实验四食用指南

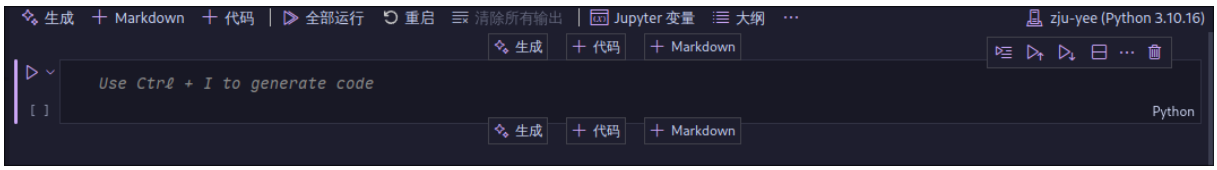
试鸢

一、前言

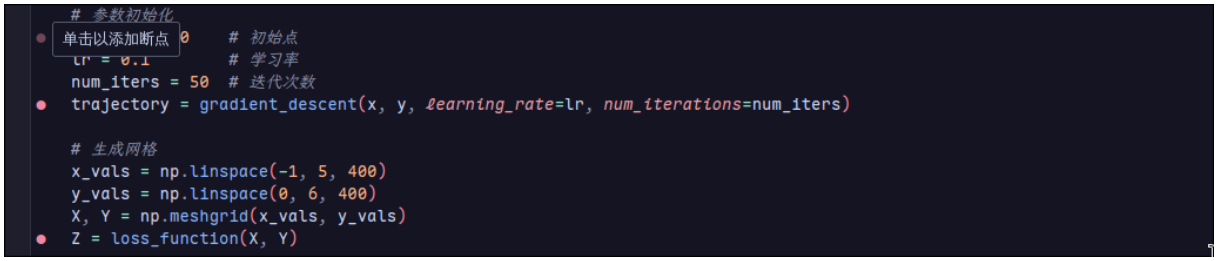
关于虚拟环境和 notebook 的配置，可以参考实验三的食用指南。我这边补充一下怎么使用 notebook（以 vsocde 环境为例）。

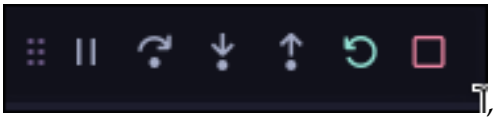
新建的 ipynb 文件都是空白的，你可以点击“+代码“来新增一个代码块来写入并运行你的代码。当你把鼠标移动到某一个单元格的上下方边框居中位置时，你也可以看到这三个按钮，显然在上方按会在上方插入，在下方反之。

ps：我不确定你们有没有生成选项，这个好像是跟 github copilot 有关的功能。



如果你学过 c 语言，应该比较熟悉这个小红点，这是断点，方便你进行调试





点击运行按钮边上的向下箭头后可以打开调试功能，当然如果代码没有问题可以点击运行而不用调试。

简单介绍一下右边图里的调试按钮的作用，

- 逐过程（第二个按钮）：调试时逐行执行代码，在遇到函数时不会进入函数体内。
- 单步执行（第三个按钮）：调试时逐行执行代码，在遇到函数时会进入函数体内。
- 单步跳出（第四个按钮）：跳出当前函数体，回到函数被调用的地方。

点击输出单元左侧的三个小点可以选择删除输出信息，像上图中的 warning 就无保留必要，可以选择删除。

二、题目 1：梯度下降

梯度下降的原理不作赘述，有兴趣可以简单看一下实验三指南，略有提及。
下面是参考代码

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# 自定义损失函数
def loss_function(x, y):
    return (x - 2) ** 2 + (y - 3) ** 2  # 修改为二次损失函数

# 对自定义损失函数的手动梯度计算
def gradient(x, y):
    partial_x = 2 * (x - 2)
    partial_y = 2 * (y - 3)
    return partial_x, partial_y

# 梯度下降算法
def gradient_descent(x, y, learning_rate=0.1, num_iterations=50):
    trajectory = [(x, y, loss_function(x, y))]
    for _ in range(num_iterations):
        grad_x, grad_y = gradient(x, y)
        x -= learning_rate * grad_x
        y -= learning_rate * grad_y
        trajectory.append((x, y, loss_function(x, y)))
    return np.array(trajectory)

# 参数初始化
x, y =  0, 0    # 初始点
lr = 0.1        # 学习率
num_iters = 50  # 迭代次数
trajectory = gradient_descent(x, y, learning_rate=lr,
num_iterations=num_iters)

