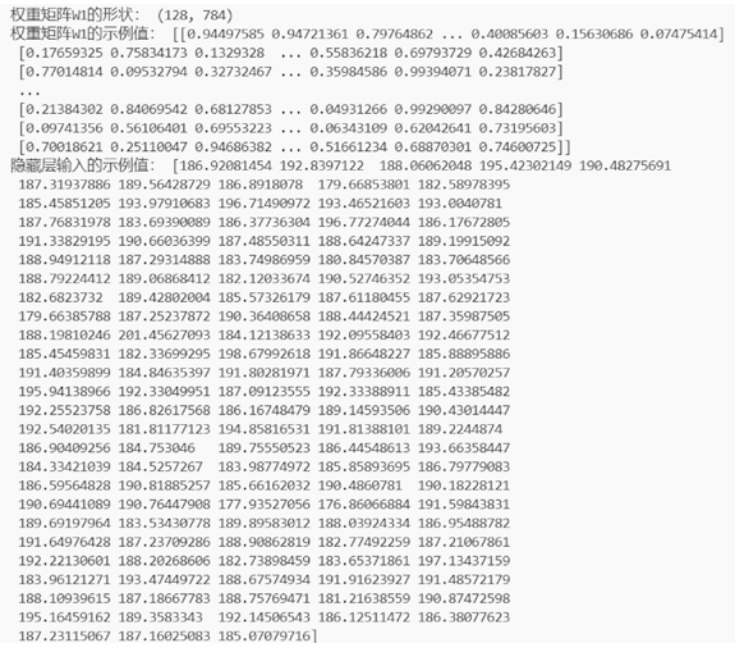


图 5-38　隐藏层矩阵输出结果示意图

隐藏层的计算过程是一个非常庞大且消耗算力的过程。已经得到输入层节点数是 184，隐藏层节点数是 128，权重矩阵的规模是 128 × 784。矩阵乘法可以看作对每个隐藏层节点计算一次加权和。最终，得出结果需要进行 200 704 次运算。

3）初始化

在神经网络的训练过程中，权重的初始化是一个至关重要的步骤。通常情况下，权重矩阵中的值被初始化在 0 与 1 之间，这一过程是为了确保网络能够有效地进行学习和建模。权重初始化的主要目的是减少数据的规模、打破对称性、避免梯度消失和梯度爆炸问题，并且帮助稳定整个训练过程。

在实际应用中，神经网络训练中常用的初始化方法有 3 种，分别是随机初始化、Xavier 初始化和 He 初始化。随机初始化是指将权重设置为较小的随机值。在进行灰度图像测试时，通常采用这种随机初始化的方法，这可以通过 Python 中的 NumPy 库中的 random( ) 方法轻松地实现。随机初始化方法简单易行，具有良好的可操作性，因此被广泛地应用于神经网络的构建中。然而，需要注意的是，这种方法适合浅层神经网络，对于深层神经网络而言，可能会面临梯度消失或梯度爆炸问题，从而影响模型的训练效果。

Xavier 初始化和 He 初始化的提出就是为了解决随机初始化的梯度消失或梯度爆炸问题。Xavier 初始化旨在保持每层输入和输出的方差一致，从而提高训练的稳定性。该方法根据前一层的神经元数量来设置权重的初始值，使权重的范围适应于网络结构。He 初始化则针对 ReLU 激活函数进行了优化，考虑了非线性因素的影响，从而进一步提高深层网络的训练效果。

随机初始化函数的代码如下：

```
// 第 5 章 / 随机初始化函数 .py
import numpy as np

def random_init(size, scale=0.01):
    return np.random.randn(*size) * scale

# 示例：初始化一个形状为 (2, 3) 的权重矩阵
W1 = random_init((2, 3))
print(W1)
```

Xavier 初始化是一种常用的权重初始化方法，特别适用于使用 Sigmoid 或 Tanh 激活函数的深层神经网络。这种初始化方法的主要目标是平衡网络每层的输入和输出的方差，以确保在前向传播过程中信号均衡，从而有效地避免梯度消失和梯度爆炸等问题。梯度消失通常会导致网络难以学习，而梯度爆炸则可能使模型参数发散，影响最终的学习效果。

Xavier 初始化的核心思想是使每层的输入和输出的方差保持一致。通过这种方式，可以避免信号在前向传播过程中被逐渐放大或缩小。这种均衡不仅提高了网络的稳定性，还加速了收敛过程，使模型在训练时能够更快地适应数据特征，从而提升整体性能。

为了达到这一目的，Xavier 初始化一般采用两种方法：均匀分布和正态分布。具体来讲，均匀分布的初始化方法通常将权重初始化为从一个小范围内随机抽取的数值，而这个范围的设置则与前一层神经元的数量有关。另一方面，正态分布的初始化则是根据前一层神经元数量的标准差来生成权重，这样能确保生成的权重值在一个期望的均值附近波动。

Xavier 均匀分布初始化的代码如下：

```
// 第 5 章 /Xavier 均匀分布初始化 .py
import numpy as np

def xavier_uniform_init(size):
    n_in, n_out = size
    limit = np.sqrt(6 / (n_in + n_out))
    return np.random.uniform(-limit, limit, size=size)

# 示例：初始化一个形状为 (3, 2) 的权重矩阵
W1 = xavier_uniform_init((3, 2))
print(W1)
```

Xavier 正态分布初始化的代码如下：

```
// 第 5 章 /Xavier 正态分布初始化 .py
import numpy as np

def xavier_normal_init(size):
    n_in, n_out = size
    stddev = np.sqrt(2 / (n_in + n_out))
```

```
    return np.random.randn(*size) * stddev

# 示例：初始化一个形状为 (3, 2) 的权重矩阵
W1 = xavier_normal_init((3, 2))
print(W1)
```

He 初始化也是一种专门用于解决神经网络在训练过程中梯度消失和梯度爆炸问题的权重初始化方法，尤其适用于使用 ReLU（Rectified Linear Unit）及其变种作为激活函数的深层神经网络。这种初始化方式的有效性主要体现在能够保持信号在前向传播和反向传播过程中的稳定性，进而提升模型的训练效率和准确性。

具体流程包括几个重要步骤：首先，需要确定权重矩阵的大小，这通常与神经网络的架构及每层神经元的数量有关。权重矩阵的大小会直接影响模型的学习能力和最终的预测性能，因此在设计网络结构时，需要对每层的神经元数量进行合理规划。其次，需要计算方差，这个计算是基于对 ReLU 激活函数特性的分析，其目的是在前向传播时保持信号的有效传递，同时减少非线性引入后对梯度计算的不利影响。最后，根据经计算得到的标准差，从正态分布中随机生成权重。

He 初始化方法的代码如下：

```
// 第 5 章 /He 初始化 .py
import numpy as np

def he_normal_init(size):
    n_in, n_out = size
    stddev = np.sqrt(2 / n_in)
    return np.random.randn(*size) * stddev

# 示例：初始化一个形状为 (3, 2) 的权重矩阵
W1 = he_normal_init((3, 2))
print(W1)
```

4）浅层神经网络和深层神经网络

神经网络，无论是浅层还是深层，计算量都是巨大的。在日常生活中，可以用到的计算量大多不超过十万，即使如此，对计算机的负荷也是非常大的。通常情况下，在笔记本电脑上，这样的计算量，大多需要数小时才能完成，因此神经网络，即使是浅层神经网络，对计算机配置的要求也会很高。如果想要训练神经网络模型，则可以借助云服务器等手段进行完成。同时，除提升计算机配置以外，了解浅层及深层神经网络的结构，并且可以粗略地对其计算量做出一个判断，对后续研究也是非常有帮助的。

浅层神经网络的定义是指包含一至两个隐藏层的神经网络。由于其结构较为简单，所以浅层神经网络在功能实现上通常适用于处理相对简单的任务。相比于深层神经网络，浅层神经网络的隐藏层数量较少，这使其构造和理解相对容易。这样的结构设计不仅降低了模型的复杂性，同时也简化了网络的训练过程，从而提高了模型的可解释性。

在计算机进行数据处理时，每次运算都会消耗计算资源，尤其是在面对大型数据集或复杂模型时，运算的开销会显著地增加。浅层神经网络的优势在于其计算量较少，运行效率较高。由于其层数和节点数的限制，训练和推理所需的计算量相对较低，因此其运行时间显著短于深层网络。在实践中，这使浅层神经网络在执行简单任务时更加高效。

深层神经网络通常被定义为具有 3 个或更多隐藏层的神经网络。这类网络由于隐藏层数量较多，结构相对复杂，因此在处理信息时能够提取更高层次的特征，适用于更复杂的计算任务。相较于浅层神经网络，深层神经网络在构建和训练过程中需要更多的计算资源，这使其运行开销显著增加。这种复杂性不仅增加了计算时间，还对计算机的硬件配置提出了更高的要求。

深层神经网络之所以被广泛地应用于诸如图像识别、自然语言处理和语音识别等领域，正是因为其能够通过多层次的抽象能力更有效地处理复杂的数据。这些任务通常涉及大量的输入特征和高维数据，深层网络能够通过其多层结构逐层提取特征，从而实现更高的准确性和稳健性。例如，在图像识别中，深层网络可以从低级特征（例如边缘和角点）逐步学习到高级特征（例如物体和场景），这对于识别的准确性至关重要。

使用 Keras 库构建并编译一个简单的神经网络模型的代码如下：

```
// 第 5 章 / 深层神经网络 .py
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D

# 创建一个顺序模型
model = Sequential()

# 添加卷积层和池化层
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=
(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))

# 添加更多的卷积层和池化层
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# 展平层
model.add(Flatten())

# 添加全连接层
model.add(Dense(128, activation='relu'))

# 添加输出层
model.add(Dense(10, activation='softmax'))

# 编译模型
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=
['accuracy'])
```

5）偏置向量

在神经网络中，除了输出层都需要计算权重。偏置向量便是神经网络中计算权重的一个参数。它的作用是在输入数据经过权重矩阵线性变换后，添加一个额外的参数，从而调整模型的输出。这有助于模型更好地拟合数据。

偏置向量的公式为

$$z=Wx+b \tag{5-16}$$

其中，z 表示每层神经网络的输出；W 表示权重矩阵；x 表示输入向量（由上一层向这一层进行输出）；b 表示偏置向量。

偏置向量 b 的作用就是调整每个神经元的激活值，使神经元的输出不仅依赖于输入数据的加权和，还能够进行一个额外的平移。这样可以帮助模型更好地捕捉数据的复杂模式。

具体来讲，在神经网络中，每个神经元都会接收来自前一层的输入数据。每个输入数据都会先乘以一个对应的权重，然后将这些乘积加在一起，这就是所谓的加权和，但是仅仅有加权和是不够的，因为纯靠公式算出来的数据太过于理想，因此需要设置一些偏移量。偏移量是一个额外的参数，每个神经元都有自己的偏移量。偏移量允许在计算加权和之后进行一个固定的平移。通过添加偏移量，每个神经元的输出可以进行一个额外的平移，使模型能够更灵活地拟合数据。由此，用户可以捕捉到数据的复杂性。

定义具有偏移量的神经网络层，代码如下：

```
// 第 5 章 / 神经网络偏移量 .py
from keras.models import Sequential
from keras.layers import Dense

# 创建一个顺序模型
model = Sequential()

# 添加输入层和第 1 个隐藏层，使用默认的偏置向量
model.add(Dense(units=32, input_dim=64, activation='relu'))

# 添加输出层，使用默认的偏置向量
model.add(Dense(units=10, activation='softmax'))

# 编译模型
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=
['accuracy'])
```

6）激活函数

激活函数（Activation Function）是神经网络中的一个关键组件，它们被应用于神经网络层中的每个神经元的输出，决定了是否及如何将信号传递到下一层。

激活函数的主要作用有两个：非线性引入和激活或抑制。在没有激活函数的情况下，神经网络的每层只是对上一层的线性变换，这样的网络即使堆叠了多层，也只能表示线性

关系。激活函数通过引入非线性，使神经网络能够学习复杂的模式。

举个例子，今天，我们决定去爬山，但是山路地形崎岖。如果我们非要选择一条直线走上去，看似会快，但是中间可能会遇到很多困难，导致我们没有办法进行下去，以至于爬不上山顶；但是如果我们换个角度，选择曲线爬山，当碰到较为陡峭的地形时就绕过去，这样看似慢了，但实则更有利于我们爬上山顶。

常用的激活函数有 3 种，分别是 Sigmoid 激活函数、Tanh 激活函数和 ReLU 激活函数。

Sigmoid 激活函数的公式如下：

$$\sigma(x)=\frac{1}{1+e^{x}} \tag{5-17}$$

其中，σ 在神经网络中表示激活函数；e 表示自然对数，约等于 2.718 28。

Sigmoid 激活函数的输出值介于 0~1，呈 S 形曲线。当输入值接近 0 时，输出值为 0.5；当输入值变大或变小时，输出值分别趋于 1 和 0，如图 5-39 所示。

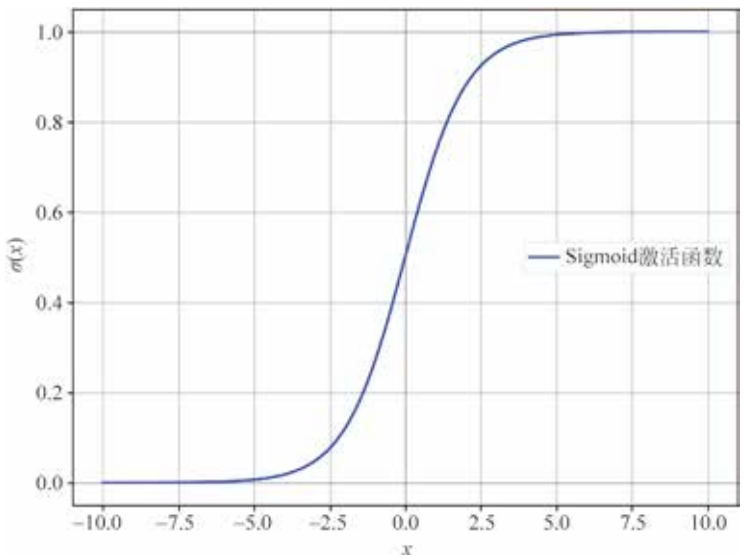


图 5-39　Sigmoid 激活函数曲线图

Sigmoid 激活函数实现过程的代码如下；

```
// 第 5 章 /Sigmoid 激活函数 .py
import numpy as np
import matplotlib.pyplot as plt

# 定义 Sigmoid 激活函数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 生成 x 值
```

```
x = np.linspace(-10, 10, 400)
# 计算 y 值
y = sigmoid(x)

# 绘制 Sigmoid 激活函数曲线
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Sigmoid Function', color='blue')
plt.title('Sigmoid Activation Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (σ(x))')
plt.axhline(0, color='gray', lw=0.5)
plt.axhline(1, color='gray', lw=0.5)
plt.axvline(0, color='gray', lw=0.5)
plt.legend()
plt.grid(True)
plt.show()
```

