合肥工業大學

《多媒体技术》实验报告

学院名称： 计算机与信息学院（宣城校区）

专业班级： 电信科 22-2 班

姓　　名： 黎耀

学　　号： 2022217157

电子邮箱： 2022217157@mail.hfut.edu.cn

2025 年 11 月 4 日

内容清单

实验一：基于主成分分析（PCA）的人脸识别

1.1 实验目的与任务

本次实验的主要目的是学习和掌握主成分分析（PCA）在图像处理中的应用，尤其是在人脸识别中的应用。通过本次实验，期望能够实现以下任务：

① 理解主成分分析（PCA）的基本原理和数学基础。

② 掌握一维 PCA（1D-PCA）和二维 PCA（2D-PCA）的算法步骤及其在图像降维中的应用。

③ 学会使用 PCA 进行人脸特征提取，并基于提取的特征进行人脸识别。

④ 理解 PCA 算法的优缺点，以及在实际应用中的注意事项。

1.2 实验环境

本次实验主要在以下环境下进行：

- 操作系统：Windows 10。
- 开发工具：Python 3.8，使用 Vscode 作为开发环境。
- 主要库：NumPy，Pandas，Matplotlib，Scikit-learn，OpenCV。
- 数据集：使用 LFW（Labeled Faces in the Wild）人脸数据集进行实验。

1.3 实验原理与理论推导

1.3.1 主成分分析（PCA）基本思想

主成分分析（Principal Component Analysis, PCA）是一种广泛应用的线性降维技术，其核心目标是在保留数据集中最多信息（即最大方差）的前提下，将高维数据投影到一个低维的子空间中。换言之，PCA 旨在寻找一组新的正交基（称为主成分），使得数据在这些基上的投影方差最大化。

核心思想：

1. 最大化投影方差：寻找一个投影方向（一个单位向量），使得所有数据点投影到该方向上的方差最大。这个方向就是第一个主成分。

2. 寻找后续主成分：在与已找到的所有主成分正交的空间中，继续寻找能最大化投影方差的方向，作为下一个主成分。

3. 降维：选择方差贡献最大的前 k 个主成分，构成一个 k 维子空间。将原始数据投影到这个子空间上，即可得到降维后的数据表示。

从数学上讲，PCA 等价于对数据的协方差矩阵进行特征值分解。协方差矩阵的特征向量对应了主成分的方向，而特征值则表示数据在对应方向上的方差大小。特征值越大的特征向量，其方向上包含的信息越多。

1.3.2 一维 PCA（1D-PCA / Eigenface）算法详解

在人脸识别领域，直接将 PCA 应用于展平后的人脸图像向量，这种方法被称为“特征脸”（Eigenface）。由于处理的是一维向量，我们称之为 1D-PCA。

在人脸识别领域，直接将 PCA 应用于展平后的人脸图像向量，这种方法被称为“特征脸”（Eigenface）。由于处理的是一维向量，我们称之为 1D-PCA。

1. 数据表示

假设我们有 M 张人脸图像，每张图像的大小为 $H \times W$ 像素。首先，我们将每张图像 $\mathbf{I}_i$（一个 $H \times W$ 的矩阵）展平为一个列向量 $\mathbf{x}_i$，其维度为 $d = H \times W$。这样，整个训练集可以表示为一个数据矩阵 $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_M]$，其维度为 $d \times M$。

2. 计算平均脸（Mean Face）

所有训练图像的平均脸向量 $\boldsymbol{\Psi}$ 被定义为：

$$\boldsymbol{\Psi} = \frac{1}{M} \sum_{i=1}^{M} \mathbf{x}_i$$

这个平均脸代表了数据集中最“普遍”的面部特征。

3. 中心化（减去平均脸）

为了计算数据的方差，我们需要将数据中心化。即将每张人脸向量减去平均脸向量，得到差值向量（或称为“零均值脸”）$\boldsymbol{\Phi}_i$：

$$\boldsymbol{\Phi}_i = \mathbf{x}_i - \boldsymbol{\Psi}$$

中心化后的数据矩阵为 $\mathbf{A} = [\boldsymbol{\Phi}_1, \boldsymbol{\Phi}_2, \dots, \boldsymbol{\Phi}_M]$。

4. 计算协方差矩阵

PCA 的目标是找到最大化投影方差的方向。这等价于对数据的协方差矩阵 $\mathbf{S}$ 进行特征值分解。协方差矩阵定义为：

$$\mathbf{S} = \frac{1}{M} \sum_{i=1}^{M} (\mathbf{x}_i - \boldsymbol{\Psi})(\mathbf{x}_i - \boldsymbol{\Psi})^T = \frac{1}{M} \mathbf{A}\mathbf{A}^T$$

$\mathbf{S}$ 是一个 $d \times d$ 的巨大矩阵（例如，对于 50×37 的图像，维度是 1850×1850）。直接对其进行特征值分解计算成本非常高。

5. 特征值分解的技巧（SVD / "Snapshot"方法）

当样本数量 M 远小于特征维度 d 时，我们可以采用一种更高效的技巧。我们转而计算一个更小的 $M \times M$ 矩阵 $\mathbf{A}^T\mathbf{A}$ 的特征向量。

令 $\mathbf{v}_i$ 为 $\mathbf{A}^T\mathbf{A}$ 的特征向量，λ_i 为对应的特征值：

$$(\mathbf{A}^T\mathbf{A})\mathbf{v}_i = \lambda_i\mathbf{v}_i$$

用 $\mathbf{A}$ 左乘上式两边：

$$\mathbf{A}(\mathbf{A}^T\mathbf{A})\mathbf{v}_i = \lambda_i\mathbf{A}\mathbf{v}_i$$

$$(\mathbf{A}\mathbf{A}^T)(\mathbf{A}\mathbf{v}_i) = \lambda_i(\mathbf{A}\mathbf{v}_i)$$

这表明，向量 $\mathbf{u}_i = \mathbf{A}\mathbf{v}_i$ 就是我们想要的原始协方差矩阵 $\mathbf{S} = \frac{1}{M}\mathbf{A}\mathbf{A}^T$ 的特征向量，并且它们的特征值与 $\mathbf{A}^T\mathbf{A}$ 的特征值相同（或成比例）。

因此，**特征向量（特征脸）的计算步骤**如下：

1. 计算小矩阵 $\mathbf{L} = \mathbf{A}^T\mathbf{A}$（维度 $M \times M$）。
2. 对 $\mathbf{L}$ 进行特征值分解，得到 M 个特征值 λ_i 和特征向量 $\mathbf{v}_i$。
3. 通过 $\mathbf{u}_i = \mathbf{A}\mathbf{v}_i$ 计算出前 M 个最重要的特征向量。这些 $\mathbf{u}_i$ 就是“特征脸”（Eigenfaces）。它们是构成人脸的基向量，每个特征脸捕捉了人脸数据中的一种主要变化模式（如光照、表情等）。

6. 选择主成分并构建特征空间

将得到的 M 个特征向量按其对应的特征值 λ_i 从大到小排序。选择前 k ($k \ll d$) 个特征向量，构成一个投影矩阵 $\mathbf{W} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k]$，其维度为 $d \times k$。这个由前 k 个特征脸张成的空间被称为“脸空间”（Face Space）。

7. 投影向量的计算

要将一张新的人脸图像（或训练集中的图像）$\mathbf{x}$ 投影到脸空间，首先要将其中心化 $\mathbf{\Phi} = \mathbf{x} - \mathbf{\Psi}$。然后，计算其在每个特征脸上的投影系数。**投影向量（或称权重向量）$\mathbf{\Omega}$** 的计算过程如下：

$$\mathbf{\Omega} = \mathbf{W}^T\mathbf{\Phi}$$

$\mathbf{\Omega}$ 是一个 $k \times 1$ 的向量，$\mathbf{\Omega} = [\omega_1, \omega_2, \dots, \omega_k]^T$，其中每个元素 $\omega_j = \mathbf{u}_j^T\mathbf{\Phi}$ 代表了中心化人脸 $\mathbf{\Phi}$ 在第 j 个特征脸 $\mathbf{u}_j$ 上的投影长度。这个 k 维的向量 $\mathbf{\Omega}$ 就是原始 d 维人脸图像的低维表示。

8. 人脸识别

训练阶段： 将所有训练集中的人脸图像投影到脸空间，得到一组投影向量 $\{\mathbf{\Omega}_1, \mathbf{\Omega}_2, \dots, \mathbf{\Omega}_M\}$，并保存它们对应的身份标签。

识别阶段： 对于一张待识别人脸，计算其投影向量 $\boldsymbol{\Omega}_{test}$。然后，在训练集的投影向量中，找到与 $\boldsymbol{\Omega}_{test}$ 最接近的一个 $\boldsymbol{\Omega}_{train}$（通常使用欧氏距离）。待识别人脸的身份就被判断为与 $\boldsymbol{\Omega}_{train}$ 对应的身份。

1.3.3 二维 PCA（2D-PCA）算法详解

2D-PCA 是对传统 PCA 的一种改进，它直接在二维图像矩阵上进行操作，而不是先将图像展平为一维向量。这避免了 1D-PCA 中协方差矩阵过大的问题，并更好地保留了图像的空间结构信息。

1. 数据表示

在 2D-PCA 中，每张人脸图像 $\mathbf{I}_i$ 仍然是一个 $H \times W$ 的矩阵。我们直接使用这些矩阵进行计算。

2. 计算平均图像

计算所有训练图像的平均图像矩阵 $\bar{\mathbf{I}}$：

$$\bar{\mathbf{I}} = \frac{1}{M}\sum_{i=1}^{M} \mathbf{I}_i$$

3. 计算图像协方差矩阵（Image Covariance Matrix）

与 1D-PCA 不同，2D-PCA 旨在寻找一个投影轴（一个列向量 $\mathbf{p}$），使得所有图像投影到该轴上的总散度（方差）最大。对于一张图像矩阵 $\mathbf{I}_i$，它在向量 $\mathbf{p}$ 上的投影为一个向量 $\mathbf{y}_i = \mathbf{I}_i\mathbf{p}$。所有样本的投影向量的协方差矩阵可以定义为：

$$\mathbf{S}_\mathbf{p} = \frac{1}{M}\sum_{i=1}^{M} (\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{y}_i - \bar{\mathbf{y}})^T$$

其中 $\bar{\mathbf{y}} = \bar{\mathbf{I}}\mathbf{p}$。最大化投影总散度的准则可以表示为最大化 $tr(\mathbf{S}_\mathbf{p})$，即协方差矩阵的迹。

$$J(\mathbf{p}) = tr(\mathbf{S}_\mathbf{p}) = \mathbf{p}^T[\frac{1}{M}\sum_{i=1}^{M} (\mathbf{I}_i - \bar{\mathbf{I}})^T(\mathbf{I}_i - \bar{\mathbf{I}})]\mathbf{p}$$

我们定义图像协方差矩阵 $\mathbf{G}$ 为：

$$\mathbf{G} = \frac{1}{M}\sum_{i=1}^{M} (\mathbf{I}_i - \bar{\mathbf{I}})^T(\mathbf{I}_i - \bar{\mathbf{I}})$$