# 生成3D坐标图底部网格
x_vals = np.linspace(-1, 5, 400)  # 这里决定了网格的范围
y_vals = np.linspace(0, 6, 400)   # 你可以调整参数并观察输出图像坐标轴的变化
                    # 我改成这个是为了对称，你可以都换成-1,5使梯度下降过程更直观
X, Y = np.meshgrid(x_vals, y_vals)
Z = loss_function(X, Y)

# 创建3D图形
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection="3d")  # 111表示1行1列第1个子图
# 图案样式，可以选择不同的cmap，具体可以查看文档，google，gpt；alpha表示透明度，
ax.plot_surface(X, Y, Z, cmap="viridis", alpha=0.7)

# 设置标签和标题
ax.set_xlabel("X axis")
ax.set_ylabel("Y axis")
ax.set_zlabel("Loss")
ax.set_title("Gradient Descent on Loss Function")

# 设置视角
ax.view_init(elev=30, azim=120)
```

```
# 初始化轨迹线和点
(trajectory_line,) = ax.plot([], [], [], color="red", linewidth=2,
label="Trajectory")
(point,) = ax.plot([], [], [], "ro")  # 当前点

# 更新动画的函数
def update(frame):
    trajectory_line.set_data(trajectory[: frame + 1, 0], trajectory[:
frame + 1, 1])
    trajectory_line.set_3d_properties(trajectory[: frame + 1, 2])

    # 更新当前点
    point.set_data([trajectory[frame, 0]], [trajectory[frame, 1]])
    point.set_3d_properties([trajectory[frame, 2]])

    # 更新视角
    ax.view_init(elev=30, azim=120 - frame * 2) # 旋转视角

    return trajectory_line, point

# 创建动画
ani = animation.FuncAnimation(fig, update, frames=len(trajectory),
interval=200, blit=False)
ax.legend()
plt.show()
```

理论上应该显示一个旋转的动画，但是在 jupyter notebook 中无法显示。运行 py 文件可以显示动图。

取消 ani 赋值会弹出 warning 信息，猜测是因为 cell 运行结束导致所有进程关闭，使得动画无法持续渲染。

你也可以注释 `ani=...`这一行，插入下面的代码，这样会显示一个静止的图像，方便在 jupyter notebook 中查看。