Tanh 激活函数的图像也是一条曲线，形状和 Sigmoid 激活函数类似，但是范围却是不同的，其输出值在 −1~1，使数据的均值更接近于零，这有助于加快神经网络的训练收敛速度，因此在一些神经网络的隐藏层中被广泛使用。

Tanh 激活函数的公式如下：

$$\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{5-18}$$

Tanh 激活函数是一个平滑的函数，其在 x 轴上是单调递增且关于原点对称的。函数图像如图 5-40 所示。

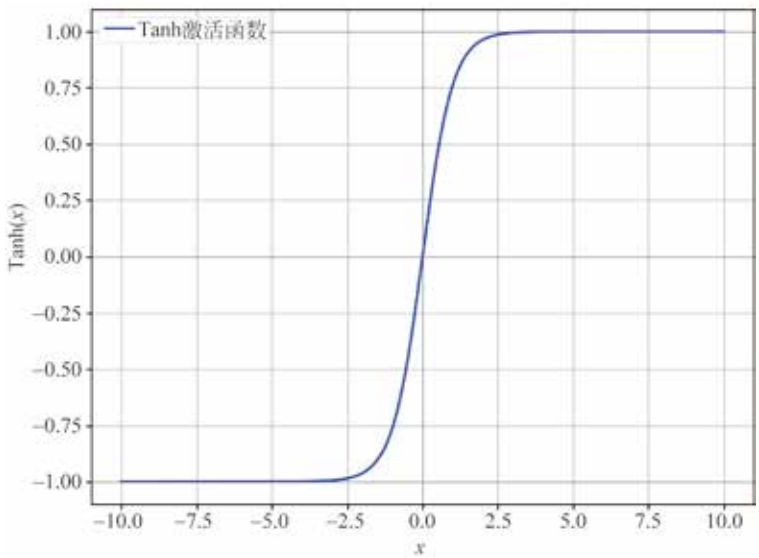


图 5-40　Tanh 激活函数图像

Tanh 激活函数实现过程的代码如下：

```
// 第 5 章 /Tanh 激活函数 .py
import numpy as np
import matplotlib.pyplot as plt

# 定义 Tanh 激活函数
def tanh(x):
    return np.tanh(x)

# 生成 x 值
x = np.linspace(-10, 10, 400)
# 计算 y 值
y = tanh(x)

# 绘制 Tanh 激活函数曲线
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Tanh Function', color='blue')
plt.title('Tanh Activation Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (tanh(x))')
plt.axhline(0, color='gray', lw=0.5)
plt.axhline(1, color='gray', lw=0.5)
plt.axhline(-1, color='gray', lw=0.5)
plt.axvline(0, color='gray', lw=0.5)
plt.legend()
plt.grid(True)
plt.show()
```

ReLU 激活函数计算非常简单，不涉及复杂的数学运算，因此计算效率高。ReLU 激活函数被广泛地应用于各种深层神经网络的隐藏层，包括卷积神经网络（CNN）和循环神经网络（RNN）。它的输出范围介于 0 和正无穷之间。尽管公式看起来很简单，但 ReLU 激活函数引入了非线性特性，使神经网络能够拟合复杂的数据分布。当输入值为负时，ReLU 激活函数的输出为 0，这意味着它可以在神经网络中引入稀疏性，使部分神经元在某些情况下不激活，有助于缓解过拟合，但是当输入值总是负数时，ReLU 激活函数的输出为 0，对应的导数也为 0，这意味着这些神经元在反向传播过程中无法更新，导致这些神经元“死亡”。

ReLU 激活函数的公式如下：

$$\mathrm{ReLU}(x)=\max(0,x) \tag{5-19}$$

ReLU 激活函数在非负区间是单调递增的，尽管它看起来像线性函数，但由于它在负区间输出为 0，故引入了非线性特性。函数图像如图 5-41 所示。

ReLU 激活函数实现过程的代码如下：

```
// 第 5 章 /ReLU 激活函数 .py
import numpy as np
import matplotlib.pyplot as plt
```

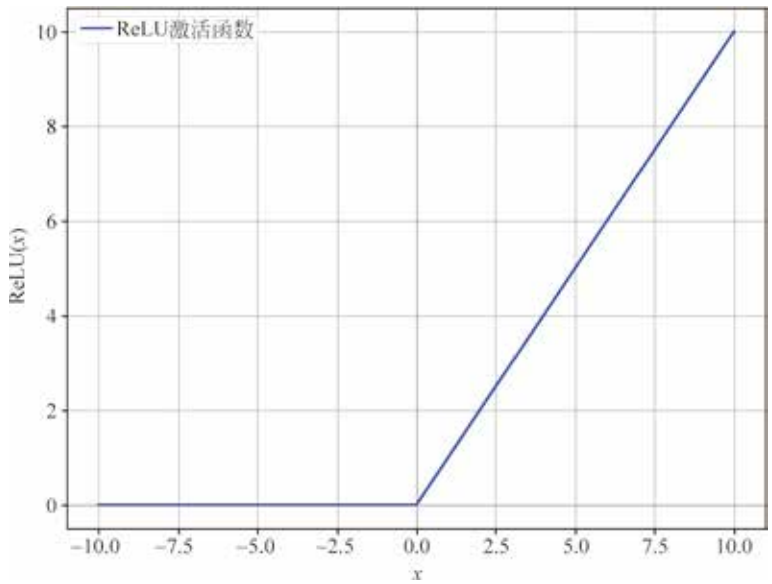


图 5-41　ReLU 激活函数图像

```
# 定义 ReLU 激活函数
def relu(x):
    return np.maximum(0, x)

# 生成 x 值
x = np.linspace(-10, 10, 400)
# 计算 y 值
y = relu(x)

# 绘制 ReLU 激活函数曲线
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='ReLU Function', color='blue')
plt.title('ReLU Activation Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (ReLU(x))')
plt.axhline(0, color='gray', lw=0.5)
plt.axvline(0, color='gray', lw=0.5)
plt.legend()
plt.grid(True)
plt.show()
```

7）输出层

神经网络的输出层是神经网络的最后一层，负责产生模型的最终输出。输出层的设计取决于解决的问题类型和所需的预测输出。

主要的输出类型分为两种：一种是回归输出型；另一种是分类输出型。两种不同的输

出方式，在后续诸多操作上都有很大的不同。

回归任务的输出层通常只有一个节点，这是因为模型需要预测的是一个连续值。激活函数通常是一个线性函数，或者不使用激活函数，即将加权输入的线性组合直接作为输出。这样设计的原因是为了保持输出值的连续性，可以满足任意实数范围内的输出需求。回归输出通常应用于客户行为预测、房价预测、股票价格预测等。

分类任务的输出层通常有多个节点，每个节点对应一个可能的类别。输出层的激活函数通常使用 Softmax 激活函数。Softmax 激活函数能够将每个节点的输出压缩到 0 到 1 之间，并确保所有输出节点的总和为 1，因此可以解释为每个类别的概率。输出层的主要任务是将神经网络学习到的特征映射为每个类别的概率分布，应用于图像分类、文本分类、语音识别、疾病诊断等。

常见的激活函数分为 3 种：①线性激活函数，通常应用于回归任务，直接输出加权输入的线性组合；② Softmax 激活函数，适用于多分类任务，将输出转换为每个类别的概率分布，确保所有输出值在 0~1 且总和为 1；③ Sigmoid 激活函数，适用于二分类任务，将输出压缩到 0~1，表示某个类别的概率。

在输出层的选择上，大致可以分为 3 种：①回归任务，通常使用线性激活函数，直接输出数值；②二分类任务，输出层使用 Sigmoid 激活函数，单个节点输出二分类概率；③多分类任务，输出层使用 Softmax 激活函数，多个节点输出各个类别的概率。

在输出层损失函数的选择上，回归任务通常使用均方误差（MSE）作为损失函数；分类任务通常使用交叉熵损失函数（Cross-entropy）。例如，对于二分类任务可以使用二元交叉熵，对于多分类任务可以使用多元交叉熵。

在整个第 2 小节中，我们大体上对输入层、隐藏层、输出层进行了详细介绍，并对其中计算的关键数据进行了详细解释，例如初始化、深层 / 浅层神经网络、偏置向量、激活函数等。在了解了这些基础概念之后，当遇到一个新的神经网络模型时各位读者也可以自行尝试对其进行拆解。

3. 神经网络模型介绍

神经网络模型是机器学习的重要组成部分。在理解了神经网络的各类概念之后，本节将对当今比较流行的神经网络模型加以介绍，主要介绍 5 个神经网络模型：卷积神经网络（Convolutional Neural Network，CNN）、循环神经网络（Recurrent Neural Network，RNN）、长短时记忆模型（Long Short-Term Memory，LSTM）、Transformer 模型及 GPT 模型。

1）卷积神经网络

卷积（Convolution）在卷积神经网络中是一种重要的数学运算，它用于将卷积核（Filter）应用于输入数据，以此来提取数据中的特征。这一过程是通过在输入数据上滑动卷积核，并在每个位置进行点积运算来实现的。每当卷积核在输入数据上滑动到一个新的位置时都会计算该位置上卷积核与输入数据对应部分的加权和。最终，这一过程生成一张特征图（Feature Map），该特征图能够突出输入数据中的重要模式和特征，从而帮助后续的层更有效地进行信息处理。

为了更好地理解这一概念，可以参考一个简单的例子。在卷积神经网络中，通常会有 3 个基本部分：输入层、隐藏层和输出层。在卷积神经网络中，输入层通常是图像或与图像类似的多维数据。在这一层，输入的图像可以用一个二维数组来表示，其中每个元素对应于图像的一像素值。例如，对于一幅灰度图像，输入的二维数组可以表示为一个 $m \times n$ 的矩阵，其中 m 和 n 分别是图像的高度和宽度。图像的像素值范围通常在 0~255，表示图像的亮度信息，如图 5-42 所示。

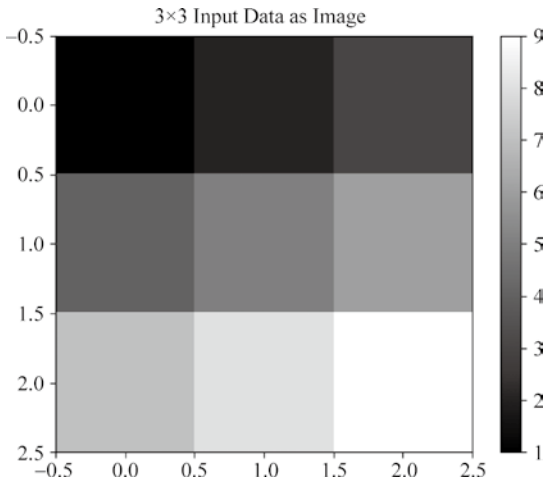


图 5-42　卷积神经网络输入数据图

由于图片是不能参与运算的，因此图片还需被转换为数值，方可继续进行。将图像转换为数值的代码如下：

```
// 第 5 章 / 卷积神经网络输入 .py
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# 定义 3×3 的输入数据
input_data = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# 将数据转换为图像并保存
plt.imshow(input_data, cmap='gray')
plt.title('Input Data as Image')
plt.colorbar()
plt.savefig('input_image.png', dpi=300)        # 保存图像时使用更高的 DPI
plt.show()

# 重新加载图像并转换为数组
image_path = 'input_image.png'
image = Image.open(image_path).convert('L')   # 将图像转换为灰度
image_array = np.array(image, dtype=np.uint8)

# 设置打印选项以显示所有数组元素
np.set_printoptions(threshold=np.inf)

# 将数组保存到文本文件
output_file = 'image_array.txt'
```

```
    with open(output_file, 'w') as f:
        f.write("Reconstructed array from image:\n")
        f.write(np.array2string(image_array, separator=', '))

    print(f"Saved reconstructed array to {output_file}")
```

```
Reconstructed array from image:
[[255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]
 ...
 [255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]]
```

图 5-43　图片转数组结果图

在代码及输出结果中，有两点需要格外注意：第一点，因为没有现成的图片，因此图片一开始用数组生成。之后，用数组再对图片进行解析，那么按照逻辑来讲，前后的数值应该是一样的。第二点，用数组对生成的图片进行提取，此时会发现结果并不一样，如图 5-43 所示。

在图像处理的过程中，常常将原始图像转换为数组，以便进行后续计算和分析，然而，有时在这一过程中，观察到输出的数组中所有的值都变成了 255。255 在灰度图像中表示白色，估计不少初次遇到的读者会比较疑惑：是运算中出现了问题，还是我们的逻辑有误？为什么会出现大面积的白色？

事实上，这种现象的出现通常并不是运算错误的表现，而是由于图像的边缘区域存在大量的空白像素，这些像素通常被赋予最大值 255。具体而言，当我们处理一张图像时，尤其是当图像被生成或裁剪时，其周围可能会留下一圈白色的空白区域。这些空白区域的像素值被设定为 255，表示没有有效的图像信息。因此，最终生成的数组可能会在周边部分包含大量的 255 值，从而导致整体看起来都是白色的情况。

为了解决这一问题，就需要读者下一番功夫去深入挖掘真实的图像数据。具体来讲，真实有效的图像信息可能被隐藏在输出结果的省略号部分中，如图 5-43 所示。为了进一步分析这些数据，可以将省略号所代表的结果全部展开。展开后的结果可以保存到一个文本文件中，以便后续进行查看和分析，正如图 5-44 所示。这种方式能够更清晰地观察到图像中实际存在的特征。

```
85518   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85519   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85520   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85521   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85522   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85523   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85524   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85525   160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
85526   133,   0,   0,   0, 212, 255, 255, 255, 255, 255, 255, 255, 255, 255,
85527   255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
85528   255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
85529   255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
85530   255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
85531   255, 255, 255, 255, 255, 212,   0,   0,   0, 126, 152, 152, 152, 152,
85532   152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152,
85533   152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152,
85534   152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152,
85535   152, 152, 152, 152, 126,   0,   0,   0, 212, 255, 255, 255, 255, 255,
85536   255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
85537   255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
```