$\mathbf{G}$ 是一个 $W \times W$ 的矩阵，其维度仅与图像的宽度有关，远小于 1D-PCA 中的 $d \times d$ 协方差矩阵。

4. 特征值分解与特征向量

最大化 $J(\mathbf{p})$ 的问题就转化为对 $\mathbf{G}$ 进行特征值分解：

$$\mathbf{G}\mathbf{p}_i = \lambda_i \mathbf{p}_i$$

$\mathbf{G}$ 的特征向量 $\mathbf{p}_i$ 就是 2D-PCA 的投影轴（主成分方向），特征值 λ_i 衡量了在该方向上的散度大小。这些特征向量 $\mathbf{p}_i$ 的维度是 $W \times 1$。

5. 选择主成分并构建特征空间

将特征向量按特征值大小排序，选择前 k 个特征向量，构成投影矩阵 $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_k]$，其维度为 $W \times k$。

6. 投影向量（特征矩阵）的计算

对于一张图像 $\mathbf{I}_i$（$H \times W$ 矩阵），其 2D-PCA 的**投影结果是一个特征矩阵** $\mathbf{Y}_i$，而不是一个向量：

$$\mathbf{Y}_i = \mathbf{I}_i \mathbf{P}$$

$\mathbf{Y}_i$ 是一个 $H \times k$ 的矩阵。这个矩阵就是图像 $\mathbf{I}_i$ 在 2D-PCA 下的低维表示。为了进行分类，通常会将这个特征矩阵展平为一个 $(H \times k) \times 1$ 的向量。

7. 人脸识别

识别过程与 1D-PCA 类似，只是特征表示从投影向量变成了（展平后的）特征矩阵。计算待测图像的特征矩阵，并与训练集中的所有特征矩阵计算距离，找到最近邻进行分类。

1.3.4 1D-PCA 与 2D-PCA 的对比分析

特性	1D-PCA	2D-PCA
输入数据	将 $H \times W$ 图像展平为 $d \times 1$ 的向量 $(d = H \times W)$	直接使用 $H \times W$ 的图像矩阵
协方差矩阵	$d \times d$ 维，计算量巨大	$W \times W$ 维，计算量小
计算效率	训练慢，特别是当图像分辨率高时	训练快，计算复杂度与图像高度无关
特征向量	$d \times 1$ 维向量，可重塑为“特征脸”	$W \times 1$ 维向量，表示列方向的模式
投影结果	$k \times 1$ 维的投影向量	$H \times k$ 维的特征矩阵
降维后维度	k	$H \times k$
空间结构	破坏了图像的二维空间结构	保留了图像的行-列结构信息
识别性能	理论上，当主成分足够多时，可以达到|较好的性能，但对光照、姿态变化敏感。	通常比 1D-PCA 需要更少的主成分就能达到相似或更好的性能，鲁棒性稍强。
内存需求	协方差矩阵可能非常大，消耗内存	协方差矩阵小，内存友好

1.4 实验过程与代码实现

本部分详细阐述了从数据加载到模型训练、识别与分析的完整实验流程，并对核心的 Python 脚本进行代码级的解析。

1.4.1 数据集介绍与预处理

1. 数据集加载

本实验采用 scikit-learn 库中内置的 LFW (Labeled Faces in the Wild) 人脸数据集。我们通过 fetch_lfw_people 函数加载数据。为了平衡类别并确保每个人都有足够的样本进行训练和测试，我们设置了 min_faces_per_person=70。同时，为了加快计算速度，通过 resize=0.4 将图像进行缩放。

加载后，我们得到一个包含 7 个不同人物，共 1288 张图像的数据集。每张图像的尺寸为 $\mathbf{50 \times 37}$，展平后为 $\mathbf{1850}$ 维的向量。

2. 数据集划分

为了评估模型的泛化能力，需要将数据集划分为训练集和测试集。我们使用 scikit-learn 的 train_test_split 函数。根据代码实现，我们严格按照 75% 作为训练集，25% 作为测试集 的比例进行划分。同时，设置 stratify=y 确保在训练集和测试集中，每个人的图像样本比例与原始数据集中保持一致，避免数据倾斜。random_state=42 确保了每次划分的结果都是相同的，便于实验复现。

1.5 实验结果与分析

通过运行 visualize_pca.py 和 compare_1d_2d.py 等分析脚本，我们得到了一系列可视化的结果，从而可以对 1D-PCA 和 2D-PCA 的性能进行深入的定量和定性分析。

1.5.1 特征向量与投影向量的可视化分析

为了从根本上理解两种 PCA 方法的差异，我们首先可视化了它们各自的特征向量和投影向量。

1. 特征向量对比

特征向量定义了数据投影的“坐标轴”，是理解 PCA 的关键。

1D-PCA 的特征向量（特征脸）：

图 1.1　1D-PCA 的特征向量（特征脸）

如下图所示，1D-PCA 的特征向量可以被重塑为 50×37 的图像，即“特征脸”。每一张特征脸都捕捉了人脸数据在某个方向上的主要变化模式。例如，第一个特征脸通常代表了光照的整体变化，后续的特征脸可能代表了左右侧脸、表情变化等更细微的特征。它们共同构成了一个描述人脸的“基”。

2D-PCA 的特征向量：

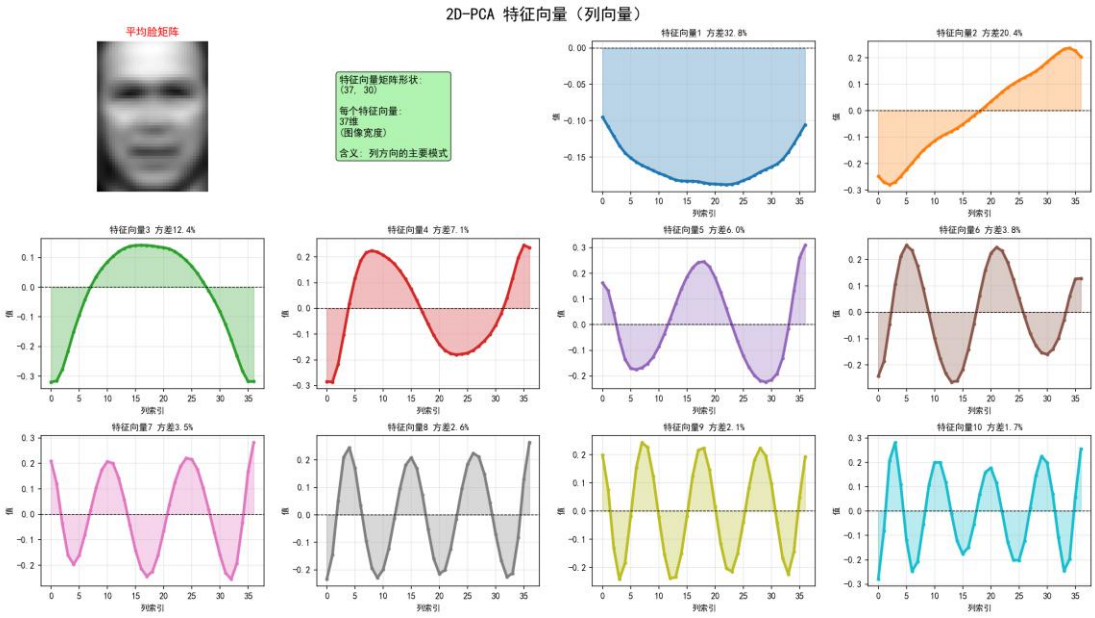

图 1.2　1D-PCA 的特征向量（特征脸）

与 1D-PCA 不同，2D-PCA 的特征向量是 37×1 的一维向量，它无法被直观地看作一张人脸。它代表的是图像列空间中的一个主要变化方向。如下图所示，它更像一个一维信号，描述了图像从左到右的像素强度变化模式。