```
trajectory_line.set_data(trajectory[:, 0], trajectory[:, 1])
trajectory_line.set_3d_properties(trajectory[:, 2])
point.set_data([trajectory[-1, 0]], [trajectory[-1, 1]])
point.set_3d_properties([trajectory[-1, 2]])
```

然后给出一个自动微分函数，应该可以用来计算任意可微函数的梯度。

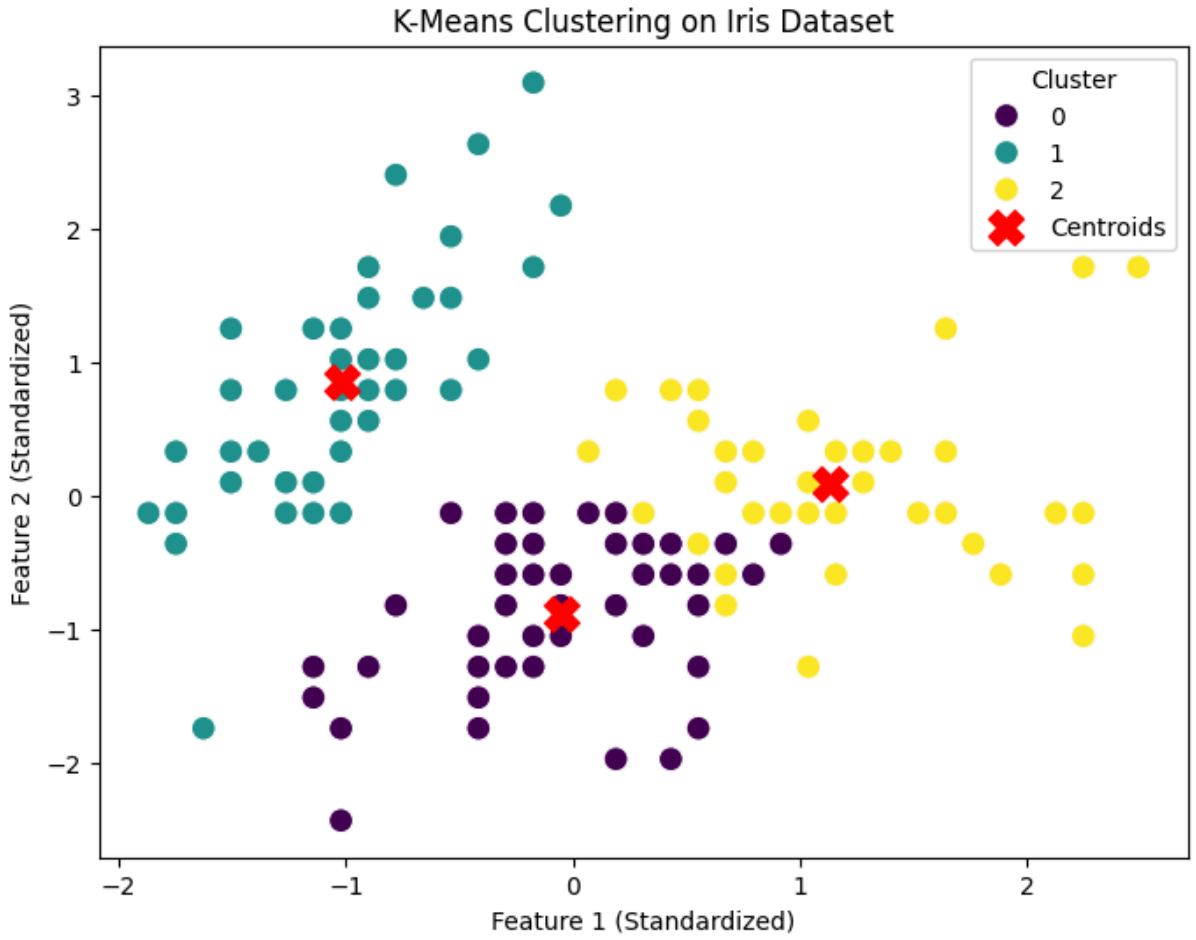
```
def auto_gradient(func,delta=1e-4):
    def wrapper(*args):
        vars_num = len(args)
        grads = []
        for i in range(vars_num):
            args_plus = list(args)
            # print(args_plus)
            args_plus[i] += delta
            args_minus = list(args)
            args_minus[i] -= delta
            grad = (func(*args_plus) - func(*args_minus)) / (2 * delta)
            grads.append(grad)
            # 四舍五入
            grads[i] = round(grads[i], 10)
        return tuple(grads)
    return wrapper
```

三、题目 2：鸢尾花聚类

K-means 是最常用的基于欧式距离的聚类算法，其认为两个目标的距离越近，相似度越大。

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score
from scipy.stats import mode

# 第一步：加载鸢尾花数据集
iris = datasets.load_iris()
X = iris.data
y = iris.target

# 第二步：数据预处理（标准化）
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 第三步：训练K-Means模型
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
kmeans.fit(X_scaled)

# 获取聚类标签
cluster_labels = kmeans.labels_

# 第四步：将聚类标签映射到实际类别标签
# 由于K-Means分配任意的聚类编号，我们需要将它们映射到实际类别
mapped_labels = np.zeros_like(cluster_labels) # 新建一个同大小的数组
for i in range(3): # 对每个聚类
    mask = (cluster_labels == i) # 获得一个布尔数组
    mapped_labels[mask] = mode(y[mask], keepdims=True).mode[0]
    # 使用mode函数计算每个聚类的实际类别,看docstring是在查找众数。

# 第五步：计算准确率
accuracy = accuracy_score(y, mapped_labels)
print(f"Accuracy of K-Means clustering: {accuracy:.2f}")

# 第六步：显示聚类组成
unique, counts = np.unique(cluster_labels, return_counts=True)
cluster_composition = dict(zip(unique, counts))
print("Cluster Composition:", cluster_composition)

# 第七步：可视化聚类
plt.figure(figsize=(8, 6))
sns.scatterplot(
  x=X_scaled[:, 0],  # 特征1
  y=X_scaled[:, 1],  # 特征2
  hue=cluster_labels,  # 聚类标签
  palette="viridis",
  s=100) # 聚类中心的大小

plt.scatter(
    kmeans.cluster_centers_[:, 0],  # 聚类中心的x坐标
```