图 5-44　输出结果图

在卷积神经网络的架构中，隐藏层通常包括多个层级，其中包括卷积层、激活层、池化层及全连接层。每层在网络中都扮演着独特的角色，共同致力于从输入数据中提取有效的特征，并最终实现目标任务的预测或分类。

卷积层是卷积神经网络中的核心部分，其主要功能是通过特征提取来识别输入数据中的模式和特征。卷积层的操作通常可以分为 4 个步骤：首先需要定义输入图像的局部区域，其次进行点积计算，然后滑动卷积核，并最终生成特征图。

仍以 3 × 3 像素输入图像为例，输入内容是一个 3 × 3 像素的数组，而卷积核是 2 × 2 的，如图 5-45 所示。进行点积计算，将 2 × 2 的卷积核套在输入层上，其中输入层的 1 和卷积核的 1 相乘，输入层的 2 和卷积核的 0 相乘，输入层的 4 和卷积核的 0 相乘，输入层的 5 和卷积核的 1 相乘。最后，将之前所有相乘所得结果的积相加，得出结果为 6。

之后滑动卷积核，将卷积核向右边滑动，滑动完之后，再向下方滑动，如图 5-46 所示。第 1 次卷积核计算得到的结果为 6，第 2 次移动后得到的结果为 8，第 3 次得到的结果为 12，第 4 次得到的结果为 14。具体运算过程如图 5-46 所示，最终生成特征图。

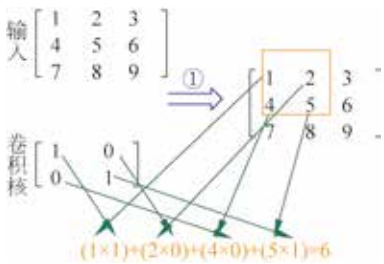


图 5-45　卷积第 1 层

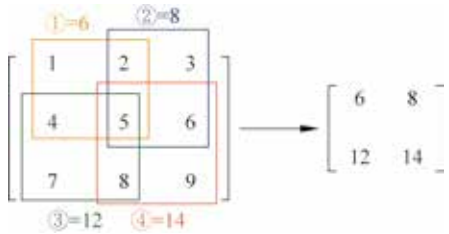


图 5-46　卷积结果

卷积结果可视化如图 5-47 所示，顺序遵循图 5-46 中的计算过程，以及图 5-45 中的计算逻辑。

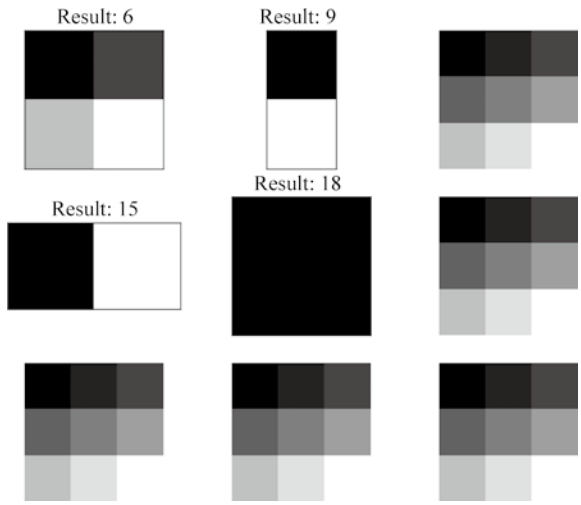


图 5-47　卷积结果可视化

卷积结果计算及可视化过程的代码如下：

```
// 第 5 章 / 卷积 .py
import numpy as np
import matplotlib.pyplot as plt

# 定义输入图像和卷积核
input_image = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

filter_kernel = np.array([
    [1, 0],
    [0, 1]
])

# 计算卷积
def convolve2d(image, kernel, stride=1):
    kernel_size = kernel.shape[0]
    output_size = ((image.shape[0] - kernel_size) //stride) + 1
    output = np.zeros((output_size, output_size))

    for y in range(0, image.shape[0] - kernel_size + 1, stride):
        for x in range(0, image.shape[1] - kernel_size + 1, stride):
            region = image[y:y+kernel_size, x:x+kernel_size]
            output[y//stride, x//stride] = np.sum(region * kernel)

    return output

# 可视化卷积过程
def plot_convolution(input_image, filter_kernel, output_image):
    fig, axes = plt.subplots(3, 3, figsize=(10, 10))

    for i in range(3):
        for j in range(3):
            axes[i, j].imshow(input_image, cmap='gray', interpolation='none')
            axes[i, j].set_xticks([])
            axes[i, j].set_yticks([])
            if i < 2 and j < 2:
                y = i * 2
                x = j * 2
                region = input_image[y:y+2, x:x+2]
                axes[i, j].imshow(region, cmap='gray', interpolation='none')
                result = np.sum(region * filter_kernel)
                axes[i, j].set_title(f"Result: {result}")
            else:
```

```
                    axes[i, j].axis('off')

        plt.show()

    # 计算卷积结果
    output_image = convolve2d(input_image, filter_kernel)

    # 可视化卷积过程
    plot_convolution(input_image, filter_kernel, output_image)

    print("卷积结果:")
    print(output_image)
```

注意： 因为目前是 3×3 矩阵，并且卷积核是 2×2，因此需要计算 4 次。如果还是 3×3 矩阵，但是卷积核变为了 3×1，则只需计算 3 次，中间的逻辑过程读者可以自行推理。

卷积神经网络激活层的常见激活函数便是 ReLU 激活函数，在神经网络一章的第 2 节里的激活函数中，有详细介绍，读者可以自行查阅，这里不再进行介绍。

池化是一个很生动的说法。就好比整个数据是一个大水坑，在这个水坑中，我们选择一些数据进行计算，但是因为数据量太大了，根本计算不过来，于是就在这片区域中随机选择一片数据，类似于随机抽样的形式，用部分来代表整体。池化层用于减少卷积神经网络中特征图的空间尺寸，同时保留重要特征。它是通过降低维度的方式来减小计算量的。常见的操作有两个：最大池化和平均池化。最大池化是从池化窗口内选择最大值作为输出；平均池化是从池化窗口内计算平均值作为输出。

全连接层通常位于卷积神经网络的最后几层。它先将前一层输出的特征图展平成一个向量，然后通过一个或多个全连接层进行处理，用于综合特征并进行分类或其他任务。它主要由两部分构成：第一是神经元，每个神经元都要与上一层的神经元相互联系；第二便是权重和偏置，每个连接都有相应的权重和偏置，通过训练过程不断调整。

经过输入、卷积、池化和全连接 4 个关键步骤，最终的结果便是网络对输入数据的分类或预测输出。在整个过程中每个步骤都至关重要，确保网络能够有效地从原始输入中提取并处理信息，从而实现更高效的学习和准确的预测。

可以用卷积神经网络实现一个简易的手写数字识别系统。首先需要导入必要的 TensorFlow、Keras 和 Matplotlib 等第三方库，其中 TensorFlow 和 Keras 用于构建和训练神经网络，Matplotlib 用于可视化训练过程。之后加载数据及完成数据预处理工作，然后使用 Keras 的 Sequential 模型构建一个卷积神经网络，网络结构包括多个卷积层、池化层和全连接层。接着，使用 Adam 优化器计算损失函数。模型训练 5 个周期，使用验证集监控训练效果。训练效果会通过进度条及计算结果的形式呈现出来。最后进行保存及可视化操作，如图 5-48 所示。

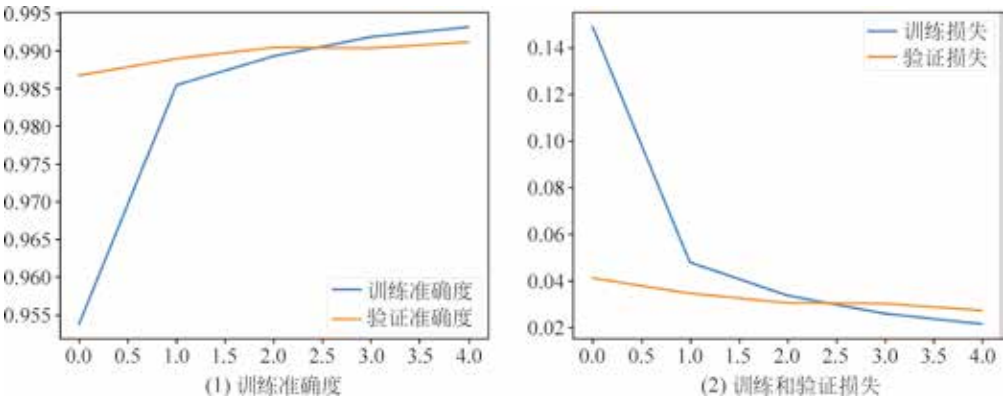


图 5-48　训练结果可视化图

训练过程及结果展示的代码如下：

```
// 第 5 章 / 卷积神经网络项目 .py
import tensorflow as tf
from tensorflow.keras import layers, models, datasets
import numpy as np
import matplotlib.pyplot as plt

# 加载和预处理数据
def load_and_preprocess_data():
    (train_images, train_labels), (test_images, test_labels) = datasets.
mnist.load_data()
    train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') /
255
    test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') /
255
    return (train_images, train_labels), (test_images, test_labels)

    (train_images, train_labels), (test_images, test_labels) = load_and_
preprocess_data()

# 构建卷积神经网络模型
def build_cnn_model():
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,
28, 1)))
   model.add(layers.MaxPooling2D((2, 2)))
   model.add(layers.Conv2D(64, (3, 3), activation='relu'))
   model.add(layers.MaxPooling2D((2, 2)))
   model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10, activation='softmax'))
```

```
    return model

model = build_cnn_model()

# 编译模型
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# 训练模型
history = model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))

# 评估模型
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')

# 保存模型
model.save('cnn_mnist_model.h5')

# 可视化训练过程
def plot_history(history):
    plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')

plt.show()

plot_history(history)
```

注意： 在代码的导入过程中会提示报错，但是不影响运行结果，并且报错行不可删除。

上述代码实现的是手绘数字识别的过程，但是仅仅停留在训练，而没有投入到使用当中。将上述代码在外面加入一个 UI 界面之后，便可以投入到实际使用中，如图 5-49 所示。其中，Load Image 表示加载图片，在 UI 右边是手绘的数字 1。将右边的图片加载到 UI 界面

当中，在 UI 界面的下方便会出现一个预测值，目前的预测值为 1。与输入的手绘数字相符。

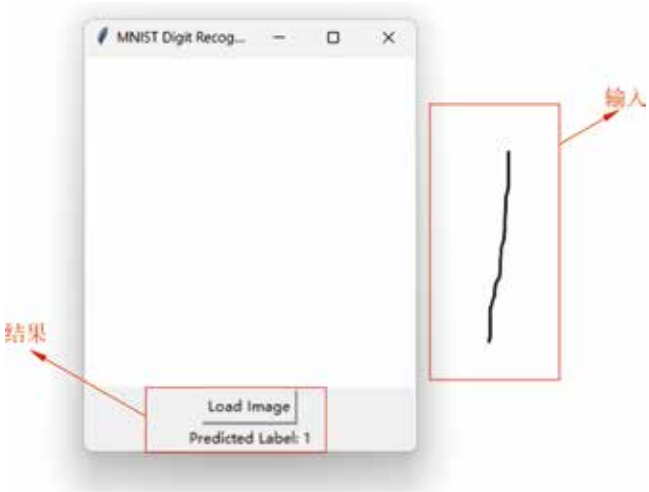


图 5-49　UI 界面显示

UI 界面实现过程的代码如下：

```
// 第 5 章 / 卷积神经网络项目 UI.py
import tkinter as tk
from tkinter import filedialog, messagebox
from PIL import Image, ImageTk
import numpy as np
import tensorflow as tf

# 加载预训练模型
model = tf.keras.models.load_model('cnn_mnist_model.h5')

# 预处理图像函数
def preprocess_image(image):
image = image.resize((28, 28)).convert('L')
image = np.array(image)
  image = image.reshape(1, 28, 28, 1).astype('float32') / 255
  return image

# 预测函数
def predict_image(image):
    preprocessed_image = preprocess_image(image)
  predictions = model.predict(preprocessed_image)
  predicted_label = np.argmax(predictions)
  return predicted_label

# 加载图像函数
def load_image():
```

```
        file_path = filedialog.askopenfilename()
        if not file_path:
            return
    image = Image.open(file_path)
    photo = ImageTk.PhotoImage(image)
    canvas.create_image(0, 0, anchor=tk.NW, image=photo)
    canvas.image = photo
    label = predict_image(image)
    result_label.config(text=f'Predicted Label: {label}')

# 创建主窗口
root = tk.Tk()
root.title("MNIST Digit Recognizer")

# 创建 Canvas 来显示图像
canvas = tk.Canvas(root, width=280, height=280)
canvas.pack()

# 创建按钮来加载图像
load_button = tk.Button(root, text="Load Image", command=load_image)
load_button.pack()

# 创建标签来显示预测结果
result_label = tk.Label(root, text="Predicted Label: ")
result_label.pack()

# 运行主循环
root.mainloop()
```

2）循环神经网络

与循环神经网络相对的概念是前馈神经网络。前馈神经网络是最基本和最简单的神经网络类型，它们在结构上没有循环或反馈连接，信息在网络中单向流动，从输入层经过隐藏层最终到达输出层。而循环神经网络则不同，它的信息可以循环流动，当前时间步的信息依赖于之前时间步的状态。因为它可以倒回去寻找信息，所以具有记忆能力，可以利用之前时间步的信息来影响当前输出。正因为循环神经网络具有这些特点，所以适用于处理序列数据，例如自然语言处理、语音识别、时间序列预测。后续将讲的长短期神经网络模型便是循环神经网络的一个变种。