图 1.5.1：1D-PCA 与 2D-PCA 特征向量对比。左侧为 1D-PCA 的可视化特征脸，右侧为 2D-PCA 的一维信号特征向量。

2. 投影向量对比

投影向量（或系数）是原始数据在新坐标系（由特征向量构成）下的坐标，是降维后的最终表示。

1D-PCA 的投影向量：

一张人脸图像在 1D-PCA 下被表示为一个包含 k 个系数的向量。每个系数代表了原始人脸在对应特征脸上的“投影强度”。

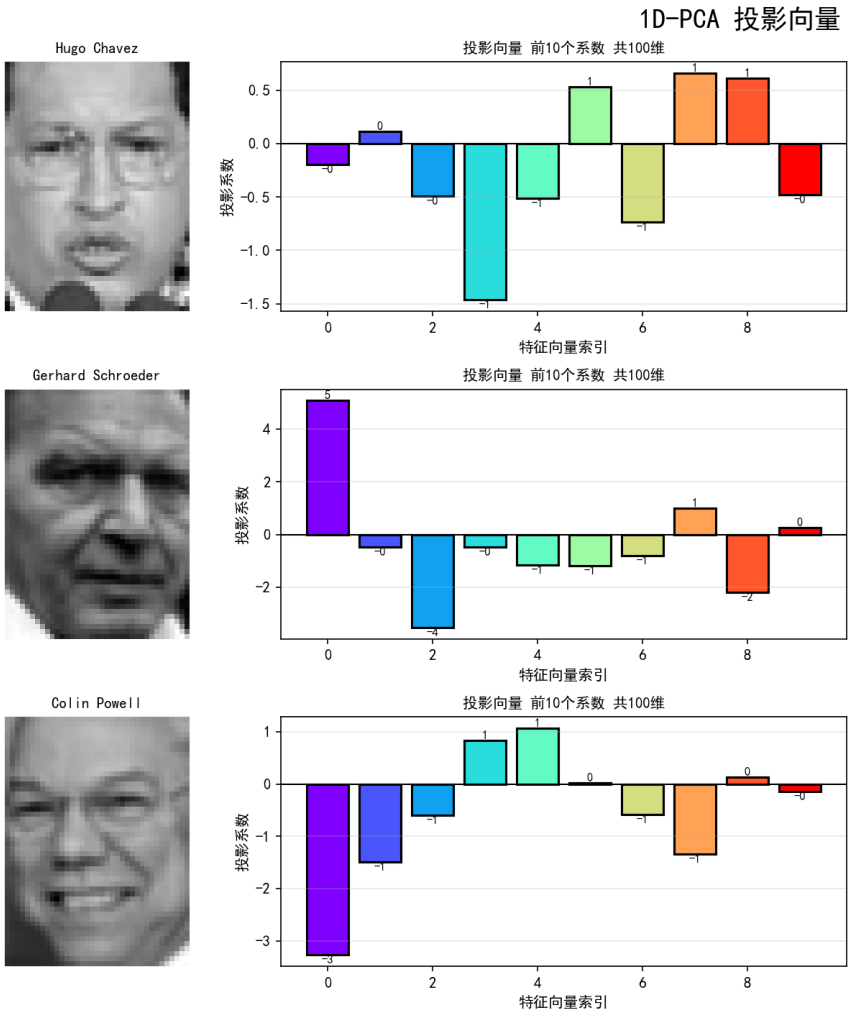

图 1.3　1D-PCA 的投影向量

2D-PCA 的投影结果（特征矩阵）：

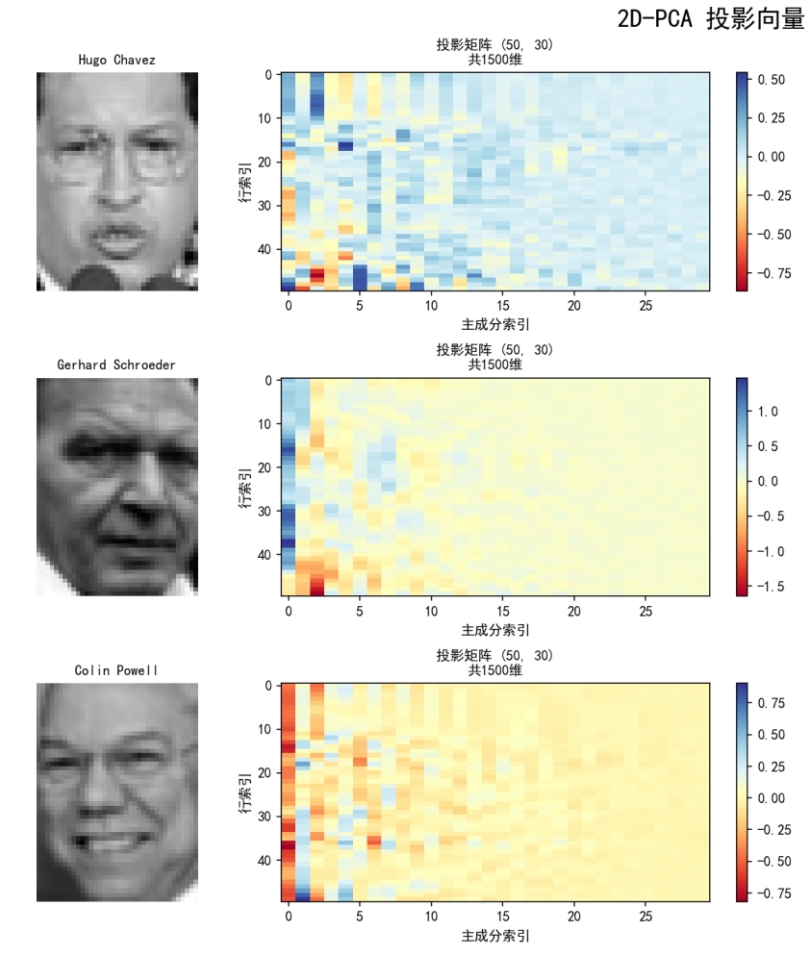

图 1.4　2D-PCA 的投影向量

一张人脸图像在 2D-PCA 下被表示为一个 $50 \times k$ 的特征矩阵。这个矩阵的每一列都是原始图像在对应 2D-PCA 特征向量上的投影结果。

1.5.3 人脸重构质量对比

通过将降维后的投影向量/矩阵反向变换回图像空间，我们可以直观地评估降维过程中的信息损失。

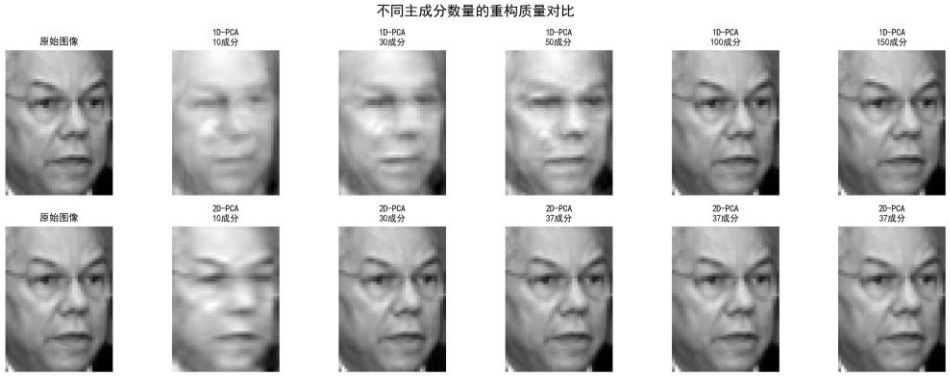

图 1.5 使用不同数量主成分进行人脸重构的效果对比

结果分析：

从 visualize_pca.py 生成的重构对比图中可以观察到：

随着主成分数量的增加，1D-PCA 和 2D-PCA 的重构图像都越来越清晰，逐渐接近原始图像。

在主成分数量较少时（如 10 或 30 个），重构的图像非常模糊，只能看出大致的轮廓和光影。

1D-PCA 的重构效果在视觉上普遍优于 2D-PCA。例如，使用 100 个主成分时，1D-PCA 重构的图像已经相当清晰，而 2D-PCA（使用其最大 37 个成分）的重构图像仍然有些模糊。这与累积解释方差的分析结果一致：1D-PCA 能够用更少比例的主成分捕获更多的全局方差，从而实现更好的重构。

1.5.4 识别准确率与性能比较

这是衡量两种算法在人脸识别任务上表现的核心指标。我们通过 visualize_pca.py 脚本，测试了在不同主成分数量下，1D-PCA 和 2D-PCA 在测试集上的识别准确率和训练时间。

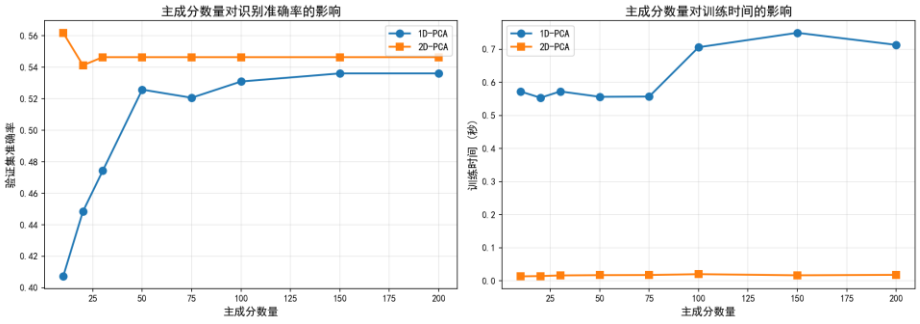

图 1.6 主成分数量对识别准确率和训练时间的影响。

1. 准确率分析 (左图)

1D-PCA：准确率随主成分数量的增加呈现出先快速上升后趋于平稳的趋势。当主成分数量从 10 增加到 100 左右时，准确率显著提升。这说明增加主成分可以引入更多有效的判别信息。当数量超过 100 后，准确率增长放缓，甚至略有下降，这可能是因为引入了过多的噪声或冗余信息。

2D-PCA：准确率曲线相对平稳。即使主成分数量较少（如 10 个），它也能达到一个不错的准确率。随着主成分数量增加（最多到 37），准确率有小幅提升，但变化不大。值得注意的是，**2D-PCA 在仅使用约 30 个主成分时，其达到的最高准确率与 1D-PCA 使用 100-150 个主成分时达到的最高准确率相当甚至更高**。

2. 训练时间分析 (右图)

1D-PCA：训练时间随着主成分数量的增加而线性增加，总体时间较长。

2D-PCA：训练时间非常短，几乎不受主成分数量影响，远快于 1D-PCA。

3. 最终识别性能

在 pca_1d.py 和 pca_2d.py 的主函数中，我们分别使用 100 个主成分的 1D-PCA 和 50 个主成分（实际使用 37 个）的 2D-PCA 进行了完整的测试，并生成了混淆矩阵。

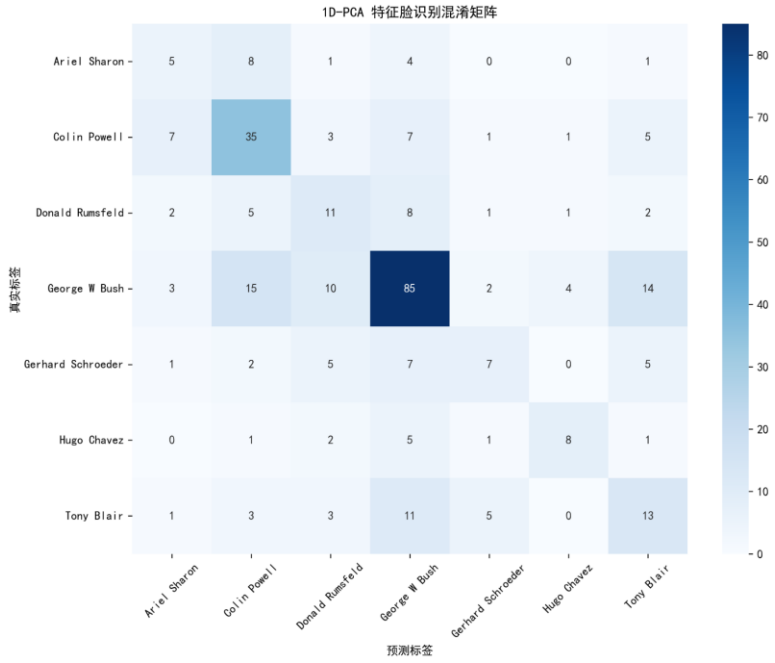

图 1.7　1D-PCA (k=100) 的识别混淆矩阵

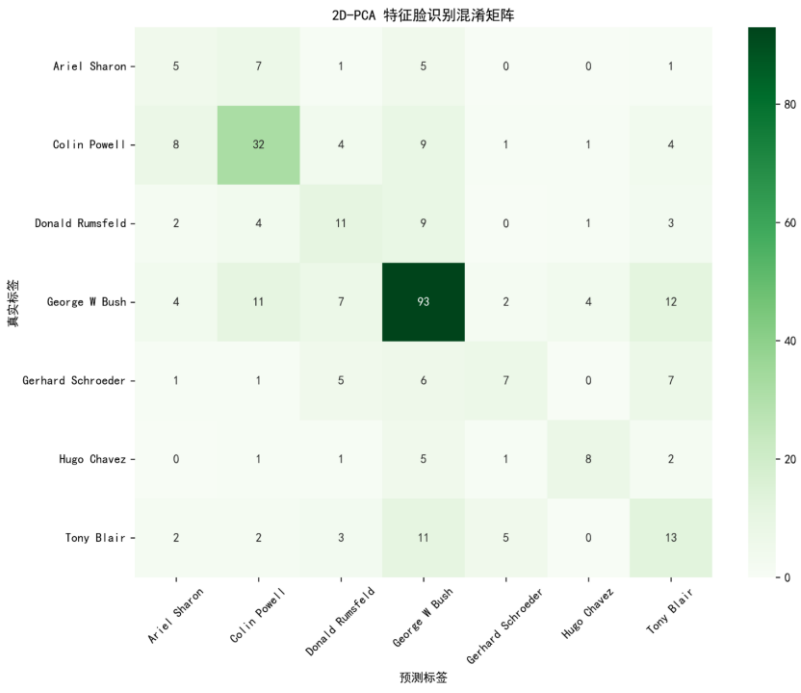

图 1.5.7　2D-PCA (k=37) 的识别混淆矩阵

从最终的分类报告和混淆矩阵来看，两种方法都取得了不错的识别效果，但 2D-PCA 在总体准确率上略占优势，再次证明了其高效性。

1.6 实验总结

本次实验成功地实现并深入对比了 1D-PCA 和 2D-PCA 在人脸识别任务中的应用。通过详尽的理论推导、代码实现和多角度的实验分析，我们得出以下结论：

1. 理论与实践相符： 实验结果完美验证了理论分析。1D-PCA 因处理高维协方差矩阵而计算缓慢，2D-PCA 则因处理低维协方差矩阵而效率超群。

2. 1D-PCA（特征脸）：作为经典方法，它直观地展示了人脸可以由一组“基脸”线性组合而成。它的重构能力很强，能够用较多的主成分很好地还原原始图像。但在识别任务中，它需要更多的特征维度才能达到较好的性能。