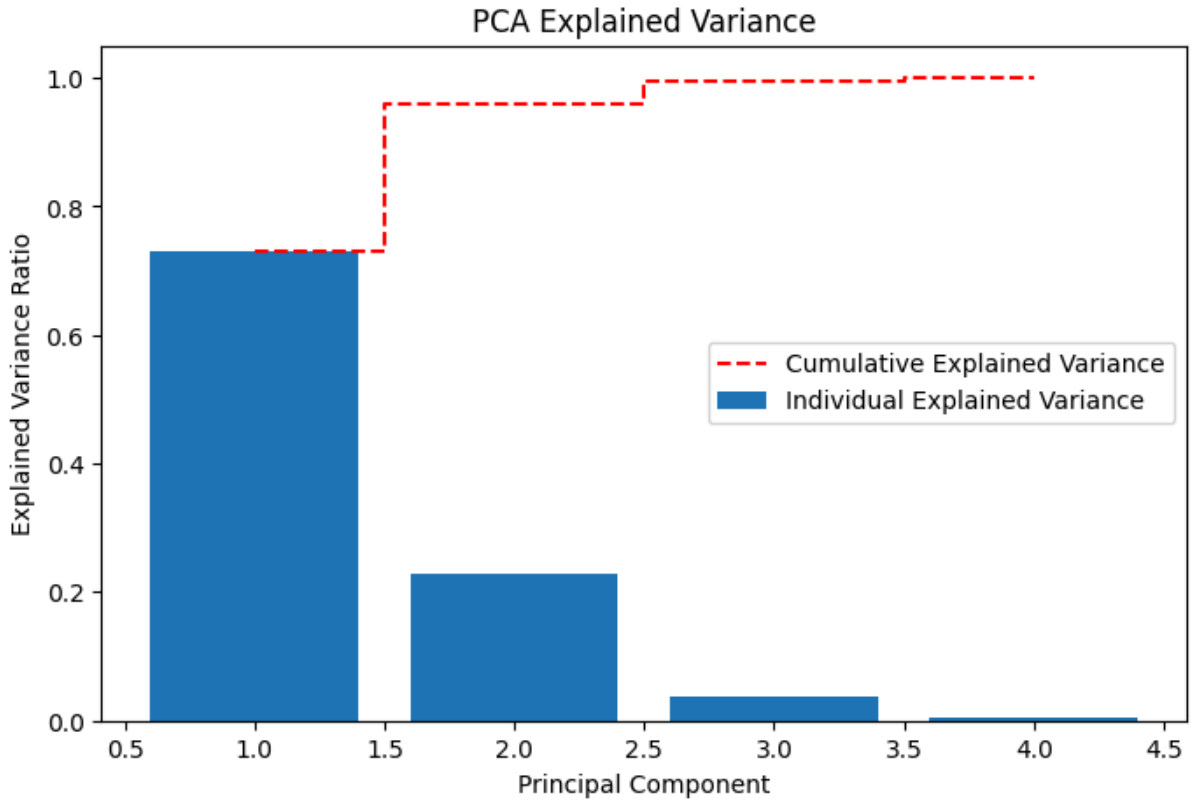
```
    kmeans.cluster_centers_[:, 1],  # 聚类中心的y坐标
    c="red", # 聚类中心的颜色
    marker="X", # 聚类中心的标记
    s=200, # 聚类中心的大小
    label="Centroids" # 聚类中心的标签)
plt.title("K-Means Clustering on Iris Dataset")
plt.xlabel("Feature 1 (Standardized)")
plt.ylabel("Feature 2 (Standardized)")
plt.legend(title="Cluster")
plt.show()

# 第八步：用新样本测试模型
sample_data = np.array([[5.1, 3.5, 1.4, 0.2]])
 # 示例鸢尾花样本
sample_scaled = scaler.transform(sample_data)
predicted_cluster = kmeans.predict(sample_scaled)

print(f"The sample data {sample_data} is predicted to belong to cluster
{predicted_cluster[0]}")
```



四、题目 3：PCA

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 第一步：加载鸢尾花数据集
iris = datasets.load_iris()
X = iris.data # 特征矩阵
y = iris.target # 真实标签

# 第二步：标准化特征
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 第三步：应用PCA
pca = PCA(n_components=None) # 保留所有4个成分以分析解释的方差
X_pca = pca.fit_transform(X_scaled)