循环神经网络的输入通常是一个序列数据，每个输入值都有一个输入向量。在文本处理中，输入可以是词序列；在时间序列模型中，输入可以是时间步序列。值得注意的是，循环神经网络的序列，可以不固定长度，即它的长度是可变的。但是，虽然长度是可变的，但是输入的顺序是有讲究的，需要输入数据按时间步逐步输入网络中，每个时间步的数据点依次传递给循环神经网络。循环神经网络数据输入的容器，通常是一个列表。当然，也可以将它们转换为数组，这样计算效率会更高。

一个简单的循环神经网络的隐藏层大致分为 3 部分：嵌入层、LSTM 层和全连接层。其中，嵌入层主要涉及将字符索引转换为向量表示；LSTM 层负责处理序列数据；全连接层负责最终输出预测结果。

假设，现在要实现一个自动创作古诗的系统，那么首先便需要寻找古诗作为输入，然后进行训练，之后再输出。我们选择两首诗进行输入（这里仅仅是输入层），分别是李白的《静夜思》和孟浩然的《春晓》。由于文本是不能直接参与运算的，因此需要将文本转换为向量值，方可参与运算。针对每首古诗，将其转换为一个序列（由字符索引组成的列表），然后生成多个输入序列和对应的目标，这就涉及隐藏层的工作了。

将输入集文本转换为向量的代码如下：

```
// 第 5 章 / 循环神经网络输入 .py
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# 示例古诗数据
poems = [
    "春眠不觉晓，处处闻啼鸟。",
    "夜来风雨声，花落知多少。",
    "床前明月光，疑是地上霜。",
    "举头望明月，低头思故乡。"
]

# 将文本转换为序列数据
tokenizer = Tokenizer(char_level=True)    # 逐字级别
tokenizer.fit_on_texts(poems)
sequences = tokenizer.texts_to_sequences(poems)
word_index = tokenizer.word_index

# 生成输入序列和目标
maxlen = 5                                # 序列长度
X = []
y = []

for seq in sequences:
    for i in range(1, len(seq)):
        input_seq = seq[:i]
        target = seq[i]
        X.append(input_seq)
        y.append(target)

# 填充序列
X = pad_sequences(X, maxlen=maxlen)
y = np.array(y)

# 打印数据
```

```
print("Tokenized sequences:")
for i, sequence in enumerate(x):
    target = y[i]
    print(f"Input sequence: {sequence}, Target: {target}")
```

运算结果如图 5-50 所示。在序列数据的示例中，每个数字代表经过 Tokenizer 编码的字符。Tokenizer 将每个字符映射到一个唯一的整数编码。经过 Tokenizer 处理后，每个字符被编码为一个整数。例如，假设字符“春”被编码为 7，字符“眠”被编码为 8，以此类推。这些编码是根据字符在 Tokenizer 中出现的顺序来确定的。在生成训练序列时，我们将每个古诗句子切分为固定长度的序列，并为每个序列设置一个目标，即下一个字符。例如，7 代表“春”，下一个目标值是 8，汉字的含义是“眠”。其中，春、眠、不、觉、晓这些字符经过 Tokenizer 编码后，可能对应的整数编码分别是 7、8、9、10、11，所以，这个序列的输入就是[7，8，9，10，11]，而目标值“，”对应的编码可能是 1，因此输入序列和目标值的具体数字编码取决于字符在 Tokenizer 中的编码顺序。每个字符的编码是唯一的，并且由 Tokenizer 在 fit_on_texts 时根据输入文本中字符的出现频率来分配。

```
Input sequence: [0 0 0 0 7], Target: 8
Input sequence: [0 0 0 7 8], Target: 9
Input sequence: [0 0 7 8 9], Target: 10
Input sequence: [ 0  7  8  9 10], Target: 11
Input sequence: [ 7  8  9 10 11], Target: 1
Input sequence: [ 8  9 10 11  1], Target: 3
Input sequence: [ 9 10 11  1  3], Target: 3
Input sequence: [10 11  1  3  3], Target: 12
Input sequence: [11  1  3  3 12], Target: 13
Input sequence: [ 1  3  3 12 13], Target: 14
Input sequence: [ 3  3 12 13 14], Target: 2
Input sequence: [ 0  0  0  0 15], Target: 16
Input sequence: [ 0  0  0 15 16], Target: 17
Input sequence: [ 0  0 15 16 17], Target: 18
Input sequence: [ 0 15 16 17 18], Target: 19
Input sequence: [15 16 17 18 19], Target: 1
Input sequence: [16 17 18 19  1], Target: 20
Input sequence: [17 18 19  1 20], Target: 21
Input sequence: [18 19  1 20 21], Target: 22
Input sequence: [19  1 20 21 22], Target: 23
Input sequence: [ 1 20 21 22 23], Target: 24
Input sequence: [20 21 22 23 24], Target: 2
Input sequence: [ 0  0  0  0 25], Target: 26
Input sequence: [ 0  0  0 25 26], Target: 4
Input sequence: [ 0  0 25 26  4], Target: 5
Input sequence: [ 0 25 26  4  5], Target: 27
Input sequence: [25 26  4  5 27], Target: 1
Input sequence: [26  4  5 27  1], Target: 28
Input sequence: [ 4  5 27  1 28], Target: 29
Input sequence: [ 5 27  1 28 29], Target: 30
Input sequence: [27  1 28 29 30], Target: 31
Input sequence: [ 1 28 29 30 31], Target: 32
Input sequence: [28 29 30 31 32], Target: 2
Input sequence: [ 0  0  0  0 33], Target: 6
Input sequence: [ 0  0  0 33  6], Target: 34
Input sequence: [ 0  0 33  6 34], Target: 4
Input sequence: [ 0 33  6 34  4], Target: 5
```

图 5-50　循环神经网络输入结果

注意： 在 Tokenizer 中，是否将逗号算作一个字符取决于 Tokenizer 的配置。如果在代码中使用了 char_level=True，则意味着 Tokenizer 将按字符级别进行标记化，包括所有标点符号。反之，标点符号则不记录在内。

目前，已经有了嵌入层，即已经将文本转换为向量。之后还需要进行捕捉序列中的时间依赖性，这时将接受嵌入层的输出，并处理整个序列的数据，提取出其中的特征。最终输出结果。

循环神经网络汉字预测程序的代码如下：

```
// 第 5 章 / 循环神经网络项目 .py
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Embedding, LSTM, Dense

# 示例古诗数据
poems = [
    "春眠不觉晓，处处闻啼鸟。",
    "夜来风雨声，花落知多少。",
    "床前明月光，疑是地上霜。",
    "举头望明月，低头思故乡。"
]

# 将文本转换为序列数据
tokenizer = Tokenizer(char_level=True)                    # 逐字级别
tokenizer.fit_on_texts(poems)
sequences = tokenizer.texts_to_sequences(poems)
word_index = tokenizer.word_index

# 生成输入序列和目标
maxlen = 5                                                # 序列长度
x = []
y = []

for seq in sequences:
    for i in range(1, len(seq)):
        input_seq = seq[:i]
        target = seq[i]
        X.append(input_seq)
        y.append(target)

# 填充序列
x = pad_sequences(X, maxlen=maxlen)
y = np.array(y)

# 打印数据
print("Tokenized sequences:")
for i, sequence in enumerate(x):
    target = y[i]
    print(f"Input sequence: {sequence}, Target: {target}")

# 构建模型
vocab_size = len(word_index) + 1                          # 词汇表大小加 1
embedding_dim = 50                                        # 嵌入维度

model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_
length=maxlen))
model.add(LSTM(units=128))                                # 隐藏层
model.add(Dense(units=vocab_size, activation='softmax'))  # 输出层
```

```
# 编译模型
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# 打印模型摘要
model.summary()

# 训练模型
model.fit(x, y, epochs=50, batch_size=32)

# 预测下一个字符
def predict_next_chars(model, tokenizer, text, num_chars=5, maxlen=5):
    result = []
    for _ in range(num_chars):
        # 将输入文本转换为序列
        sequences = tokenizer.texts_to_sequences([text])
        # 填充序列
        padded_seq = pad_sequences(sequences, maxlen=maxlen)
        # 预测下一个字符的概率分布
        pred = model.predict(padded_seq, verbose=0)
        # 获取概率最高的字符索引
        next_index = np.argmax(pred)
        # 获取对应字符
        for char, index in tokenizer.word_index.items():
            if index == next_index:
                result.append(char)
                text += char
                text = text[1:]                          # 保持长度为 maxlen
                break
    return ''.join(result)

# 示例预测
input_text = "举头望明月"
predicted_chars = predict_next_chars(model, tokenizer, input_text[-maxlen:],
num_chars=5)
print(f"Input text: {input_text}")
print(f"Predicted next characters: {predicted_chars}")
```

程序的训练效果如 5-51 所示，其中每个 Epoch 都代表一次训练过程，一共需要训练 50 次。可以观察到，一直到第 39 次，训练效果仍然不是特别理想，准确率仅为 34%，但是当训练过程达到第 42 次时，准确率已经超过 50%，在最终一次训练过程中，训练准确度达到了 57%。

在对模型进行训练之后，通过调用模型函数，可以输出文本，从而实现对文本的预测。例如一开始输入的文本为“举头望明月”，其实这句话本来就是训练集里面的，但是因为准确度并没有那么高，因此不能完全正确地进行预测。最终的预测结果为“，低低思是”可以看到，首先逗号精准预测，逗号之后是“低”，但是“低”之后的文字预测错误，

仍然出现了“低”，“头”被跳过，输出了“思”和“是”，基本结果符合模型对自己准确度的一个判断。当然如果想要更高的准确度，则需要更多的输入数据才可以实现。关于更多的诗词输入数据，读者可以到 GitHub 上进行下载。

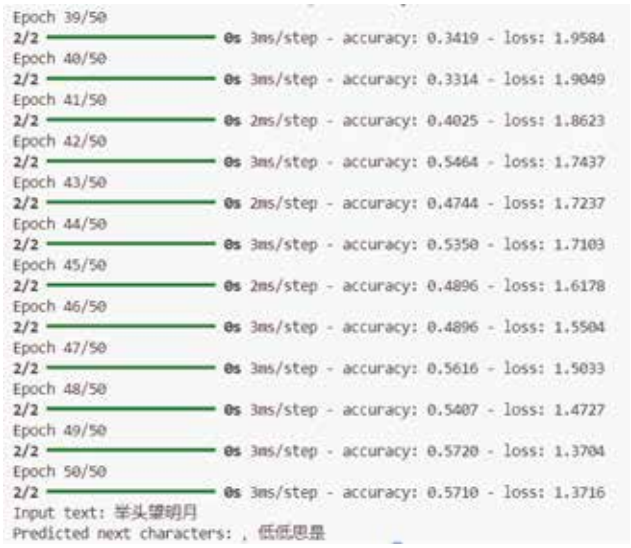


图 5-51　循环神经网络项目训练结果

3）长短时记忆模型

长短时记忆是一种特殊的循环神经网络，其提出主要是为了解决标准 RNN 在处理长序列时面临的梯度消失和梯度爆炸问题。长短时记忆模型通过引入门控机制，有效地控制信息的流动，使网络能够更好地捕捉和记忆长时间的依赖关系。

从上述介绍中可以得出，长短时记忆模型主要分为 4 个模块，分别为输入门（Input Gate）、遗忘门（Forget Gate）、候选记忆单元（Candidate Memory Cell）和输出门（Output Gate）。

注意： 输入门和输入数据是不同的，输入数据是模型接收外部的数据，而输入门则是长短时记忆模型中的一个组件，用于控制当前输入数据在多大程度上影响记忆单元。同理，输出数据和输出门也是不一样的，输出数据是整个长短时记忆模型的一个结果，而输出门则是一个内部的一个计算机制。

因此，整个长短时记忆模型的过程，应该是输入数据、输入门、遗忘门、候选记忆单元、输出门和输出数据。

假设我们有一个简单的字符预测任务，即给定一个字符序列，预测下一个字符。例如，我们有一个序列“hello”，希望训练一个模型来预测下一个字符。

首先需要将“hello”转换为一个字符索引。当然，使用什么方式对其进行转换并不重

要，重要的是，每个字符所代表的数值一定要唯一。例如，可以自行设置，“h”的索引为“0”，“e”的索引为“1”，“l”的索引为“2”，“o”的索引为“3”，那么整个字符索引列表就为［0，1，2，2，3］。除了自己对其进行规定以外，还可以使用 ASCII 值。ASCII 值中每个字符的索引都是唯一的。当然也可以自己编写一个由 26 英文字母组成的索引表进而获取“hello”字符的索引表。

在当前例子中，由于是示范数据，并且例子较为简单，因此使用索引列表［0，1，2，2，3］。读者以后在使用的过程中，可以根据自己数据的特点选择合适的索引列表进行操作。

在长短时记忆模型中，涉及的 4 个基本结构均需要进行计算。输入门的作用是决定当前输入数据对记忆单元更新的影响程度。具体来讲，输入门通过一个 Sigmoid 激活函数将当前输入数据和前一时间步的隐藏状态结合起来，生成一个 0~1 的权重（输入门的输出）。这个权重用于调节当前输入数据对记忆单元的更新影响。

初始状态下，$h_0=0$ 及 $C_0=0$，当时间步 $t=1$ 时，第 1 个字符的索引为“0”，输入数据 $x_1=0$，$\boldsymbol{i}_1=\sigma(\boldsymbol{W}_i*[h_0, x_1]+b_i)$，假设 $\boldsymbol{W}_i$ 和 b_i 的初始值分别是 0.5 和 0.1，计算 $\boldsymbol{i}_1$ 的值。其中，$\boldsymbol{W}_i$ 是输入门的权重矩阵，b_i 是输入门的偏置项，σ 是 Sigmoid 激活函数，输出 $\boldsymbol{i}_1$ 是一个 0~1 的向量。

当时间步 $t=1$ 时，输入门运行计算。代码如下：

```
// 第 5 章 / 输入门 .py
import numpy as np

#Sigmoid 激活函数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 初始化输入数据和前一时间步的隐藏状态
x_t = np.array([0])             # 第 1 个字符的索引为 0
h_t_minus_1 = np.array([0])     # 初始隐藏状态为 0

# 拼接隐藏状态和输入数据
combined_input = np.concatenate((h_t_minus_1, x_t))

# 初始化输入门的权重矩阵和偏置项
W_i = np.array([0.5, 0.5])      # 输入门权重矩阵为 0.5
b_i = 0.1                       # 输入门偏置项

# 计算输入门的输出值
i_t = sigmoid(np.dot(W_i, combined_input) + b_i)

# 打印计算结果
print("输入门的输出值 i_t:", i_t)
```