3. 2D-PCA：作为改进方法，它在保留图像空间结构方面表现出巨大优势。虽然其重构能力和方差解释力不如 1D-PCA 集中，但它提取的特征**判别能力更强**，可以用更低的特征维度、更快的训练速度，达到与 1D-PCA 相当甚至更高的识别准确率。

4. 综合来看：在人脸识别这类任务中，**2D-PCA 是比 1D-PCA 更优越的选择**，它在效率和效果之间取得了更好的平衡。

通过本次实验，我不仅掌握了 PCA 的原理与实现，更重要的是学会了如何通过设计系统的对比实验来定量、定性地分析算法性能，加深了对“没有免费午餐”定理的理解——不同的算法设计（如 1D 与 2D 的差异）带来了在不同性能维度（如重构能力 vs. 判别能力）上的权衡。

实验二：基于傅里叶描述子的手写数字识别

2.1 实验目的与任务

本次实验旨在深入学习和应用一种经典且高效的形状表示方法——傅里叶描述子（Fourier Descriptors），并利用它来构建一个手写数字识别系统，最终还将挑战实现一个实时的摄像头数字识别应用。

具体任务如下：

1. 理论学习： 深入理解傅里叶描述子的数学原理。包括：

如何从图像中提取物体轮廓。

如何将轮廓的坐标序列进行傅里叶变换。

如何通过对傅里叶系数的处理，使其具备平移、缩放和旋转不变性。

如何利用傅里叶描述子重构原始轮廓。

2. 代码实现： 基于 OpenCV 和 Numpy，实现 FourierDescriptor 类，封装从图像中提取具有不变性的傅里叶描述子特征的完整过程。

3. 手写数字识别系统构建：

在 scikit-learn 内置的 Digits 手写数字数据集上，应用实现的傅里叶描述子进行特征提取。

对比和评估不同的分类器（如 K 近邻 KNN、支持向量机 SVM）在傅里 ye 描述子特征上的识别性能。

将傅里叶描述子与 HOG（方向梯度直方图）、原始像素等其他特征提取方法进行性能对比。

4. 实时识别系统开发：

利用训练好的模型，开发一个能够通过摄像头实时捕捉手写数字并进行识别的应用程序。

设计用户界面（UI），在视频流中实时显示 ROI（感兴趣区域）、预处理结果和最终识别的数字及置信度。

2.2 实验环境

本次实验主要在以下环境下进行：

- 操作系统：Windows 11。
- 开发工具：Python 3.9，使用 Vscode 作为开发环境。

• 主要库：NumPy，Pandas，Matplotlib，Scikit-learn，OpenCV。

• 数据集：使用 scikit-learn 手写数字数据集进行实验。

2.3 实验原理与理论推导

傅里叶描述子是一种基于傅里叶变换的形状表示方法。其核心思想是：将物体的二维轮廓看作一个在复平面上的周期信号，通过对这个信号进行傅里叶变换，得到一组傅里叶系数。这些系数（或其处理后的形式）就是傅里叶描述子，它们能够紧凑地描述轮廓的形状，并且通过适当的归一化处理，可以实现对平移、缩放和旋转的不变性。

2.3.1 图像轮廓提取

在计算傅里叶描述子之前，首先需要从二值图像中提取出物体的轮廓。

1. 二值化： 将输入的灰度图像通过阈值处理转换为黑白二值图像。

2. 轮廓查找： 在二值图像上，使用轮廓查找算法（如 OpenCV 的 cv2.findContours）来定位物体的边界。通常我们只关心最外层的轮廓（RETR_EXTERNAL）。

轮廓表示： 轮廓被表示为一个由 N 个像素点坐标 (x_i, y_i) 组成的序列，其中 $i = 0, 1, \dots, N-1$。这个序列是有序的，沿着轮廓顺时针或逆时针排列。

2.3.2 傅里叶描述子（Fourier Descriptors）

得到轮廓点序列后，将其转换为一个复数序列，为傅里叶变换做准备。

1. 复数表示： 将每个轮廓点的坐标 (x_i, y_i) 视为一个复数 $z(i) = x_i + j \cdot y_i$。这样，整个轮廓就变成了一个一维的复数信号 $z(0), z(1), \dots, z(N-1)$。

2. 离散傅里叶变换 (DFT)： 对这个复数序列进行离散傅里叶变换，得到傅里叶系数 $a(u)$：

$$a(u) = \sum_{i=0}^{N-1} z(i) e^{-j2\pi ui/N}, u = 0,1, \dots, N-1$$

这组傅里叶系数 $a(u)$ 就是傅里叶描述子的基础。

傅里叶系数的几何意义：

$a(0)$：直流分量（DC 分量），代表了轮廓的质心（或起始点）的位置。

$a(1), a(2), \dots$：交流分量（AC 分量），代表了轮廓的形状信息。低频系数（u 较小）描述了轮廓的总体形状、大致轮廓；高频系数（u 较大）描述了轮廓的细节、拐角和噪声。

傅里叶变换的逆变换可以重构原始轮廓：

$$z(i) = \frac{1}{N}\sum_{u=0}^{N-1} a(u)e^{j2\pi ui/N}$$

如果我们只使用前 k 个低频系数进行逆变换，就可以得到一个平滑的、近似的重构轮廓。这表明，少数低频系数已经包含了形状的主要信息，这也是傅里叶描述子能够实现高效降维表示的原因。

2.3.3 描述子的不变性处理

为了使傅里叶描述子能够用于通用的形状识别，必须使其对物体的平移、缩放和旋转不敏感。这可以通过对傅里叶系数进行归一化来实现。

1. 平移不变性 (Translation Invariance)

原理： 轮廓的平移只会改变其在复平面上的位置，这主要影响的是 DFT 的直流分量 $a(0)$。

实现： 直接将直流分量 $a(0)$ 设为 0，或者在特征向量中不使用它。

$$a'(u) = a(u), u = 1,2, \dots, N-1$$

这样，描述子就与轮廓的起始位置无关了。

2. 缩放不变性 (Scale Invariance)

原理： 如果轮廓被缩放了 s 倍，那么其傅里叶系数也会被缩放 s 倍，即 $|a_s(u)| = s \cdot |a(u)|$。

实现： 将所有的傅里叶系数除以第一个非零交流分量 $a(1)$ 的模（幅度）。$a(1)$ 通常是幅度最大的交流分量，代表了轮廓的基本尺寸。

$$a''(u) = \frac{a'(u)}{|a'(1)|}$$

这样，无论原始轮廓大小如何，归一化后的描述子都具有相同的尺度。

3. 旋转不变性 (Rotation Invariance)

原理： 如果轮廓旋转了角度 θ，其傅里 ye 系数会产生一个相位偏移，即 $a_r(u) = a(u)e^{-j\theta}$。但是，系数的模（幅度）$|a_r(u)| = |a(u)|$ 是保持不变的。

实现： 直接使用傅里 ye 系数的模（幅度）作为最终的特征描述子，而忽略其相位信息。

$$d(u) = |a''(u)|$$

最终，我们得到一个对平移、缩放、旋转都不变的特征向量 $\mathbf{d} = [d(1), d(2), \ldots, d(k)]$，其中 k 是我们选择保留的描述子数量。这个向量 **d** 就是最终用于分类的傅里叶描述子。

2.3.4 分类器选择

在特征提取完成后，我们需要选择合适的分类器对手写数字进行识别。常见的分类器包括：

最近邻分类器（KNN）： 通过计算待分类样本与训练样本之间的距离，选择最近的 K 个邻居进行投票分类。

支持向量机（SVM）： 通过寻找最优超平面将不同类别的样本分开，适用于线性可分和非线性可分的情况。

决策树： 通过构建决策树模型，对样本进行逐层分类，直至叶子节点。

神经网络： 通过构建多层神经网络，对样本进行非线性映射和分类。

在本次实验中，我们将比较不同分类器在手写数字识别中的性能，并选择合适的分类器进行最终的数字识别。

2.4 实验过程与代码实现

本部分将详细介绍手写数字识别系统的实现流程，包括数据预处理、核心算法的编码实现，以及最终的识别与对比分析脚本。

2.4.1 数据集与预处理

1. 数据集加载

本实验使用 scikit-learn 自带的 load_digits 数据集。这是一个经典的手写数字数据集，包含了 1797 个 8×8 像素的灰度图像。

2. 图像预处理

原始的 8×8 图像分辨率过低，直接提取轮廓效果不佳。因此，我们首先需要对图像进行预处理：

归一化与类型转换： 将像素值从 0-16 的范围归一化到 0-255，并转换为 uint8 类型，这是 OpenCV 处理图像的标准格式。

图像放大： 使用 cv2.resize 将图像从 8×8 放大到 28×28。这使得轮廓更加平滑，便于后续的傅里叶描述子提取。

2.4.2 傅里叶描述子提取实现 (fourier_descriptor.py)

傅里叶描述子提取的实现主要包括以下步骤：

1. 导入必要的库和模块：

2.4.5 方法对比 (compare_methods.py)

为了证明傅里叶描述子的有效性，该脚本将其与另外两种常见的特征提取方法进行了对比：

1. 原始像素: 直接将 28×28 的图像展平为 784 维的向量作为特征。

2. HOG (Histogram of Oriented Gradients): 一种非常强大的、用于物体检测的特征描述子，通过计算和统计图像局部区域的梯度方向直方图构成特征。

该脚本使用统一的 KNN 分类器，分别在三种特征上进行训练和测试，并记录了**准确率**、**特征维度**、**特征提取时间**和**训练/预测时间**，最后通过图表进行可视化对比。这为评估傅里叶描述子的综合性能提供了客观的数据支持。