# 第四步：分析主成分
explained_variance = pca.explained_variance_ratio_  # 每个成分解释的方差
cumulative_variance = np.cumsum(explained_variance)  # 累积方差
"""
Example:
  Explained variance ratios:
      [0.72962445 0.22850762 0.03668922 0.00517871]
  Cumulative explained variance:
      [0.72962445 0.95813207 0.99482129 1.        ]
"""

# 打印每个成分解释的方差
for i, variance in enumerate(explained_variance):
    print(f"Principal Component {i+1}: {variance:.2%} variance
explained")

# 第五步：绘制解释的方差
plt.figure(figsize=(8, 5))
plt.bar(
    range(1, len(explained_variance) + 1),
    explained_variance,
    align="center",
    label="Individual Explained Variance",
)

plt.step(
    range(1, len(cumulative_variance) + 1),
    cumulative_variance,
    where="mid", # mid, pre, post
    linestyle="--",
    color="red",
    label="Cumulative Explained Variance",
)
```

```
plt.xlabel("Principal Component")
plt.ylabel("Explained Variance Ratio")
plt.title("PCA Explained Variance")
plt.legend()
plt.show()
```



4.1 原理解释

假设你学过线性代数

4.1.1 基变换

假设在二维空间中有一个向量$(3,2)$,如果想求它在$(1,0)(0,1)$这组基下的坐标，可以直接内积

$$\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\cdot\begin{pmatrix}3\\ 2\end{pmatrix}=\begin{pmatrix}3\\ 2\end{pmatrix}$$

显然还是$(3,2)$。但是在$\left(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}}\right)\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}}\right)$这组基下的坐标，内积得到

$$\begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\end{pmatrix}\cdot\begin{pmatrix}3\\ 2\end{pmatrix}=\begin{pmatrix}\frac{5}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix}$$

这是矩阵的线性变换，不同的基可以对同样一组数据给出不同的表示，如果基的数量少于向量本身的维数，可认为向量被降维。问题在于：

如果我们有一组 N 维向量，现在要将其降到 K 维（K 小于 N），那么我们应该如何选择这 K 个基才能最大程度保留原有的信息？

很直观的想法是选择 K 个基使得它们尽可能正交，或者说它们之间的夹角尽可能大，且变量方差则尽可能大，这样就能最大程度保留原有的信息。

4.1.2 协方差

协方差是用来衡量两个变量之间的线性关系的统计量。协方差为 0 时，两个变量线性不相关。

其计算公式为：

$$Cov(a,b)=\frac{1}{n-1}\sum_{i=1}^{n}(a_i-\mu_a)(a_i-\mu_b)$$

其中，μ_a 和 μ_b 分别是变量 a 和 b 的均值。且，注意无偏估计使用$n-1$而不是n。但是数据样本量较大时，$n-1$和n的差别可以忽略不计。

数据标准化之后均值为 0，可认为协方差公式为：

$$Cov(a,b)=\frac{1}{n}\sum_{i=1}^{n}a_ib_i$$

4.1.3 协方差矩阵

令矩阵X为

$$X=\begin{pmatrix}a\\b\end{pmatrix}=\begin{pmatrix}a_1 & a_2 & a_3 & ... & a_n\\ b_1 & b_2 & b_3 & ... & b_n\end{pmatrix}$$

然后

$$\frac{1}{n}X\cdot X^T=\frac{1}{n}\begin{pmatrix}\sum_{i=1}^{n}a_i^2 & \sum_{i=1}^{n}a_ib_i\\ \sum_{i=1}^{n}a_ib_i & \sum_{i=1}^{n}b_i^2\end{pmatrix}=\begin{pmatrix}Cov(a,a) & Cov(a,b)\\ Cov(b,a) & Cov(b,b)\end{pmatrix}$$

推广到一般情况，对于 m 个 n 维向量，排列成矩阵$X_{n,m}$,令 $C=\frac{1}{m}X\cdot X^T$,则 C 是一个对称矩阵，其对角线分别对应各个变量的方差，而第 i 行 j 列和 j 行 i 列元素相同，表示 i 和 j 两个变量的协方差。

根据我们的目标，我们需要将除对角线外的其它元素化为 0，并且在对角线上将元素按大小从上到下排列（变量方差尽可能大）。

4.1.4 特征值分解

设原始数据矩阵 X 对应的协方差矩阵为 C，而 P 是一组基按行组成的矩阵，设 $Y=PX$，则 Y 为 X 对 P 做基变换后的数据。设 Y 的协方差矩阵为 D，则有：

$$D=\frac{1}{m}Y\cdot Y^T=\frac{1}{m}PX\cdot X^TP^T=\frac{1}{m}PCP^T$$

此时可以发现，优化目标变成了寻找一个矩阵 P，满足 PCP^T 是一个对角矩阵，并且对角元素按从大到小依次排列，那么 P 的前 K 行就是要寻找的基，用 P 的前 K 行组成的矩阵乘以 X 就使得 X 从 N 维降到了 K 维。

根据所学的线性代数知识，一个实对称矩阵必然可以找到 n 个单位正交的特征向量组成的矩阵E，使得E^TCE是一个对角矩阵。并且特征值按从大到小排列在对角线上。即：

$$E^TCE=\Lambda=\begin{pmatrix}\lambda_1 & & & \\ & \lambda_2 & & \\ & & ... & \\ & & & \lambda_n\end{pmatrix}$$

其中，λ_i 是 C 各特征向量对应的特征值。

到这里，我们发现我们已经找到了需要的矩阵 $P=E^T$。

P 是协方差矩阵的特征向量单位化后按行排列出的矩阵，其中每一行都是 C 的一个特征向量。如果设 P 按照 Λ 中特征值的从大到小，将特征向量从上到下排列，则用 P 的前 K 行组成的矩阵乘以原始数据矩阵 X，就得到了我们需要的降维后的数据矩阵 Y。

五、剩余 4 道算法题

以下全部由 copilot 生成，可能有错误，欢迎指正。

5.1 装箱子