遗忘门虽然逻辑上在输入门之后，但是在长短时记忆模型中，遗忘门和输入门采用的却是两种独立的门控机制，各自独立且并行计算都依赖于相同的输入数据和前一时间步的

隐藏状态。

具体来讲，输入门和遗忘门的计算顺序是并行的，它们在同一时间步内同时计算。它们之间的关系可以通过它们共享的输入数据和隐藏状态来建立联系，但是输入门的输出不会直接影响遗忘门的计算结果。

遗忘门的公式为 $\boldsymbol{f}_t=\sigma(\boldsymbol{W}_f*[h_{t-1}, x_t]+b_f)$。其中，$\boldsymbol{f}_t$ 代表遗忘门的输出，是一个介于 0 和 1 之间的向量，表示每个记忆单元状态中各部分的保留或遗忘程度；$\boldsymbol{W}_f$ 代表遗忘门的权重矩阵，用来加权输入数据 x_t 和前一时间步的隐藏状态 h_{t-1}，以影响遗忘门的输出；b_f 是遗忘门的偏置项，用来调整遗忘门的输出偏移量，控制其输出的起始位置；σ 是 Sigmoid 激活函数，将线性组合的结果压缩到 0~1，表示每部分的权重。输入当前数据 $x_1=0$，假设 $\boldsymbol{W}_f$ 的初始值为 0.5，b_f 的初始值为 0.1，其余假设不变或保持输入门状态。

遗忘门计算过程的代码如下：

```
// 第 5 章 / 遗忘门 .py
import numpy as np

#Sigmoid 激活函数
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 初始化输入数据和前一时间步的隐藏状态
x_t = np.array([0])            # 第 1 个字符的索引为 0
h_t_minus_1 = np.array([0])    # 初始隐藏状态为 0

# 拼接隐藏状态和输入数据
combined_input = np.concatenate((h_t_minus_1, x_t))

# 初始化输入门和遗忘门的权重矩阵和偏置项
W_i = np.array([0.5, 0.5])     # 输入门权重矩阵为 0.5
W_f = np.array([0.5, 0.5])     # 遗忘门权重矩阵为 0.5
b_i = 0.1                      # 输入门偏置项
b_f = 0.1                      # 遗忘门偏置项

# 计算输入门和遗忘门的输出值
i_t = sigmoid(np.dot(W_i, combined_input) + b_i)
f_t = sigmoid(np.dot(W_f, combined_input) + b_f)

# 打印计算结果
print("输入门的输出值 i_t:", i_t)
print("遗忘门的输出值 f_t:", f_t)
```

候选记忆单元生成新的候选记忆，结合输入门的输出更新当前记忆单元。公式为 $\widetilde{C}_t=\tanh(\boldsymbol{W}_c*[h_{t-1}, x_t]+b_c)$。其中，$\widetilde{C}_t$ 为候选记忆单元；tanh 为激活函数；在 tanh 中 $\boldsymbol{W}_c$ 是候选记忆单元的权重矩阵，用来调整输入数据 x_t 和前一时间步的隐藏状态 h_{t-1} 在候选记忆单元中的影响力；b_c 是候选记忆单元的偏置项，用来调整候选记忆单元的输出偏移量，

控制其输出的起始位置。整个 $\boldsymbol{W}_c*[h_{t-1}, x_t]+b_c$ 的一个线性组合的结果被输入 Tanh 函数中，Tanh 函数将输入压缩到 −1~1，生成候选记忆单元的输出向量。在输入门和遗忘门中得出运算结果，输入门的输出值是 0.524 979 187 478 94，遗忘门的输出值是 0.524 979 187 478 94。其中我们预先规定，$\boldsymbol{W}_c$ 的初始值是 0.5，b_c 的初始值是 0.1。

候选记忆单元计算过程的代码如下：

```
// 第 5 章 / 候选记忆单元 .py
import numpy as np

# 定义双曲正切函数
def tanh(x):
    return np.tanh(x)

# 给定的输入门和遗忘门的输出值
i_t = 0.52497918747894
f_t = 0.52497918747894

# 初始化输入数据和前一时间步的隐藏状态
x_t = np.array([0])             # 假设第 1 个字符的索引为 0
h_t_minus_1 = np.array([0])     # 初始隐藏状态为 0

# 初始化候选记忆单元的权重矩阵和偏置项
W_c = np.array([0.5, 0.5])      # 候选记忆单元权重矩阵
b_c = 0.1                       # 候选记忆单元偏置项

# 拼接隐藏状态和输入数据
combined_input = np.concatenate((h_t_minus_1, x_t))

# 计算候选记忆单元的值
tilde_C_t = tanh(np.dot(W_c, combined_input) + b_c)

# 打印计算结果
print("候选记忆单元的值 tilde_C_t:", tilde_C_t)
```

输出门决定了当前记忆单元有多少信息需要输出到隐藏状态。输出门公式为 $\boldsymbol{o}_t=\sigma(\boldsymbol{W}_0*[h_{t-1}, x_t]+b_c)$。其中，$\boldsymbol{o}_t$ 是输出门的输出，是一个 0~1 的向量，表示每个记忆单元状态中各部分对当前时间步隐藏状态的贡献程度；$\boldsymbol{W}_0$ 是输出门的权重矩阵，用来调整输入数据 x_t 和前一时间步的隐藏状态 h_{t-1} 在输出门中的影响力；b_0 为输出门的偏置项，用来调整输出门的输出偏移量，控制其输出的起始位置；σ 为 Sigmoid 激活函数，将线性组合 $\boldsymbol{W}_0*[h_{t-1}, x_t]+b_0$ 的结果压缩到 0~1，表示每部分的权重。

结合给定参数 W_0 为 0.01，b_0 为 0.1，计算输出门结果。代码如下：

```
// 第 5 章 / 输出门 .py
import numpy as np

# 定义 Sigmoid 激活函数
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 给定的参数
W_o = np.array([0.01, 0.01, 0.01, 0.01])     # 输出门的权重向量
b_o = 0.1                                    # 输出门的偏置项

# 假设的输入数据和前一时间步的隐藏状态
x_t = np.array([0.4, 0.5])                   # 当前时间步的输入数据
h_t_minus_1 = np.array([0.3, 0.5])           # 前一时间步的隐藏状态

# 拼接隐藏状态和输入数据
combined_input = np.concatenate((h_t_minus_1, x_t))

# 计算输出门的加权和
output_gate_input = np.dot(W_o, combined_input) + b_o

# 计算输出门的输出
o_t = sigmoid(output_gate_input)

# 打印计算结果
print("输出门的输出 o_t:", o_t)
```

整体代码汇总，该项目的全部代码如下：

```
// 第 5 章 / 长短时记忆模型字符训练 .py
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# 自定义字符到索引的映射
char_to_index = {'h': 0, 'e': 1, 'l': 2, 'o': 3}
index_to_char = np.array(['h', 'e', 'l', 'o'])

text = "hello"

# 准备训练数据
sequence_length = 3
dataX = []
dataY = []
for i in range(0, len(text) - sequence_length, 1):
    seq_in = text[i:i + sequence_length]
    seq_out = text[i + sequence_length]
    dataX.append([char_to_index[char] for char in seq_in])
    dataY.append(char_to_index[seq_out])


n_patterns = len(dataX)
```

```
X = np.reshape(dataX, (n_patterns, sequence_length, 1))
y = np.array(dataY)

# 定义模型
model = Sequential()
model.add(LSTM(50, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(len(char_to_index), activation='softmax'))

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

model.fit(X, y, epochs=100, batch_size=1, verbose=2)

# 示例预测
test_seq = np.array([[char_to_index[c] for c in 'hel']])
test_seq = np.reshape(test_seq, (1, sequence_length, 1))
prediction = model.predict(test_seq, verbose=0)
predicted_char = index_to_char[np.argmax(prediction)]
print("Predicted next character:", predicted_char)
```

在上述代码中，并没有对 LSTM 的全过程进行展示，相反，相比于对长短时记忆模型的 4 部分介绍，全部的代码显得短了很多。主要是由于调用了第三方库 TensorFlow。TensorFlow 对长短时记忆模型的代码做了一个封装，使用者只需关注于功能实现，对于其中原理部分并不需要有太多的关注。这降低了使用者在使用深度学习模型时的难度。

除了可以预测字符以外，长短时记忆模型还可以应用在数据分析方面，进行时间序列预测。门控机制允许 LSTM 在长序列中选择性地记忆和遗忘信息，从而能够更好地捕捉和利用时间序列中的长期依赖关系。同时，LSTM 通过细胞状态（Cell State）来保持信息，这种长期记忆的机制使网络能够记住和利用之前时间步的信息，而不会丢失关键的历史信息。与 RNN 相比，LSTM 的设计使它更容易并行化处理，这在训练过程中可以显著地提高效率。

为了很好地对股票价格进行模拟，我们随机生成示例数据，但是使用正弦函数加上一些随机噪声生成时间序列数据，为数据加入干扰，以加强数据的真实性。之后，进行数据预处理操作，通过 MinMaxScaler 将数据缩放到 0~1，并将数据序列化成 LSTM 模型可以接受的格式。之后，创建 LSTM 模型，定义一个包含 LSTM 层和全连接层的模型，并编译模型以准备训练。训练过程共包含 50 个训练周期，批量大小为 32。LSTM 模型创建，同样使用第三方库 TensorFlow 进行快捷创建。使用生成的数据来训练 LSTM 模型。最后使用训练好的模型预测未来的值，并将预测结果与原始数据进行对比和可视化。当然还包括一个绘制原始数据和预测数据的对比图，以展示 LSTM 模型的预测效果，如图 5-52 所示。

以上展示了如何使用 LSTM 模型来进行时间序列预测，虽然简单，但却能够帮助理解 LSTM 在处理时间序列数据时的基本应用方法。

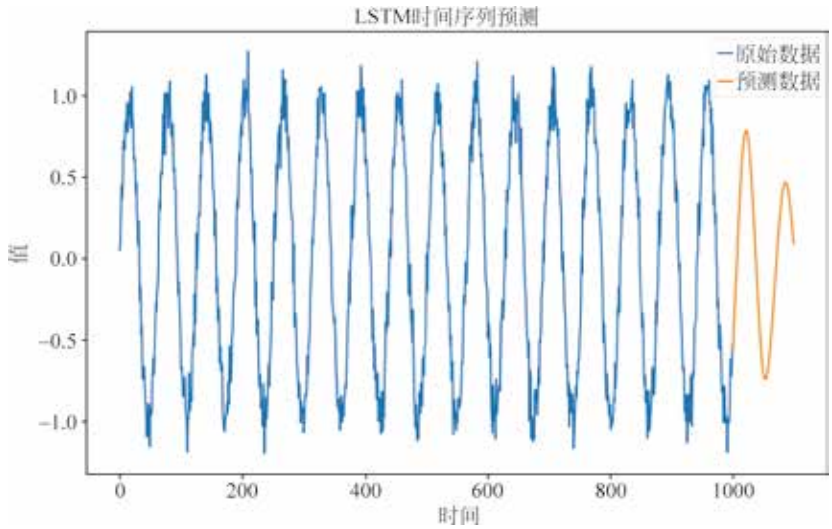


图 5-52　原始数据和预测数据对比图

LSTM 时间预测模型的代码如下：

```
// 第 5 章 / 候选记忆单元 .py
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler

# 生成一些示例数据（模拟股票价格）
def generate_data(n):
    time = np.arange(0, n, 1)
    data = np.sin(time * 0.1) + np.random.normal(0, 0.1, size=n)
    return data.reshape(-1, 1)

# 数据预处理
def preprocess_data(data, look_back):
    scaler = MinMaxScaler(feature_range=(0, 1))
    data_normalized = scaler.fit_transform(data)
    X, y = [], []
    for i in range(len(data_normalized) - look_back):
        X.append(data_normalized[i:(i + look_back), 0])
        y.append(data_normalized[i + look_back, 0])
    return np.array(X), np.array(y), scaler

# 定义模型
def create_lstm_model(look_back):
    model = Sequential()
    model.add(LSTM(units=50, input_shape=(look_back, 1)))
    model.add(Dense(units=1))
```

```
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# 生成示例数据
np.random.seed(42)
data = generate_data(1000)

# 数据预处理及分割
look_back = 10
X, y, scaler = preprocess_data(data, look_back)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))

# 创建并训练模型
model = create_lstm_model(look_back)
model.fit(X, y, epochs=50, batch_size=32, verbose=1)

# 预测未来值
predicted_values = []
last_sequence = X[-1].flatten().tolist()

for i in range(100):
    last_sequence_np = np.array(last_sequence[-look_back:], dtype=np.float32)
                                                    # 只取最后的 look_back 个数据作为输入
    last_sequence_reshaped = last_sequence_np.reshape(1, look_back, 1)
    next_predicted_value = model.predict(last_sequence_reshaped, verbose=0)
    predicted_values.append(next_predicted_value[0, 0])
    last_sequence.append(next_predicted_value[0, 0])

# 反向转换预测值
predicted_values = scaler.inverse_transform(np.array(predicted_values).reshape
(-1, 1))

# 绘制预测结果
plt.figure(figsize=(10, 6))
plt.plot(np.arange(len(data)), data, label='Original Data')
plt.plot(np.arange(len(data), len(data) + len(predicted_values)), predicted_
values, label='Predicted Data')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('LSTM Time Series Prediction')
plt.legend()
plt.show()
```

4）Transformer 模型

注意力机制是 Transformer 模型的核心思想。传统的循环神经网络和长短期记忆模型在处理长序列时容易遗忘早期的信息，而对早期信息的遗忘，对于 RNN 来讲，是由于梯度爆炸和梯度消失而导致的。针对这一点，LSTM 模型的提出也是为了解决 RNN 的梯度爆炸和梯度消失问题。LSTM 通过引入门机制（门控单元）来缓解 RNN 中的梯度消失问