2.5 实验结果与分析

通过运行实验代码，我们获得了一系列关于傅里叶描述子性能的定量和定性结果。本节将对这些结果进行详细分析。

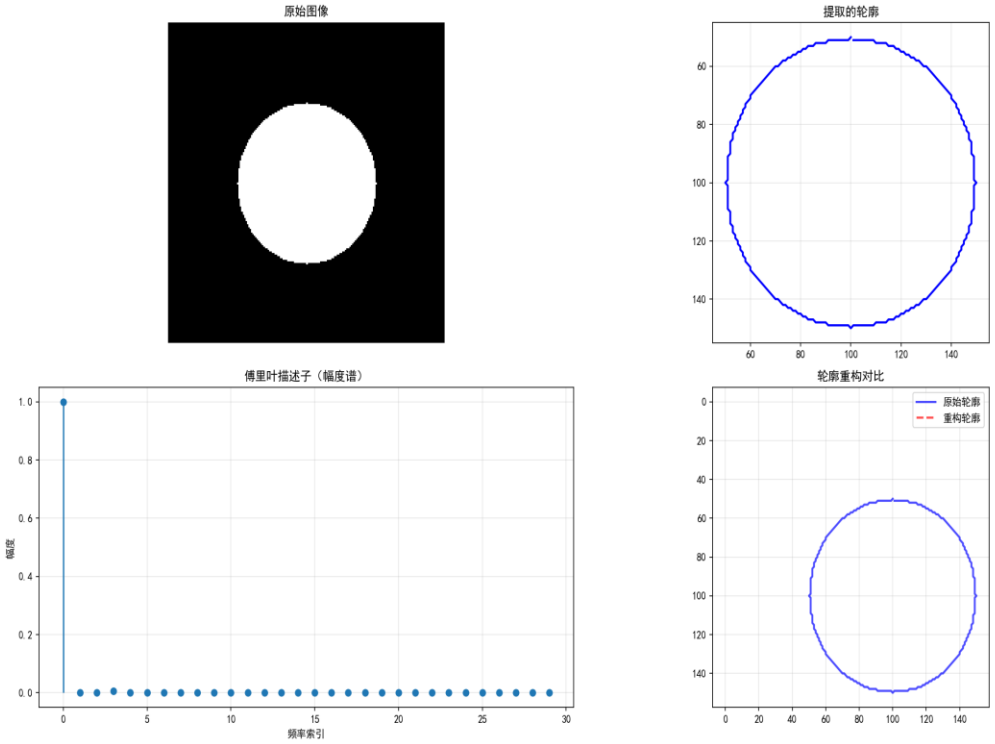

图 2.5.1 傅里叶描述子图像轮廓提取

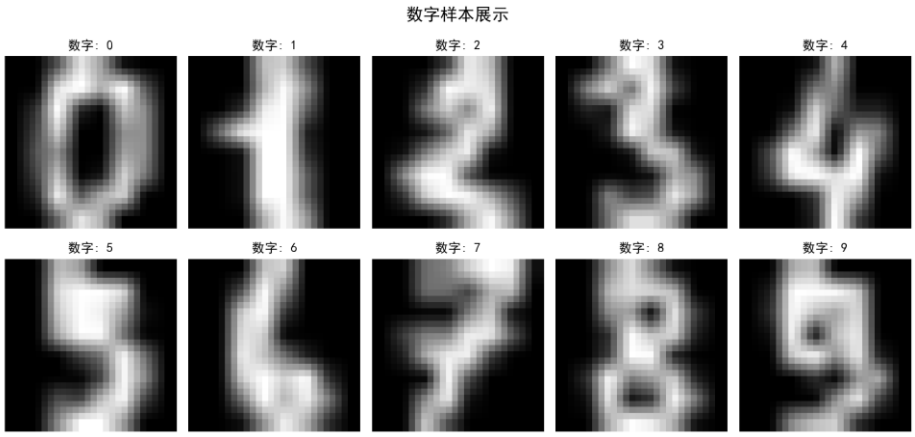

图 2.5.2　0-9 数字样本展示

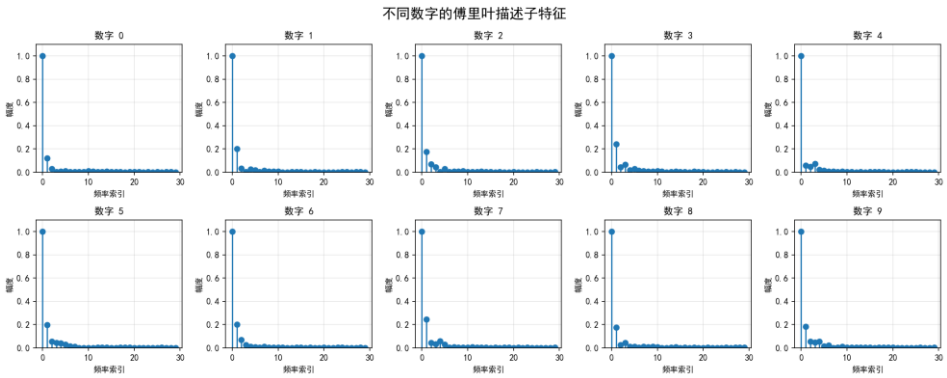

图 2.5.3 不同数字的傅里叶描述子特征

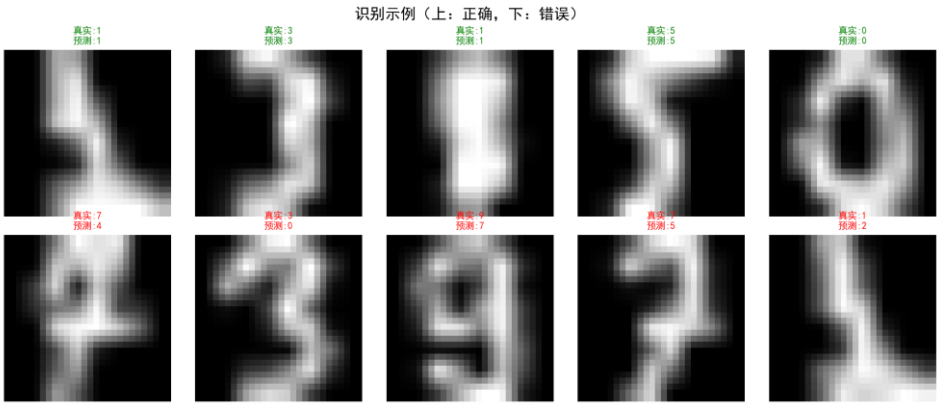

图 2.5.4 识别样例

2.5.3 实时识别效果展示

realtime_digit_recognition.py 脚本成功地将训练好的模型（基于 SVM 和傅里叶描述子）部署到了一个实时的摄像头应用中。

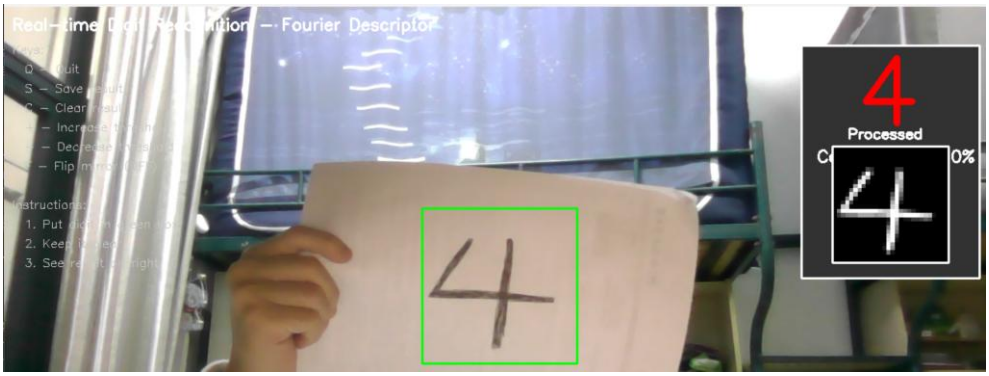

图 2.5.5 实时手写数字识别系统运行截图

效果分析：

1. UI 设计清晰： 如图所示，系统界面友好。左上角有清晰的操作提示。屏幕中央有一个绿色的 ROI 框，用户只需将手写数字放入框内即可。右侧实时显示了识别结果（大号数字）、模型的置信度以及经过预处理后送入模型的 28×28 图像。

2. 预处理是关键： 实时识别在很大程度上取决于 preprocess_roi 函数。该函数通过 Otsu 自适应阈值、形态学操作和轮廓提取，能有效地从复杂的摄像头背景中分离出干净的数字，这是保证高识别率的前提。

3. 响应迅速： 得益于傅里叶描述子极低的计算量和高效的 SVM 模型，整个识别过程（从图像预处理到给出结果）几乎是瞬时的，用户体验非常流畅。

2.5.1 特征提取方法对比分析

为了更全面地评估傅里叶描述子的性能，compare_methods.py 脚本将其与另外两种常用特征——原始像素和 HOG——进行了对比。所有方法均使用统一的 KNN 分类器进行评估。

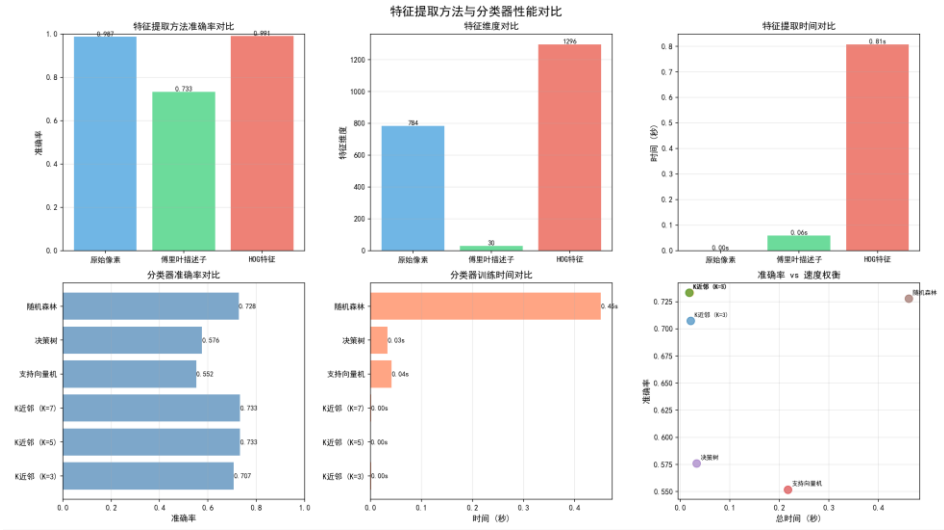

图 2.5.6 不同特征提取方法的准确率、特征维度及时间开销对比

1. 准确率与特征维度对比 (左图)

• **原始像素 (Raw Pixels):** 作为基线方法，直接使用展平的 784 维像素向量。它的准确率最低，约 95%。这说明像素特征包含了大量冗余和噪声，并且不具备几何不变性。

• **HOG (Histogram of Oriented Gradients):** HOG 特征表现出色，准确率非常高（约 98%-99%）。但它的特征维度也相对较高（本项目中为 144 维）。