```python
def box_packing(boxes, capacity):
    """
    装箱子问题求解。
    参数:
        boxes: 每个箱子的重量列表
        capacity: 箱子的最大容量
    返回:
        最少需要的箱子数量
    """
    boxes.sort(reverse=True)  # 按重量降序排序
    bins = []
    for box in boxes:
        placed = False
        for b in bins:
            if sum(b) + box <= capacity:
                b.append(box)
                placed = True
                break
        if not placed:
            bins.append([box])
    return len(bins)

# 示例
boxes = [4, 8, 1, 4, 2, 1]
capacity = 10
print(f"最少需要的箱子数量: {box_packing(boxes, capacity)}")
```

5.2 用动态规划实现斐波那契数列求解

```python
def fibonacci_dp(n):
    """
    用动态规划求解斐波那契数列。
    参数:
        n: 第 n 项
    返回:
        第 n 项的值
    """
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]

# 示例
n = 10
print(f"斐波那契数列第 {n} 项的值: {fibonacci_dp(n)}")
```

5.3 八数码问题

```python
from collections import deque

def eight_puzzle(start, goal):
    """
    八数码问题求解。
    参数:
        start: 初始状态（元组）
        goal: 目标状态（元组）
    返回:
        最短步数
    """
    def neighbors(state):
        idx = state.index(0)
        x, y = divmod(idx, 3)
        moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                nidx = nx * 3 + ny
                new_state = list(state)
                new_state[idx], new_state[nidx] = new_state[nidx],
new_state[idx]
                moves.append(tuple(new_state))
        return moves

    queue = deque([(start, 0)])
    visited = set()
    visited.add(start)

    while queue:
        state, steps = queue.popleft()
        if state == goal:
            return steps
        for neighbor in neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, steps + 1))

# 示例
start = (1, 2, 3, 4, 5, 6, 7, 8, 0)
goal = (1, 2, 3, 4, 5, 6, 0, 7, 8)
print(f"八数码问题最短步数: {eight_puzzle(start, goal)}")
```

5.4 用回溯法实现 n 皇后问题求解

```python
def solve_n_queens(n):
    """
    用回溯法求解 n 皇后问题。
    参数:
        n: 棋盘大小
    返回:
        所有解的列表，每个解是一个列表，表示每行皇后所在的列索引
    """
    def is_valid(board, row, col):
        for i in range(row):
            if board[i] == col or abs(board[i] - col) == abs(i - row):
                return False
        return True

    def backtrack(row):
        if row == n:
            solutions.append(board[:])
            return
        for col in range(n):
            if is_valid(board, row, col):
                board[row] = col
                backtrack(row + 1)
                board[row] = -1

    solutions = []
    board = [-1] * n
    backtrack(0)
    return solutions

# 示例
n = 8
solutions = solve_n_queens(n)
print(f"{n} 皇后问题共有 {len(solutions)} 个解")
```