题，从而更好地捕捉长距离依赖，然而，LSTM 在处理特别长的序列时仍然面临一些挑战：其一，便是复杂的门控制，在非常长的序列中可能会变得不稳定，仍然会有信息遗忘的风险；其二，LSTM 的复杂结构使每个时间步的计算量较大。在处理非常长的序列时，计算成本和训练时间显著增加，因此整体来讲，LSTM 等模型的训练效率要低于 Transformer 模型等新型模型。

举个例子来讲，RNN 就好像是一个传话游戏，一句话由 A 传递到 B 再传递到 C，中间的意思已经发生了很大的改变，并且存在很多歧义。LSTM 就好比是一个备忘录，它能够完好地记录下当时的所思所想，但是当过了很长的时间后再去看时，已经不能再体会到当时的感觉了，只记得当时有这么一件事。而 Transformer 模型好比是一个实时翻译的全球会议，在会上，每个人都可以畅所欲言，但是注意，每个人也都需要通过翻译才能去理解他人所讲的话的意思，因此即使是 Transformer 模型，它在转述信息时，依然是有误差的，但是相比于 RNN 和 LSTM 来讲，这个误差已经小了很多，毕竟已经在间接面对面交流了。

传统的模型多采用顺序处理机制，而 Transformer 模型则采用自注意力机制（Self-Attention）来捕捉序列中任意位置之间的依赖关系。这使 Transformer 模型可以在训练过程中实现完全并行化，大大地提高了训练效率。自注意力机制通过计算输入序列中每个元素与其他元素的相关性来决定其重要性。除了单一的注意力机制，还可以使用多头注意力机制（Multi-Head Attention）通过并行化多个注意力头来捕捉输入序列中的不同子空间信息。

在模型设计方面，Transformer 模型主要包含 4 个层面，分别是嵌入层、位置编码、编码层和解码层。嵌入层负责将输入的 token（通常是词汇表中的单词或字符）映射为高维度的向量表示。在 Transformer 模型中，每个 token 被映射为一个固定大小的向量，这些向量既可以在训练过程中通过学习得到，也可以使用预训练的词嵌入向量。

假设现在有一个词汇表，词汇表大小为 10 000，意味着有 10 000 个不同的单词或标记。将每个单词映射为长度为 512 的向量意味着为词汇表中的每个单词或 token 分配一个固定长度的向量表示。这个长度是自定义的，只要合适就好。使用第三方库 torch 定义嵌入层，其中输入中的数字代表索引，输出的结果有 3 个变量，分别是批量大小、序列长度、嵌入维度。

嵌入层构建过程的代码如下：

```
// 第 5 章 / 嵌入层 .py
import torch
import torch.nn as nn

# 假设词汇表大小为 10 000，将每个单词映射为长度为 512 的向量
vocab_size = 10000
embedding_dim = 512

# 定义嵌入层
embedding_layer = nn.Embedding(vocab_size, embedding_dim)
```

```
# 示例输入
input_tokens = torch.LongTensor([[1, 2, 3, 4, 5]])

# 将输入 token 映射为向量表示
embedded_tokens = embedding_layer(input_tokens)

print("Embedded tokens shape:", embedded_tokens.shape)    # 输出
torch.Size([1, 5, 512])
```

这样进行表述，可能还不是特别清楚，现在将用更生动的表述再次进行阐述。假设，我们现在有一个单词表，这个单词表是非常简单的，只包含 3 个单词，分别是猫、狗、鸟。定义一个嵌入层，将每个单词的嵌入维度指定为 2。希望用一个嵌入层将这些单词映射到一个二维向量空间中，每个单词对应一个二维向量。二维向量需要两个值，分别是 x 值和 y 值。在原点右边，x 值为正，在原点上边，y 值为正。初始化时向量是随机的，也是由机器自行进行识别的，例如“猫”的向量为［0.1，0.3］，“狗”的向量为［0.5，0.7］，“鸟”的向量为［0.2，0.4］，如图 5-53 所示。

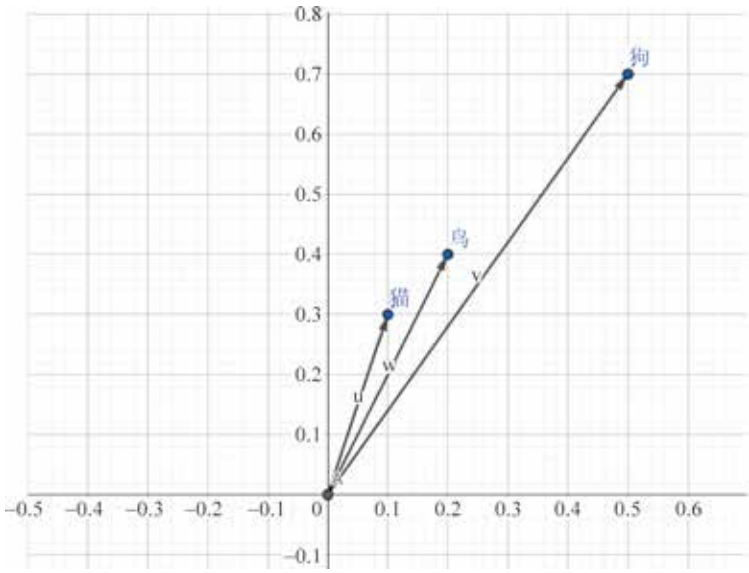


图 5-53　向量坐标表示

向量坐标表示方法的代码如下：

```
// 第 5 章 / 示例坐标 .py
import matplotlib.pyplot as plt

# 定义初始嵌入向量
vectors = {
    'cat': [0.1, 0.3],
```

```
    'dog': [0.5, 0.7],
    'bird': [0.2, 0.4]
}

# 创建绘图
plt.figure(figsize=(8, 6))

# 绘制每个单词的向量
for word, vector in vectors.items():
    plt.scatter(vector[0], vector[1], label=word)

# 添加标签
for word, vector in vectors.items():
    plt.annotate(word, (vector[0], vector[1]))

# 设置图形标题和轴标签
plt.title("A two-dimensional representation of an embedding vector")
plt.xlabel("Dimension1")
plt.ylabel("Dimension2")

# 显示图例
plt.legend()

# 显示图形
plt.show()
```

“猫”的向量为［0.1，0.3］，“狗”的向量为［0.5，0.7］，“鸟”的向量为［0.2，0.4］，这些向量是示例中的初始化嵌入向量。通常这些初始值是通过随机初始化生成的，这里采用的方式是自定义。在实际训练过程中，嵌入向量的初始值可以通过多种方式生成，例如均匀分布或正态分布。这些初始化的值本身没有特殊的含义，它们只是模型训练的起点。

在实际应用中，最终的嵌入向量是通过模型训练计算得出的，需要考虑到词汇表大小、嵌入维度等。最终得到词嵌入后的坐标，“猫”的坐标为［−0.383 629 2，−0.080 007 08］，“狗”的坐标为［0.164 916 75，−0.522 192 36］，鸟的坐标为［0.549 380 8，0.839 382 2］，如图 5-54 所示。

在位置编码部分，Transformer 模型没有像循环神经网络或卷积神经网络那样的显式顺序信息，因此需要一种方法来表达输入序列中 token 的顺序。位置编码的作用就是为输入序列中的每个 token 提供一个位置信息。位置编码需要两个数值来表示其唯一编码，假设示例长度为 50，嵌入维度为 512，位置编码如图 5-55 所示。

位置编码正弦函数部分：

$$\mathrm{PE}_{(\mathrm{pos},\,2i)}=\sin\left(\frac{\mathrm{pos}}{10\,000^{\frac{2i}{d_{\mathrm{model}}}}}\right) \tag{5-20}$$

其中，d_{model} 表示嵌入维度大小；pos 表示位置；i 表示索引；sin 表示取正弦值。

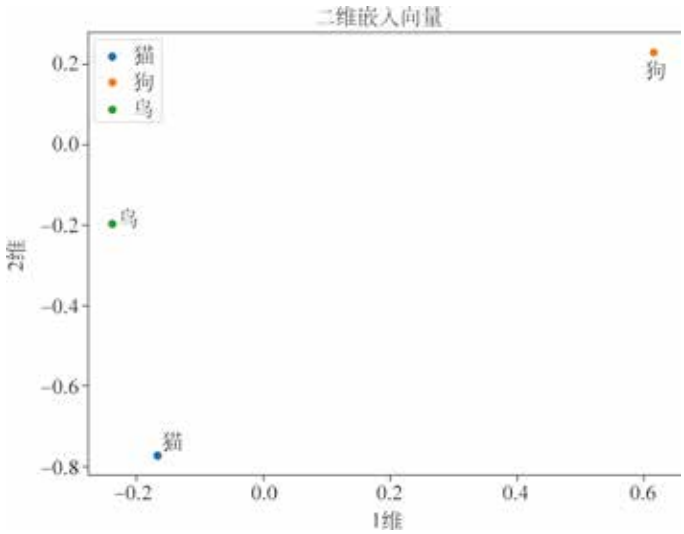


图 5-54　最终词嵌入二维向量坐标

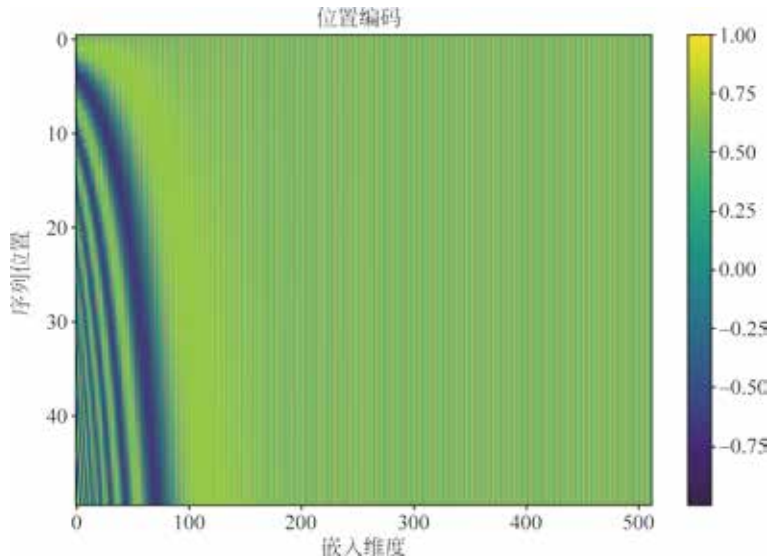


图 5-55　位置编码图示

位置编码余弦函数部分：

$$\mathrm{PE}_{(\mathrm{pos},\,2i+1)}=\cos\left(\frac{\mathrm{pos}}{10\,000^{\frac{2i}{d_{\mathrm{model}}}}}\right) \tag{5-21}$$

其中，d_{model} 表示嵌入维度大小；pos 表示位置；i 表示索引；cos 表示取余弦值。

在 Transformer 模型中，位置编码被添加到嵌入向量中，以便为模型提供每个词的位置信息：输入向量 = 嵌入向量 + 位置编码。这种方法确保了模型在处理输入序列时能够识

别每个元素的相对位置。

位置编码实现过程的代码如下：

```
// 第 5 章 / 位置编码实现 .py
import numpy as np
import matplotlib.pyplot as plt

def get_position_encoding(seq_len, d_model):
    # 创建一个矩阵，其中 seq_len 是序列长度，d_model 是嵌入维度
    position_encoding = np.zeros((seq_len, d_model))

    # 计算每个位置的编码值
    for pos in range(seq_len):
        for i in range(0, d_model, 2):
            position_encoding[pos, i] = np.sin(pos / (10000 ** ((2 * i) /
d_model)))
            if i + 1 < d_model:
                position_encoding[pos, i + 1] = np.cos(pos / (10000 **
((2 * i) / d_model)))

    return position_encoding

# 示例参数
seq_len = 50      # 序列长度
d_model = 512     # 嵌入维度

# 获取位置编码
position_encoding = get_position_encoding(seq_len, d_model)

# 可视化位置编码
plt.figure(figsize=(10, 8))
plt.imshow(position_encoding, aspect='auto', cmap='viridis')
plt.colorbar()
plt.title("Location coding")
plt.xlabel("Embedding dimensions")
plt.ylabel("Sequence position")
plt.show()
```

除了嵌入层和位置编码外，Transformer 模型中的另外一组比较重要的概念便是编码器和解码器，它们分别承担着输入信息的编码任务和输出信息的解码任务。

Transformer 模型的编码器 - 解码器结构源自序列到序列（Seq2Seq）任务，例如机器翻译。在这种任务中，输入是一个序列（例如英语句子），输出是另一个序列（例如法语句子）。编码器和解码器分别处理输入序列和输出序列，确保模型能够有效地将输入序列转换为目标序列。编码器由多个编码器层组成，每个编码器层包含自注意力机制、前馈神经网络和残差连接。解码器接收编码器输出（通常是编码器的最终隐藏状态）作为输入，生成目标序列的输出。它也由多个解码器层组成，包括解码器 - 自注意力、编码器 - 解码

器注意力和前馈神经网络。

编码器和解码器通过注意力机制相互作用。编码器首先将输入序列转换为一组上下文感知的表示，然后解码器使用这些表示生成目标序列。在生成目标序列的过程中，解码器会不断地参考编码器生成的表示，以确保输出序列与输入序列的语义一致。

其他模型（例如简单的前馈神经网络、卷积神经网络等）不需要编码层和解码层，通常是因为它们处理的任务不同于序列到序列（Seq2Seq）任务，或者它们通过其他结构实现了类似的功能。

前馈神经网络既不需要处理序列数据，也不需要将一个序列映射到另一个序列，因此不需要编码器和解码器结构，它直接将输入映射到输出。

CNN 使用卷积层和池化层提取空间特征，它关注的是局部特征和层级特征，而不是序列的顺序和依赖关系。由于图像处理不涉及序列到序列映射，因此不需要编码器和解码器结构。

虽然 RNN 处理的是序列数据，但它通过其内部的循环结构自然地处理序列中的依赖关系。RNN 在每个时间步都会接收上一个时间步的隐状态，并结合当前输入生成新的隐状态。这种机制使 RNN 能够处理序列数据，而不需要显式的编码器和解码器结构。