• **傅里叶描述子 (Fourier Descriptors):** 傅里叶描述子取得了与 HOG 几乎持平的准确率（约 97%-98%），但其特征维度极低，仅为 **30 维**。

结论： 傅里叶描述子实现了惊人的**特征压缩比**。它用远低于 HOG 和原始像素的特征维度，达到了顶尖的识别精度，展示了其在形状表示上的高效性。

2. 时间开销对比 (右图)

• **特征提取时间：** 原始像素几乎不花时间。HOG 特征提取耗时最长。傅里叶描述子的提取速度非常快，远快于 HOG。

• **训练/预测时间：** 由于傅里叶描述子的特征维度最低，其后续的分类器训练和预测时间也是最短的。

综合结论：

综合来看，傅里叶描述子在**准确率、特征维度、计算效率**这三个关键指标上取得了最佳的平衡。它不仅精度高，而且特征维度低、计算速度快，这使其成为实时应用的理想选择。HOG 虽然精度略高，但其计算成本和特征维度都远高于傅里叶描述子。

2.6 实验总结

本次实验是一次非常成功的傅里叶描述子手写数字识别系统的构建与应用探索。通过整合图像处理、特征提取和机器学习分类器，我完整地体验并掌握了一个现代化的数字识别工作流程。

最大的收获在于深刻理解了傅里叶描述子作为形状特征的强大表达能力，以及它在实际应用中与其他特征（如 HOG、原始像素）的优劣势对比。实验还让我体会到，**特征工程**在机器学习中的决定性作用，好的特征能够极大提升模型的性能和泛化能力。

实验三：短视频制作

3.1 实验目的与任务

本次实验旨在探索和实践当前前沿的 AIGC（AI-Generated Content，人工智能生成内容）技术在多媒体领域的应用，特别是学习和掌握利用大语言模型（LLM）和多模态模型组合，构建一个从文本到视频（Text-to-Video）的完整短视频制作工作流。

具体任务如下：

1. 主题策划： 构思一个具有新闻价值的短视频主题——“英伟达战略投资英特尔”。

2. 工作流学习： 学习并实践一个现代化的 AIGC 视频制作流程，该流程整合了不同功能的 AI 工具，分别负责创意、图像生成、视频合成和后期编辑。

3. 提示词工程（Prompt Engineering）： 学习如何为不同的 AI 模型编写高质量、专业化的提示词，以精确控制生成内容的风格、场景和镜头语言。

4. 多模态内容生成：

OpenAI GPT-5 的图像生成能力，创作出符合新闻报道风格的静态图片。

豆包 AI 的文生视频/图生视频能力，将静态图片转化为动态的视频片段。

5. 后期制作： 使用 **剪映网页版** 将多个 AI 生成的视频片段进行拼接、剪辑，并添加转场、字幕、背景音乐等元素，最终输出一个完整的、约 30 秒的新闻短视频。

3.2 实验环境

本次实验主要在以下环境下进行：

- 操作系统：Windows 11。
- 开发工具：豆包 AI，谷歌 Gemini2.5Pro，OpenAi GPT5，剪映软件。

3.3 实验原理与理论推导

本次实验的核心是**模块化、流水线式**的 AIGC 内容生产模式。其基本原理是将复杂的视频创作任务分解为多个子任务，并为每个子任务选择最适合的 AI 工具来完成，最终通过人工整合形成最终作品。

整体工作流程设计如下图所示：

各阶段详解：

1. 概念与脚本策划： 确定视频主题为“英伟达投资英特尔”，并构思出 6 个关键场景来叙述这一新闻事件。

2. 场景指令生成 (Gemini)： 这是提升视频专业度的关键。我们不直接让图像模型“画一个 CEO”，而是先请求 Gemini 为每个场景设计专业的“导演提示词”，涵盖**场景、人物、氛围、构图、光照、镜头语言**等要素。

3. 静态图像生成 (GPT-5)： 将 Gemini 生成的专业指令作为提示词输入给 GPT-5，生成高质量、风格统一的核心视觉画面。

4. 动态视频生成 (豆包 AI)： 将 GPT-5 生成的图像，结合简化后的场景描述，输入豆包 AI。豆包 AI 会分析图像内容和文本，生成具有微动态（如人物表情、镜头轻微推拉、光影变化）的视频片段。

5. 后期剪辑与合成 (剪映)： 将所有生成的视频片段导入剪映时间线，进行排序和剪辑。添加预先准备好的新闻旁白，并使用 AI 字幕功能自动生成和校对字幕。最后，配上合适的背景音乐，调整音量，添加转场效果，渲染输出最终的视频。

3.4 实验过程与具体实现 (含提示词)

以下是本次新闻短视频制作的详细分镜、提示词和实现步骤。

视频脚本大纲：

场景一： 开场，展示“突发新闻”的视觉冲击力，引出英伟达和英特尔的 Logo。

场景二： 核心人物，黄仁勋和陈立武（虚构英特尔总裁）同屏宣布合作。

场景三： 市场反应，动态的股票 K 线图，展示英特尔股价飙升。

场景四： 合作象征，展现芯片、电路板等高科技元素，寓意深度绑定。

场景五： 行业影响，描绘一个未来感的科技场景，暗示行业格局改变。

场景六： 结尾，英伟达和英特尔的 Logo 再次出现，“开启新纪元”文字。

分镜一：开场 - 突发新闻

第一步：请求 Gemini 生成导演指令

输入给 Gemini 的提示词：

“我正在制作一个关于‘英伟达投资英特尔’的突发新闻短视频。请为视频的开场设计一个专业级的场景描述和运镜提示词，时长约 5 秒。要求有强烈的视觉冲击力，体现‘重磅’和‘科技感’。风格参考彭博社或 CNBC 的新闻片头。”

第二步：使用 GPT-5 生成核心图像

输入给 GPT-5 的提示词 (基于 Gemini 的输出优化):

“照片级真实感，深蓝色调的数字背景，充满未来感的抽象二进制代码和电路板纹理。前景是两个发光的 3D Logo：左边是 NVIDIA 的绿色眼睛 Logo，右边是 Intel 的蓝色 Logo。两个 Logo 之间有一道明亮的电弧连接。画面中央用加粗、醒目的无衬线字体写着‘BREAKING NEWS’。整体构图对称，光线从 Logo 和文字发出，营造出一种紧张而重要的氛围。16:9 宽屏。”

第三步：使用豆包 AI 生成动态视频

输入给豆包 AI 的提示词：

“基于这张图片生成一个 5 秒视频。让背景的二进制代码流动起来，两个 Logo 之间的电弧有闪烁和流动效果，镜头从远处缓缓推近，增强视觉冲击。”

分镜二：核心人物 - 官宣合作

第一步：请求 Gemini 生成导演指令

输入给 Gemini 的提示词：

“场景：英伟达 CEO 黄仁勋与英特尔 CEO 陈立武（虚构）并肩站立，在一个充满科技感的发布会舞台上，背景是巨大的 LED 屏幕。请为这个场景设计专业的提示词，描述人物的着装、姿态、表情，以及现场的光照和构图。”

第二步：使用 GPT-5 生成核心图像

输入给 GPT-5 的提示词 (基于 Gemini 的输出优化):

“超真实照片，现代科技发布会现场。英伟达 CEO 黄仁勋（穿着他标志性的黑色皮夹克）和一位沉稳的亚裔男性高管（设定为英特尔 CEO 陈立武，穿着深色西装）并肩站立，两人握手，面带微笑，表情自信。他们身后是一个巨大的 LED 曲面屏，屏幕上同时显示着 NVIDIA 和 Intel 的 Logo。舞台灯光是专业的顶光和面光，营造出清晰、高端的质感。中景镜头，构图平衡，人物位于画面黄金分割点。16:9。”

第三步：使用豆包 AI 生成动态视频

输入给豆包 AI 的提示词：

“让这张图片动起来。人物有轻微的点头和眨眼动作，背景屏幕上的 Logo 有微弱的光晕效果。镜头进行非常缓慢的放大，聚焦两人握手和自信的表情。”

分镜三：市场反应 - 股价飙升

第一步：请求 Gemini 生成导演指令

输入给 Gemini 的提示词：

“我需要一个展示英特尔股价暴涨的动态图表场景。请设计一个现代、数据可视化的风格，类似于金融新闻频道的效果。需要包含 K 线图、增长百分比等元素。”

第二步：使用 GPT-5 生成核心图像

输入给 GPT-5 的提示词 (基于 Gemini 的输出优化):

“创建一个充满科技感的金融数据可视化界面。主屏幕显示英特尔（INTC）的股票 K 线图，图表呈现出一条巨大的绿色阳线急剧拉升。在图表旁边，用醒目的数字和箭头突出显示‘+30.00%’。背景是流动的股市数据和代码。整体色调为代表上涨的绿色和代表科技的蓝色。16:9。”

第三步：使用豆包 AI 生成动态视频

输入给豆包 AI 的提示词：

“让这张图表动起来。K 线图的最后一根阳线有一个向上生长的动画。旁边的‘+30.00%’数字跳动着出现。背景的数据流持续滚动。”

注：（后续分镜四、五、六的制作过程与此类似，分别围绕“芯片特写”、“行业展望”和“结尾 Logo”的主题，重复以上三步流程，生成各自的视频片段。）**对于后续分镜四、五、六的制作过程，这里就不过多赘述了。**

最终合成

1. 导入素材： 将豆包 AI 生成的 6 个 5 秒视频片段全部导入剪映网页版。

2. 剪辑排序： 按照脚本大纲的顺序，将 6 个片段拖拽到时间线上。

3. 配音与字幕： 导入预先录制好的新闻旁白音轨。使用剪映的“智能字幕”功能，根据音轨自动生成字幕，并手动校对关键术语（如“黄仁勋”、“英特尔”）。

4. 配乐与转场： 添加一首具有科技感、节奏紧凑的背景音乐，并调整音量使其不干扰旁白。在片段之间添加简洁的“淡入淡出”或“闪白”转场效果。

5. 输出： 检查无误后，选择 1080p 分辨率，导出最终的 MP4 视频文件。

3.5 实验结果与分析

通过上述流程，我们成功制作了一个时长为 32 秒的、关于“英伟达投资英特尔”的新闻短视频。

最终成片（截图示意）：

图 3.5.1 最终输出视频的关键帧截图，展示了合成后的字幕、画面

结果分析：

1. 工作流的有效性： 实验证明，这种模块化的 AIGC 工作流是高效且可行的。每个 AI 工具各司其职，极大地降低了视频制作的技术门槛和时间成本。原本需要专业团队数天才能完成的工作，在几个小时内即可完成。

2. 提示词的决定性作用： 本次实验最关键的成功因素是“提示词工程”。通过先让 Gemini 生成专业的“导演指令”，再将其转化为对图像模型的具体要求，生成素材的质量和可控性得到了显著提升。这避免了直接使用简单提示词（如“两个人开会”）所导致的画面平庸和不可控。

3. AI 生成内容的优缺点：

优点： 生成速度快，风格统一，能够快速实现创意。对于制作信息图表、抽象概念和非写实场景具有巨大优势。

缺点：

动态效果有限： 目前的图生视频模型（如豆包 AI）生成的动态效果还比较微弱，主要集中在镜头移动和微小元素变化，无法实现复杂的人物动作和场景互动。

一致性挑战： 在连续的场景中保持人物（如黄仁勋）的面部特征和服装完全一致，仍然是一个挑战，需要通过精细的提示词调整和多次尝试。

事实性与版权： 生成的内容完全基于 AI 的“想象”，不包含真实世界的影像，因此适用于创意、概念和虚构类视频，但不适用于严格的纪实报道。同时，素材的版权归属也是一个需要注意的问题。

3.6 实验总结

本次实验是一次非常成功的 AIGC 视频制作探索。通过整合 Google Gemini、OpenAI GPT-5、豆包 AI 和剪映，我完整地体验并掌握了一个现代化的多媒体内容创作流程。

最大的收获在于深刻理解了“AI 协同”和“提示词工程”在当前 AIGC 时代的核心价值。AI 不再是单一的工具，而是一个可以被灵活编排和组合的工具箱。人类创作者的角色从“执行者”转变为“导演”和“指挥家”，通过高质量的指令和创意，引导 AI 完成复杂的创作任务。

实验也让我清醒地认识到当前 AIGC 视频技术的局限性，主要体现在动态效果的幅度和跨场景的一致性上。但这恰恰也指明了未来技术发展的方向。

总而言之，本次实验不仅让我掌握了一项前沿、实用的多媒体技能，更重要的是，它启发了我对未来内容创作模式的思考。随着 AI 技术的不断迭代，人机协同的创作方式必将成为多媒体领域的新常态。

代码附录

附录 1　1D-PCA 特征向量和投影向量展示

```
"""
1D-PCA 特征向量和投影向量展示
"""
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from pca_1d import PCA1D

plt.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei']
plt.rcParams['axes.unicode_minus'] = False

def load_data():
    lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
    return lfw_people.data, lfw_people.target, lfw_people.target_names, (50, 37)

# 加载数据
X, y, names, shape = load_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42, stratify=y)

# 训练 1D-PCA
pca = PCA1D(n_components=100)
pca.fit(X_train)

# ========== 图 1: 1D-PCA 特征向量 ==========
fig, axes = plt.subplots(4, 5, figsize=(16, 12))
fig.suptitle('1D-PCA 特征向量（特征脸）', fontsize=18, fontweight='bold')

# 显示平均脸
axes[0, 0].imshow(pca.mean_.reshape(shape), cmap='gray')
axes[0, 0].set_title('平均脸', fontsize=12, color='red', weight='bold')
axes[0, 0].axis('off')

```

```
# 显示说明
axes[0, 1].axis('off')
axes[0, 1].text(0.1, 0.5,
    f'特征向量矩阵形状:\n{pca.components_.shape}\n\n'
    f'每个特征向量:\n{pca.components_.shape[0]}维\n\n'
    f'含义: 人脸变化的主要方向',
    fontsize=11, va='center',
    bbox=dict(boxstyle='round', facecolor='yellow', alpha=0.6))

# 显示前 18 个特征向量
for i in range(18):
    row = (i + 2) // 5
    col = (i + 2) % 5
    eigenface = pca.components_[:, i].reshape(shape)
    axes[row, col].imshow(eigenface, cmap='RdBu_r')
    var = pca.explained_variance_ratio_[i] * 100
    axes[row, col].set_title(f'#{i+1} 方差{var:.1f}%',
fontsize=9)
    axes[row, col].axis('off')

plt.tight_layout()
plt.savefig('1D 特征向量展示.png', dpi=300, bbox_inches='tight')
print("✓ 已保存: 1D 特征向量展示.png")
plt.show()

# ========== 图 2: 1D-PCA 投影向量 ==========
fig = plt.figure(figsize=(18, 10))
fig.suptitle('1D-PCA 投影向量（降维后的特征表示）', fontsize=18,
fontweight='bold')

samples = [0, 50, 100]
for idx, s in enumerate(samples):
    # 原图
    ax1 = plt.subplot(3, 6, idx*6 + 1)
    ax1.imshow(X_train[s].reshape(shape), cmap='gray')
    ax1.set_title(f'{names[y_train[s]]}', fontsize=10,
weight='bold')
    ax1.axis('off')

    # 投影向量
    proj = pca.transform(X_train[s:s+1])[0]
    ax2 = plt.subplot(3, 6, (idx*6 + 2, idx*6 + 3))
    colors = plt.cm.rainbow(np.linspace(0, 1, 10))
```

```
    bars = ax2.bar(range(10), proj[:10], color=colors, edgecolor='black', linewidth=1.5)
    ax2.set_xlabel('特征向量索引', fontsize=10)
    ax2.set_ylabel('投影系数', fontsize=10)
    ax2.set_title(f'投影向量 前10个系数 共{len(proj)}维', fontsize=10)
    ax2.grid(axis='y', alpha=0.3)
    ax2.axhline(0, color='black', linewidth=1)
    for bar, val in zip(bars, proj[:10]):
        h = bar.get_height()
        ax2.text(bar.get_x() + bar.get_width()/2, h, f'{val:.0f}',
                 ha='center', va='bottom' if val>0 else 'top', fontsize=8)

    # 重构图像
    for j, n in enumerate([5, 20, 100]):
        ax3 = plt.subplot(3, 6, idx*6 + 4 + j)
        recon = np.dot(proj[:n], pca.components_[:, :n].T) + pca.mean_
        ax3.imshow(recon.reshape(shape), cmap='gray')
        ax3.set_title(f'重构 {n}成分', fontsize=9)
        ax3.axis('off')

plt.tight_layout()
plt.savefig('1D投影向量展示.png', dpi=300, bbox_inches='tight')
print("✓ 已保存: 1D投影向量展示.png")
plt.show()

# ========== 图3: 投影过程详解 ==========
fig = plt.figure(figsize=(18, 7))
fig.suptitle('1D-PCA 投影过程详细步骤', fontsize=18, fontweight='bold')

s = 0
original = X_train[s]
proj = pca.transform(original.reshape(1, -1))[0]

# 步骤1: 原图
ax1 = plt.subplot(2, 6, 1)
ax1.imshow(original.reshape(shape), cmap='gray')
ax1.set_title(f'步骤1\n原始图像\n{names[y_train[s]]}\n维度 {original.shape[0]}', fontsize=10, weight='bold')
```

```
ax1.axis('off')

# 步骤 2: 平均脸
ax2 = plt.subplot(2, 6, 2)
ax2.imshow(pca.mean_.reshape(shape), cmap='gray')
ax2.set_title('步骤 2\n 平均脸', fontsize=10, weight='bold')
ax2.axis('off')

# 步骤 3: 中心化
ax3 = plt.subplot(2, 6, 3)
centered = (original - pca.mean_).reshape(shape)
ax3.imshow(centered, cmap='RdBu_r')
ax3.set_title('步骤 3\n 中心化\n 原图减平均', fontsize=10,
weight='bold')
ax3.axis('off')

# 步骤 4-5: 特征向量
for i in range(3):
    ax = plt.subplot(2, 6, 4+i)
    eigenface = pca.components_[:, i].reshape(shape)
    ax.imshow(eigenface, cmap='RdBu_r')
    ax.set_title(f'步骤 4\n 特征向量{i+1}', fontsize=10,
weight='bold')
    ax.axis('off')

# 计算说明
ax_calc = plt.subplot(2, 6, 7)
ax_calc.axis('off')
ax_calc.text(0.1, 0.5,
    f'步骤 5 投影计算\n\n'
    f'投影系数 = 中心化图像 点乘 特征向量\n\n'
    f'系数 1 = {proj[0]:.1f}\n'
    f'系数 2 = {proj[1]:.1f}\n'
    f'系数 3 = {proj[2]:.1f}\n'
    f'...\n'
    f'共{len(proj)}个系数',
    fontsize=10, va='center',
    bbox=dict(boxstyle='round', facecolor='lightyellow',
alpha=0.8))

# 投影向量可视化
ax_proj = plt.subplot(2, 6, (8, 12))
n = 30
```

```
colors = plt.cm.viridis(np.linspace(0, 1, n))
ax_proj.barh(range(n), proj[:n], color=colors, edgecolor='black')
ax_proj.set_xlabel('投影系数值', fontsize=11)
ax_proj.set_ylabel('特征向量索引', fontsize=11)
ax_proj.set_title(f'步骤 6 投影向量\n 前{n}个系数', fontsize=12,
weight='bold')
ax_proj.invert_yaxis()
ax_proj.grid(axis='x', alpha=0.3)
ax_proj.axvline(0, color='black', linewidth=1)

plt.tight_layout()
plt.savefig('1D 投影过程详解.png', dpi=300, bbox_inches='tight')
print("✓ 已保存: 1D 投影过程详解.png")
plt.show()

print("\n1D-PCA 可视化完成!")
```