LSTM 是 RNN 的改进版本，通过引入门机制来更好地捕捉长期依赖关系。这些模型同样通过其循环结构处理序列中的依赖关系，不需要显式的编码器和解码器结构。

编码器实现过程的代码如下：

```
// 第 5 章 / 编码器实现 .py
class EncoderLayer(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward=2048, DropOut=0.1):
        super(EncoderLayer, self).__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead, DropOut=DropOut)
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.DropOut = nn.DropOut(DropOut)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, src, src_mask=None):
        src2, _ = self.self_attn(src, src, src, attn_mask=src_mask)
        src = src + self.DropOut(src2)
        src = self.norm1(src)
        src2 = self.linear2(self.DropOut(F.relu(self.linear1(src))))
        src = src + self.DropOut(src2)
        src = self.norm2(src)
        return src

class Encoder(nn.Module):
    def __init__(self, encoder_layer, num_layers):
        super(Encoder, self).__init__()
```

```
        self.layers = nn.ModuleList([encoder_layer for _ in range(num_layers)])

    def forward(self, src, src_mask=None):
        for layer in self.layers:
            src = layer(src, src_mask)
        return src
```

解码器实现过程的代码如下：

```
// 第 5 章 / 解码器实现 .py
class DecoderLayer(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward=2048, DropOut=0.1):
        super(DecoderLayer, self).__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead, DropOut=DropOut)
        self.multihead_attn = nn.MultiheadAttention(d_model, nhead, DropOut=
DropOut)
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.DropOut = nn.DropOut(DropOut)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)

    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None):
        tgt2, _ = self.self_attn(tgt, tgt, tgt, attn_mask=tgt_mask)
        tgt = tgt + self.DropOut(tgt2)
        tgt = self.norm1(tgt)
        tgt2, _ = self.multihead_attn(tgt, memory, memory, attn_mask=
memory_mask)
        tgt = tgt + self.DropOut(tgt2)
        tgt = self.norm2(tgt)
        tgt2 = self.linear2(self.DropOut(F.relu(self.linear1(tgt))))
        tgt = tgt + self.DropOut(tgt2)
        tgt = self.norm3(tgt)
        return tgt

class Decoder(nn.Module):
    def __init__(self, decoder_layer, num_layers):
        super(Decoder, self).__init__()
        self.layers = nn.ModuleList([decoder_layer for _ in range(num_layers)])

    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None):
        for layer in self.layers:
            tgt = layer(tgt, memory, tgt_mask, memory_mask)
        return tgt
```

5）GPT 模型

GPT（Generative Pre-trained Transformer）模型是一种基于 Transformer 架构的语言模

型，由 OpenAI 开发。可以说 GPT 模型和 Transformer 模型非常相近，但是却指向了两个不同的内容。

Transformer 是一种通用的神经网络架构，适用于各种序列到序列的任务。GPT 是一种特定的语言模型，实现了 Transformer 的解码器部分，去掉了 Transformer 的编码器部分，专注于文本生成任务。Transformer 包含编码器和解码器，适用于各种输入 / 输出任务，而由于 GPT 只包含解码器，用于从前面的上下文生成后续文本，所以更加强调生成自然语言文本的能力。

由此可以得出 Transformer 的应用相对广泛，包括但不限于语言处理、图像处理、时间序列预测等。GPT 专注于语言模型，不断扩展参数和能力以提升文本生成的质量和多样性。

GPT 模型最为杰出的代表就是 ChatGPT，而在国内人们使用频率较高的是文心一言，因为它已经可以在手机端使用了，遇到问题简单查一下，就可以很快地获取一个答案。

ChatGPT 是在海量的文本数据上进行训练的，这些数据包含了互联网上各种类型的内容，例如书籍、文章、代码片段、对话等。通过在这些数据上进行训练，模型学会了大量的语言模式和知识。在大规模文本数据集上进行预训练，这些数据覆盖了广泛的话题和语言风格，使模型能够学习到丰富的语言知识和语境理解能力。

目前，如果想要实现一个自己本领域的大语言模型，则很少需要自己再去从头开始创建，至少有两种比较快捷的途径，一种是利用开源的大语言模型，例如百川、LLaMA、鹏程・盘古等，借助开源的模型，训练自己的数据会比自己开发一个模型方便很多。另外，还有一种比较便捷的开发方式便是使用一些成型大语言的模型的 API，例如 ChatGPT、LLaMA3-70B 等。

这里调用接口的方法以 ChatGPT 进行举例。使用 ChatGPT 的接口实现大语言模型对话，首先需要导入 Python 中的第三方库 openai，之后需要设置密钥，密钥的获取需要向 ChatGPT 申请，申请之后会获取一定的免费额度，免费额度用完之后如果想要继续使用，则需要花钱购买了，然后就可以定义希望模型生成文本的提示语，这个提示语在测试时，可以一开始就在代码中进行标注，但是如果想要投入到开发中进行使用，就需要使用 input() 方法了，方便用户进行输入。指定使用的 GPT-3 模型版本，这里使用 text-davinci-003，这是 GPT-3 的一个高性能版本。promopt 是提示语的意思，也可以理解为向大语言模型下达一个指令。指令的好坏在一定程度上会影响整个生成结果的好坏。市面上有好多教如何写好一个 prompt 的课程。temperature 表示控制生成文本的创意程度。较高的值（例如 0.7）会使输出更有创意，但也更不稳定。较低的值（例如 0.3）会使输出更一致，但也更保守。*n* 表示指定生成几个响应。stop 值表示指定停止生成，这里没有显式设置。

调用 ChatGPT 的 API 的方法，其代码如下：

```
// 第 5 章 / 模型接口 .py
import openai

# 设置 API 密钥
```

```
openai.api_key = 'your-openai-api-key'

# 定义提示语
prompt = "写一段关于气候变化的简短文章。"

# 调用 GPT-3 模型生成文本
response = openai.Completion.create(
    engine="text-davinci-003",
    prompt=prompt,
    max_tokens=150,      # 控制生成文本的长度
    temperature=0.7,     # 控制生成文本的创意程度
    n=1,                 # 生成一个响应
    stop=None            # 没有显式停止标记
)

# 打印生成的文本
print(response.choices[0].text.strip())
```

如果仅仅是为了实现一个聊天功能，则好像并不能满足我们的需求，因为同类产品已经很多了。ChatGPT-3 和百度的文心一言回复的准确度其实没有多少差别，那么我们为什么还要花钱实现一个并不太好的功能呢?

很多时候，我们之所以调用 API，实现对话功能仅仅是项目的一部分，而非全部。比方说，假设我们要实现一个历史博物馆领域的大语言模型，这在之前并不存在。实现对话仅仅是里面最基础的功能，我们还要考虑很多内容，例如还需要规定，大语言模型的输出必须在规定的数据库之内，以及历史数据库又该选用何种方式，是关系数据库，还是非关系数据库。如果选用非关系数据库，比方说选用图数据库 Neo4j，则还需要考虑，中心点在哪里，数据与数据之间是单向联系还是双向联系。

为了实现这个 ChatGPT 对话功能与自己数据库的一个链接，笔者在之前的项目中采用的是利用 Neo4j 作为图数据库，之后利用 LangChain 作为 ChatGPT 和数据库之间的一个连接，通过第三方库 langchain2Neo4j 实现自然语言和 Neo4j 知识图谱之间的交互。

LangChain 是一个由大语言模型驱动的框架，旨在为快速应用开发提供便利。该框架主要由六大关键模块组成，包括 Agents、Chains、Indexes、Memory、Models 和 Prompts。这个框架可以帮助我们实现两大功能：将语言模型链接到其他数据源；允许语言模型与其环境进行交互。通过 LangChain 框架，研究者可以调用大语言模型的 API 来回答自身数据集的问题。

LangChain 技术六大模块介绍。第一，模型（Models）：在语言模型方面，LangChain 共分为两大模型——大语言模型（Large Language Models，LLM）、聊天模型（Chat Models）。对于各模型，在应用方面的操作相似，均为输入文本、调用模型、输出结果。不同点在于大语言模型是将文本作为一种输入对象；聊天模型是将一个聊天信息的列表作为一个输入对象。第二，提示工程（Prompts）：提示是文本的输入，是由多个组件所构成的，共分为 4 部分——提示内容（PromptValue）、提示模板（Prompt Templates）、示例选择器（Prompt

Templates）、输出解析器（Output Parsers）。提示内容用于生成提示模板，这些方法可以将模型转换为所需要的输入内容；提示模板负责创建提示内容，起到协助作用，最终传递出语言模型的内容；示例选择器通常用于在提示符中包含提示示例；输出解析器对输出的语言文本进行结构化处理。第三，记忆（Memory）：这是 LangChain 提供的一种对文本进行长期记忆，进而实现多轮对话的一种机制。分为两种主要方法：一是基于输入，匹配相关的文本片段；二是基于输入和输出，不断更新状态。第四，索引（Indexes）：索引是构造文档的一种很好的方法，便于 LLM 直接与文档进行交互。第五，链（Chains）：单独的 LLM 对于一些简单文件是够用的，但是许多复杂的应用文件需要链接 LLM，这时就需要用到链技术。第六,一些应用程序（Agents），可能不止用到目前已有的应用模型，还可能依赖于用户输入的未知的链，这时就需要使用代理工具。综上，对 LangChain 技术进行了简要介绍，大致了解了此技术的构成，以及各部分的功能和应用。

Neo4j 被广泛认可为一种先进的图数据库管理系统，其独特的定位在于专注于图数据的存储和处理。图数据库通过节点（表示实体）和关系（表示实体之间的联系）的方式，将数据以图形的形式呈现，有力地强调了数据实体之间的关系网络。Neo4j 以其突出的特点，为复杂关系数据的建模、查询与分析提供了卓越的解决方案。Neo4j 的核心是其基于图数据模型的数据存储方式。节点代表了实际世界中的实体，而关系则体现了实体之间的各种关联。这种数据模型在不同领域的应用中具有广泛的适用性，例如，在本文所要创建的系统中，节点代表文物数据，关系可以表示为文物数据的所属关系。Neo4j 的数据模型使复杂的关系网络能够以直观且高效的方式被呈现和查询。强大的 Cypher 查询语言是 Neo4j 的另一大亮点。Cypher 提供了一种声明性的查询语法，以可读性强的方式表达了对图数据的查询需求。通过 Cypher，用户可以轻松地提取节点、关系和路径，以及执行数据的增、删、改操作。特别地，Cypher 允许用户灵活地执行关系模式匹配，从而发现潜在的数据关联。除此之外，Neo4j 的 API 集成功能为开发人员提供了更多的灵活性。通过 Neo4j 的 API，开发人员可以将 Cypher 查询嵌入应用程序中，实现对图数据的操作和查询。这种无缝的集成能力使应用程序能够充分地利用图数据库的优势，实现更深入的数据分析和洞察。

整个系统的构建主要包含 4 个层面：首先是用户进行输入，输入之后的自然语言将被转换为 Neo4j 的查询语言（Cypher），其次查询语言（Cypher）会对图数据库（Neo4j）的数据进行访问查询，在查询之后会得到一个返回结果，返回结果是数据库语言，最后需要再将其转换为查询语言，方可被用户接收到。

上述逻辑的核心代码其实很短，大多是对现有功能的复用，系统的核心代码如下：

```
// 第 5 章 / 核心代码 .py
import os
import openai
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.chains import GraphCypherQAChain
from langchain.graphs import Neo4jGraph
```

```
from langchain.chains.question_answering import load_qa_chain

os.environ[
    "OPENAI_API_KEY"] = "input your key"

openai.api_key = os.environ['OPENAI_API_KEY']

graph = Neo4jGraph(
    url="input your url",
    username="username",
    password="password")


chain = GraphCypherQAChain.from_llm(
    ChatOpenAI(temperature=0),
    graph=graph, verbose=True,)

print(chain.run("""给我说一下骨猴外观吧"""))
```

用户第 1 次向系统提问：“给我说一下功能是装饰品的文物有哪些？”。用户提出请求，需要得到在数据库当中用途是装饰品的文物。之后该自然语言经过转化，变为了 Cypher 查询语言。首先匹配到文物（Artifact），文物隶属（BELONGS_TO）于博物馆（Museum）之中，在博物馆之中，寻找用途（purpose）是装饰品的文物，并返回它的名字（name），最终得到一个以列表形式返回的结果，共有两个结果，分别是“白陶斗笠形器”和“骨猴”。它们以列表内嵌套字典的形式表达结果，最终将它们组织成文“白陶斗笠形器和骨猴都是功能是装饰品的文物。”形成自然语言答案的代码如下：

```
// 第 5 章 / 查询 .cypher
Generated Cypher:
MATCH (a:Artifact)-[:BELONGS_TO]->(m:Museum)
WHERE a.purpose = '装饰品'
RETURN a.name
Full Context:
[{'a.name': '白陶斗笠形器'}, {'a.name': '骨猴'}]

> Finished chain.
白陶斗笠形器和骨猴都是功能是装饰品的文物。
```

为了提高回复语言的可读性及流畅性。在当时的项目中，笔者还对 GraphCypherQAChain 类进行了详细拆解，该类包含 5 个参数：cls、llm、qa_prompt、cypher_prompt 及 **kwargs。其中，qa_prompt 是一个 BasePromptTemplate 类型的参数，用于指定问题回答任务的提示（prompt）。BasePromptTemplate 可能是一些预定义的模板，用于构建问题回答的输入。进一步拆解 BasePromptTemplate 中的函数 CYPHER_QA_PROMPT，发现它由两个参数组成：input_variables 和 template。模板字符串 CYPHER_QA_TEMPLATE 包含信息和问题两部分，其中问题部分会根据输入内容进行填充，而信息部分则用于约定和说明问题回答的背景信

息。通过微调信息部分，我们成功地使语言更贴近博物馆应用场景，从而实现了更自然、更富有形象的回答。微调后的回答不仅增加了修饰性的形容词，还运用了诙谐的语言和比喻等修辞手法，但同时保持了文物描述的准确性。这一调整使人工智能回答更符合正常语言表达，从而提升了用户体验，达到了本研究的预期效果。这个版本更加简洁，着重突出了关键信息，提高了可读性。

图 5-56　项目 UI 图

当然，由于源代码不能提供给用户使用，于是制作了一个简单的 UI 界面，如图 5-56 所示。软件主要分为 3 部分，上方是 Neo4j 设置，需要在其中填入 Neo4j 的 API，这样便可以调用图数据库。在 Model 设置中目前仅包含 ChatGPT，需要在其中填入 ChatGPT 的 API，由此便可实现大语言模型的对话功能。当一名用户在提问框输入问题之后，便可以单击“提交”按钮，在回答框中获取答案。单击“清除”按钮，便可删除之前的对话，进而开始下一次对话流程。当然还包含一个“联系我们”和“关于此系统”按钮。“关于我们”中是笔者的联系方式，“关于此系统”是当前版本介绍，和未来开发版本的介绍。

项目实现过程的代码如下：

```
// 第 5 章 / 项目框架 .py
import os
import sys
import openai
from PySide6.QtCore import QPropertyAnimation, QEasingCurve
from PySide6.QtGui import QPixmap, Qt, QGuiApplication,QFont, QIcon
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.chains import GraphCypherQAChain
from langchain.graphs import Neo4jGraph
from langchain.chains.question_answering import load_qa_chain
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton,
QVBoxLayout, QWidget, QLabel, QLineEdit, \
    QTextEdit, QMessageBox, QDialog, QHBoxLayout
import time


class MyMainWindow(QMainWindow):
```

```
        def __init__(self):
            super().__init__()

            # 应用样式表
            self.setStyleSheet("""
                QMainWindow {
                    background-color: #f5f5f5;
                }
                QPushButton {
                    background-color: #3498db;
                    color: white;
                    border: none;
                    padding: 8px 16px;
                    font-size: 12px;                # 调整字体大小
                    font-family: "Microsoft YaHei";  # 使用微软雅黑字体，确保字体存在
                }
                QPushButton:hover {
                    background-color: #2980b9;
                }
                QLabel {
                    font-size: 14px;
                    font-family: Arial;
                }
                QLineEdit, QTextEdit {
                    font-size: 14px;
                    font-family: Arial;
                }
                #input_textbox, #output_textbox {
                    border: 1px solid #ccc;
                    border-radius: 5px;
                    padding: 8px;
                }
                #contact_us_btn {
                    background-color: #e74c3c;
                }
            """)

            # 设置窗口的大小和位置
            self.setGeometry(100, 100, 400, 500)

            # 设置窗口的标题
            self.setWindowTitle(" 史谈文博系统 v1.0")

            # 设置窗口图标
            icon = QIcon(" 史谈文博 icon.png")        # 用实际的图标图像路径替换
            "path_to_your_icon_image.png"
            self.setWindowIcon(icon)
```

```
        # 创建按钮
        self.neo4j_btn = QPushButton("Neo4j 设置 ")
        self.model_btn = QPushButton("Model 设置 ")

        # 设置 Neo4j 选项框
        self.neo4j_url_label = QLabel("URL:")
        self.neo4j_url_input = QLineEdit()
        self.neo4j_username_label = QLabel("Username:")
        self.neo4j_username_input = QLineEdit()
        self.neo4j_password_label = QLabel("Password:")
        self.neo4j_password_input = QLineEdit()

        # 设置模型选项框
        self.api_label = QLabel("API:")
        self.api_input = QLineEdit()

        # 创建输出和输出按钮
        self.input_label = QLabel(" 向我提出你的问题 :")
        self.input_textbox = QTextEdit()
        self.output_label = QLabel(" 来看一看我的回答 :")
        self.output_textbox = QTextEdit()

        # 创建保存 / 清除 Neo4j 的设置
        self.neo4j_save_btn = QPushButton(" 保存 ")
        self.neo4j_clear_btn = QPushButton(" 清除 ")

        # 创建保存 / 清除 Neo4j 的设置
        self.model_save_btn = QPushButton(" 保存 ")
        self.model_clear_btn = QPushButton(" 清除 ")

        # 创建函数连接按钮
        self.neo4j_btn.clicked.connect(self.show_neo4j_settings)
        self.model_btn.clicked.connect(self.show_model_settings)
        self.neo4j_save_btn.clicked.connect(self.on_neo4j_save_click)
        self.neo4j_clear_btn.clicked.connect(self.on_neo4j_clear_click)
        self.model_save_btn.clicked.connect(self.on_model_save_click)
        self.model_clear_btn.clicked.connect(self.on_model_clear_click)

        # 初始化图形
        self.graph = None
        self.neo4j_saved = False

        # 创建主函数
        self.main_widget = QWidget()
        self.main_layout = QVBoxLayout(self.main_widget)
        self.main_layout.addWidget(self.neo4j_btn)
        self.main_layout.addWidget(self.model_btn)
        self.main_layout.addWidget(self.input_label)
```

```
        self.main_layout.addWidget(self.input_textbox)
        self.main_layout.addWidget(self.output_label)
        self.main_layout.addWidget(self.output_textbox)

        self.setCentralWidget(self.main_widget)

        # 创建和隐藏 Neo4j 的设置
        self.neo4j_settings_widget = QWidget()
        self.neo4j_layout = QVBoxLayout(self.neo4j_settings_widget)
        self.neo4j_layout.addWidget(self.neo4j_url_label)
        self.neo4j_layout.addWidget(self.neo4j_url_input)
        self.neo4j_layout.addWidget(self.neo4j_username_label)
        self.neo4j_layout.addWidget(self.neo4j_username_input)
        self.neo4j_layout.addWidget(self.neo4j_password_label)
        self.neo4j_layout.addWidget(self.neo4j_password_input)
        self.neo4j_layout.addWidget(self.neo4j_save_btn)
        self.neo4j_layout.addWidget(self.neo4j_clear_btn)
        self.neo4j_layout.addStretch(1)
        self.neo4j_settings_widget.hide()

        # 创建和隐藏模型的设置
        self.model_settings_widget = QWidget()
        self.model_layout = QVBoxLayout(self.model_settings_widget)
        self.model_layout.addWidget(self.api_label)
        self.model_layout.addWidget(self.api_input)
        self.model_layout.addWidget(self.model_save_btn)
        self.model_layout.addWidget(self.model_clear_btn)
        self.model_layout.addStretch(1)
        self.model_settings_widget.hide()  # Hide the Model settings widget

        # 创建提交和清除按钮
        self.output_save_btn = QPushButton("提交")
        self.output_clear_btn = QPushButton("清除")

        # 连接功能
        self.output_save_btn.clicked.connect(self.on_input_output_save_click)
        self.output_clear_btn.clicked.connect(self.on_input_output_clear_click)

        # 将提交和删除功能添加到对话框
        self.main_layout.addWidget(self.output_save_btn)
        self.main_layout.addWidget(self.output_clear_btn)

        self.contact_us_btn = QPushButton("联系我们")
        self.contact_us_btn.clicked.connect(self.show_contact_us_dialog)
        self.main_layout.addWidget(self.contact_us_btn)

        self.setCentralWidget(self.main_widget)
```

```
        # 创建“关于此系统”按钮
        self.about_btn = QPushButton(" 关于此系统 ")
        self.about_btn.clicked.connect(self.show_about_dialog)

        # 在设置主窗口的中央小部件之前，把“关于此系统”按钮添加到布局中
        self.main_layout.addWidget(self.about_btn)

        # 将主窗口的中央小部件设置为 main_widget
        self.setCentralWidget(self.main_widget)

    def show_neo4j_settings(self):
        if not self.neo4j_settings_widget.isVisible():
            self.neo4j_settings_widget.setWindowTitle("Neo4j 设置 ")
                                                    # 设置 Neo4j，设置子框的标题
            self.neo4j_settings_widget.show()
            self.neo4j_btn.setText(" 隐藏 Neo4j 设置 ")
        else:
            self.neo4j_settings_widget.hide()
            self.neo4j_btn.setText("Neo4j 设置 ")

    def show_model_settings(self):
        if not self.model_settings_widget.isVisible():
            self.model_settings_widget.setWindowTitle("Model 设置 ")
            # 设置 Model，设置子框的标题
            self.model_settings_widget.show()
            self.model_btn.setText(" 隐藏 Model 设置 ")
        else:
            self.model_settings_widget.hide()
            self.model_btn.setText("Model 设置 ")

    def on_neo4j_save_click(self):
        # 获取链接、用户名和密码
        url = self.neo4j_url_input.text()
        username = self.neo4j_username_input.text()
        password = self.neo4j_password_input.text()

        if not url or not username or not password:
            QMessageBox.warning(self, " 警告 ", " 请填写完整的 Neo4j 设置 ")
            return

        # 创建一个 Neo4j 案例
        self.graph = Neo4jGraph(url=url, username=username, password=password)
        self.neo4j_saved = True

        QMessageBox.information(self, " 提示 ", "Neo4j 保存按钮被单击了 ")

    def on_neo4j_clear_click(self):
        # 清除输入框的链接、用户名和密码
```

```
            self.neo4j_url_input.clear()
            self.neo4j_username_input.clear()
            self.neo4j_password_input.clear()

            # 弹出对话框
            message_box = QMessageBox(self)
            message_box.setText("Neo4j 内容被清除 ")
            message_box.setWindowTitle(" 提示 ")
            message_box.exec()

        def on_model_save_click(self):
            # 获取输入数据
            api_key = self.api_input.text()

            # 检查数据库是否已保存
            if not self.neo4j_saved:
                QMessageBox.warning(self, " 警告 ", " 先保存 Neo4j 设置 ")
                return

            # 使用 OpenAI 的 API 获取大语言模型
            chat_model = ChatOpenAI(temperature=0, openai_api_key=api_key)

            # 在创建链条前确保 self.graph 非空
            if self.graph is not None:
                # 使用 ChatGPT 的模型及 Neo4j 图数据库创建 GraphCypherQAChain 案例
                self.chain = GraphCypherQAChain.from_llm(chat_model, graph=self.
    graph, verbose=True)
                QMessageBox.information(self, " 提示 ", "Model 已经被保存 ")
            else:
                QMessageBox.warning(self, " 警告 ", " 先保存 Neo4j 设置 ")

            QMessageBox.information(self, " 提示 ", "Model 保存按钮被单击了 ")

        def on_model_clear_click(self):
            #Clear the content of "API" input box
            self.api_input.clear()

            message_box = QMessageBox(self)
            message_box.setText("model 内容被清除 ")
            message_box.setWindowTitle(" 提示 ")
            message_box.exec()

        def on_input_output_save_click(self):
            # 获取输入框的内容
            input_text = self.input_textbox.toPlainText()

            # 确保 Neo4j 设置已保存
            if not self.neo4j_saved:
```

```
                QMessageBox.warning(self, "警告", "先保存 Neo4j 设置")
                return

            # 确保模型已保存，并且输入框不为空
            if not hasattr(self, 'chain') or not input_text.strip():
                QMessageBox.warning(self, "警告", "先保存模型或输入内容")
                return
            # 使用 ChatOpenAI 模型处理输入，得到输出
            output_text = self.chain.run(f"""{input_text}""")

            # 将输出显示在输出框中
            self.output_textbox.setPlainText(output_text)

            QMessageBox.information(self, "提示", "已提交，请稍等")

        def on_input_output_clear_click(self):
            # Clear the content of input_textbox and output_textbox
            self.input_textbox.clear()
            self.output_textbox.clear()

            QMessageBox.information(self, "提示", "输入和输出框已经被清除")

        def show_contact_us_dialog(self):
            # 创建一个 QDialog 作为对话框
            contact_us_dialog = QDialog(self)
            contact_us_dialog.setWindowTitle("联系我们")

            # 创建一个 QVBoxLayout 布局，并将其设置为对话框的布局
            layout = QVBoxLayout(contact_us_dialog)

            # 添加联系我们的文本，使用 HTML 标签设置字体样式
            contact_us_label = QLabel()
            contact_us_label.setStyleSheet("font-size: 18px; font-family: Arial;
    text-align: center;")
            contact_us_label.setText("作者：李××<br>联系方式：150××××××××")
            layout.addWidget(contact_us_label)

            # 创建一个 QLabel，并将包含图片的 QPixmap 设置为它的图像
            contact_us_image_label = QLabel()
            contact_us_image = QPixmap("contact_us_image.jpg")
         # 替换为实际图片路径
            contact_us_image_label.setPixmap(contact_us_image)
            layout.addWidget(contact_us_image_label)

            # 将布局添加到对话框
            contact_us_dialog.setLayout(layout)

            # 显示对话框
```

```
            contact_us_dialog.exec()

        def show_about_dialog(self):
            # 创建一个 QDialog 用于“关于此系统”对话框
            about_dialog = QDialog(self)
            about_dialog.setWindowTitle(" 关于此系统 ")

            # 创建一个 QVBoxLayout 并将其设置为对话框的布局
            layout = QVBoxLayout(about_dialog)

            # 添加介绍性文本，使用 QLabel 和 HTML 标签设置样式
            about_text = """
            <h2> 欢迎使用史谈文博系统 v1.0</h2>
            <p> 在本系统的“Neo4j”和“Model”设置中您均可以调整其为自己的数据及模型。</p>
            <p> 在两个设置模块的下方，本系统提供了一个交互界面，您可以向它提问，然后它会回答
    您的问题。</p>
            <p> 请在左侧输入框中输入问题，然后单击“提交”按钮，在输出框查看回答。</p>
            <p> 希望您在使用过程中有愉快的体验。</p>
            <p> 如果您在使用过程中有任何问题，欢迎单击“联系我们”按钮，扫码与作者进行沟通。
    </p>
            <h3> 期待史谈文博系统 v2.0</h3>
            <p> 在史谈文博系统 v2.0，希望在其中可以加入语音转文字输入功能及长对话记忆功能。
    </p>
            <p> 期待您的喜欢与支持，如果您有更好的想法，也欢迎与我们进行合作。</p>
            """
            about_label = QLabel()
            about_label.setStyleSheet("font-size: 16px; font-family: Arial;")
            about_label.setText(about_text)
            layout.addWidget(about_label)

            # 添加一张图片，使用 QLabel 和 QPixmap
            about_image_label = QLabel()
            about_image = QPixmap("about_us.jpg")     # 替换为实际的图片路径
            about_image_label.setPixmap(about_image)
            layout.addWidget(about_image_label)

            # 设置对话框的布局
            about_dialog.setLayout(layout)

            # 显示对话框
            about_dialog.exec()


    if __name__ == "__main__":
        app = QApplication(sys.argv)
        window = MyMainWindow()
        window.show()
        sys.exit(app.exec())
```