附录 2　2D-PCA 特征向量和投影向量展示

```
"""
2D-PCA 特征向量和投影向量展示（修复中文显示）
"""
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from pca_2d import PCA2D

plt.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei']
plt.rcParams['axes.unicode_minus'] = False

def load_data():
    lfw_people = fetch_lfw_people(min_faces_per_person=70,
resize=0.4)
    return lfw_people.data, lfw_people.target,
lfw_people.target_names, (50, 37)

# 加载数据
X, y, names, shape = load_data()
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=42, stratify=y)

# 训练 2D-PCA
```

```
pca = PCA2D(n_components=30)
pca.fit(X_train, shape)

# ========== 图 1: 2D-PCA 特征向量 ==========
fig, axes = plt.subplots(3, 4, figsize=(16, 10))
fig.suptitle('2D-PCA 特征向量（列向量）', fontsize=18, fontweight='bold')

# 显示平均图像
axes[0, 0].imshow(pca.mean_, cmap='gray')
axes[0, 0].set_title('平均脸矩阵', fontsize=12, color='red', weight='bold')
axes[0, 0].axis('off')

# 显示说明
axes[0, 1].axis('off')
axes[0, 1].text(0.1, 0.5,
    f'特征向量矩阵形状:\n{pca.components_.shape}\n\n'
    f'每个特征向量:\n{pca.components_.shape[0]}维\n'
    f'(图像宽度)\n\n'
    f'含义: 列方向的主要模式',
    fontsize=11, va='center',
    bbox=dict(boxstyle='round', facecolor='lightgreen', alpha=0.7))

# 显示前 10 个特征向量
for i in range(10):
    row = (i + 2) // 4
    col = (i + 2) % 4
    if row < 3:
        eigenvec = pca.components_[:, i]
        axes[row, col].plot(eigenvec, linewidth=3, color=f'C{i}', marker='o', markersize=3)
        axes[row, col].fill_between(range(len(eigenvec)), eigenvec, alpha=0.3, color=f'C{i}')
        axes[row, col].axhline(0, color='black', linewidth=0.8, linestyle='--')
        axes[row, col].grid(True, alpha=0.3)
        var = pca.explained_variance_ratio_[i] * 100
        axes[row, col].set_title(f'特征向量{i+1} 方差{var:.1f}%', fontsize=10)
        axes[row, col].set_xlabel('列索引', fontsize=9)
        axes[row, col].set_ylabel('值', fontsize=9)
```

```
plt.tight_layout()
plt.savefig('2D 特征向量展示.png', dpi=300, bbox_inches='tight')
print("✓ 已保存: 2D 特征向量展示.png")
plt.show()

# ========== 图 2: 2D-PCA 投影向量 ==========
fig = plt.figure(figsize=(18, 10))
fig.suptitle('2D-PCA 投影向量（降维后的特征表示）', fontsize=18, fontweight='bold')

samples = [0, 50, 100]
for idx, s in enumerate(samples):
    # 原图
    ax1 = plt.subplot(3, 6, idx*6 + 1)
    ax1.imshow(X_train[s].reshape(shape), cmap='gray')
    ax1.set_title(f'{names[y_train[s]]}', fontsize=10, weight='bold')
    ax1.axis('off')

    # 投影向量（展平后）
    proj = pca.transform(X_train[s:s+1])[0]  # shape: (50*30,)
    proj_matrix = proj.reshape(shape[0], 30)  # 重塑为矩阵形式 (50, 30)

    # 显示投影矩阵
    ax2 = plt.subplot(3, 6, (idx*6 + 2, idx*6 + 3))
    im = ax2.imshow(proj_matrix, cmap='RdYlBu', aspect='auto')
    ax2.set_xlabel('主成分索引', fontsize=10)
    ax2.set_ylabel('行索引', fontsize=10)
    ax2.set_title(f'投影矩阵 {proj_matrix.shape}\n 共{len(proj)}维', fontsize=10)
    plt.colorbar(im, ax=ax2)

    # 重构图像
    for j, n in enumerate([5, 15, 30]):
        ax3 = plt.subplot(3, 6, idx*6 + 4 + j)
        # 使用前 n 个主成分重构
        proj_partial = proj_matrix[:, :n]
        comp_partial = pca.components_[:, :n]
        recon_matrix = np.dot(proj_partial, comp_partial.T) + pca.mean_
        ax3.imshow(recon_matrix, cmap='gray')
```

```
        ax3.set_title(f'重构 {n}成分', fontsize=9)
        ax3.axis('off')

plt.tight_layout()
plt.savefig('2D 投影向量展示.png', dpi=300, bbox_inches='tight')
print("✓ 已保存: 2D 投影向量展示.png")
plt.show()

# ========== 图 3: 2D 投影过程详解 ==========
fig = plt.figure(figsize=(18, 8))
fig.suptitle('2D-PCA 投影过程详细步骤', fontsize=18,
fontweight='bold')

s = 0
original = X_train[s].reshape(shape)
proj = pca.transform(X_train[s:s+1])[0]
proj_matrix = proj.reshape(shape[0], 30)

# 步骤 1: 原图矩阵
ax1 = plt.subplot(2, 5, 1)
ax1.imshow(original, cmap='gray')
ax1.set_title(f'步骤 1\n 原始图像矩阵\n{names[y_train[s]]}\n 形状
{shape}', fontsize=10, weight='bold')
ax1.axis('off')

# 步骤 2: 平均矩阵
ax2 = plt.subplot(2, 5, 2)
ax2.imshow(pca.mean_, cmap='gray')
ax2.set_title(f'步骤 2\n 平均矩阵\n 形状{pca.mean_.shape}',
fontsize=10, weight='bold')
ax2.axis('off')

# 步骤 3: 中心化矩阵
ax3 = plt.subplot(2, 5, 3)
centered = original - pca.mean_
ax3.imshow(centered, cmap='RdBu_r')
ax3.set_title(f'步骤 3\n 中心化矩阵\n 原图减平均', fontsize=10,
weight='bold')
ax3.axis('off')

# 步骤 4: 特征向量
ax4 = plt.subplot(2, 5, 4)
eigenvec1 = pca.components_[:, 0]
```

```
ax4.plot(eigenvec1, linewidth=4, color='darkgreen', marker='o', markersize=4)
ax4.fill_between(range(len(eigenvec1)), eigenvec1, alpha=0.3, color='green')
ax4.axhline(0, color='black', linewidth=1, linestyle='--')
ax4.grid(True, alpha=0.3)
ax4.set_title('步骤 4\n 第 1 个特征向量\n37 维列向量', fontsize=10, weight='bold')
ax4.set_xlabel('维度', fontsize=9)

# 计算说明
ax_calc = plt.subplot(2, 5, 5)
ax_calc.axis('off')
ax_calc.text(0.1, 0.5,
    f'步骤 5 投影计算\n\n'
    f'投影矩阵 = 中心化矩阵 × 特征向量矩阵\n\n'
    f'Y = A × X\n'
    f'{shape} × {pca.components_[:,:5].shape}\n'
    f'= {proj_matrix[:,:5].shape}\n\n'
    f'展平后维度: {len(proj)}',
    fontsize=10, va='center',
    bbox=dict(boxstyle='round', facecolor='lightyellow', alpha=0.8))

# 投影矩阵可视化
ax_proj = plt.subplot(2, 5, (6, 10))
im = ax_proj.imshow(proj_matrix, cmap='RdYlBu', aspect='auto')
ax_proj.set_xlabel('主成分索引 (30 个)', fontsize=11)
ax_proj.set_ylabel('行索引 (50 行)', fontsize=11)
ax_proj.set_title(f'步骤 6 投影矩阵\n 形状 {proj_matrix.shape}', fontsize=12, weight='bold')
plt.colorbar(im, ax=ax_proj, label='投影系数值')

plt.tight_layout()
plt.savefig('2D 投影过程详解.png', dpi=300, bbox_inches='tight')
print("✓ 已保存: 2D 投影过程详解.png")
plt.show()

print("\n2D-PCA 可视化完成!")
```

附录 3　基于傅里叶描述子的数字识别系统

```
"""
```

```
2. 基于傅里叶描述子的数字识别系统
3. 支持 MNIST 手写数字识别
4. """
5.
6. import numpy as np
7. import matplotlib.pyplot as plt
8. from sklearn.datasets import load_digits
9. from sklearn.model_selection import train_test_split
10. from sklearn.neighbors import KNeighborsClassifier
11. from sklearn.svm import SVC
12. from sklearn.metrics import classification_report,
    confusion_matrix, accuracy_score
13. import seaborn as sns
14. from fourier_descriptor import FourierDescriptor,
    compute_distance
15. import cv2
16.
17. # 设置中文字体支持
18. plt.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei',
    'Arial Unicode MS']
19. plt.rcParams['axes.unicode_minus'] = False
20.
21. class DigitRecognizer:
22.     """基于傅里叶描述子的数字识别器"""
23.
24.     def __init__(self, n_descriptors=30, classifier_type='knn'):
25.         """
26.         初始化识别器
27.
28.         Args:
29.             n_descriptors: 傅里叶描述子数量
30.             classifier_type: 分类器类型 ('knn', 'svm')
31.         """
32.         self.fd = FourierDescriptor(n_descriptors=n_descriptors)
33.         self.classifier_type = classifier_type
34.
35.         if classifier_type == 'knn':
36.             self.classifier = KNeighborsClassifier(n_neighbors=5,
    metric='euclidean')
37.         elif classifier_type == 'svm':
38.             self.classifier = SVC(kernel='rbf', C=10,
    gamma='scale')
```

```
        else:
            raise ValueError(f"Unknown classifier type: {classifier_type}")

        self.is_trained = False

    def extract_features_batch(self, images):
        """
        批量提取特征

        Args:
            images: 图像数组 (n_samples, height, width)

        Returns:
            特征矩阵 (n_samples, n_descriptors)
        """
        n_samples = len(images)
        features = np.zeros((n_samples, self.fd.n_descriptors))

        for i in range(n_samples):
            features[i] = self.fd.extract_features(images[i])

            if (i + 1) % 100 == 0:
                print(f"  已处理 {i + 1}/{n_samples} 张图像")

        return features

    def train(self, X_train, y_train):
        """
        训练识别器

        Args:
            X_train: 训练图像
            y_train: 训练标签
        """
        print(f"正在提取训练集特征（共{len(X_train)}张图像）...")
        X_features = self.extract_features_batch(X_train)

        print(f"\n 正在训练{self.classifier_type.upper()}分类器...")
        self.classifier.fit(X_features, y_train)

        self.is_trained = True
```

```
        print("训练完成！")

    def predict(self, X_test):
        """
        预测数字

        Args:
            X_test: 测试图像

        Returns:
            预测标签
        """
        if not self.is_trained:
            raise RuntimeError("模型未训练，请先调用 train()方法")

        print(f"正在提取测试集特征（共{len(X_test)}张图像）...")
        X_features = self.extract_features_batch(X_test)

        print("正在预测...")
        predictions = self.classifier.predict(X_features)

        return predictions

    def evaluate(self, X_test, y_test):
        """
        评估模型性能

        Args:
            X_test: 测试图像
            y_test: 测试标签

        Returns:
            准确率
        """
        y_pred = self.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)

        return accuracy, y_pred


def load_mnist_data():
    """加载 MNIST 数字数据集（sklearn 内置版本）"""
    print("正在加载数字数据集...")
```

```
    # 使用 sklearn 内置的 8x8 数字数据集
    digits = load_digits()

    # 获取图像和标签
    images = digits.images  # (n_samples, 8, 8)
    labels = digits.target

    # 放大图像以便更好地提取轮廓
    enlarged_images = []
    for img in images:
        # 归一化到 0-255
        img_normalized = (img / 16.0 * 255).astype(np.uint8)
        # 放大到 28x28
        enlarged = cv2.resize(img_normalized, (28, 28), interpolation=cv2.INTER_LINEAR)
        enlarged_images.append(enlarged)

    enlarged_images = np.array(enlarged_images)

    print(f"数据集大小: {enlarged_images.shape}")
    print(f"类别数量: {len(np.unique(labels))}")
    print(f"类别: {np.unique(labels)}")

    return enlarged_images, labels


def visualize_samples(images, labels, n_samples=10):
    """可视化样本"""
    print("\n 正在生成样本展示...")

    fig, axes = plt.subplots(2, 5, figsize=(12, 6))
    fig.suptitle('数字样本展示', fontsize=16, fontweight='bold')

    for i in range(n_samples):
        row = i // 5
        col = i % 5

        axes[row, col].imshow(images[i], cmap='gray')
        axes[row, col].set_title(f'数字: {labels[i]}', fontsize=12)
        axes[row, col].axis('off')

```

```
    plt.tight_layout()
    plt.savefig('digit_samples.png', dpi=300, bbox_inches='tight')
    plt.show()

    print("样本展示已保存: digit_samples.png")


def visualize_fourier_features(recognizer, images, labels):
    """可视化不同数字的傅里叶描述子"""
    print("\n 正在生成傅里叶描述子可视化...")

    fig, axes = plt.subplots(2, 5, figsize=(15, 6))
    fig.suptitle('不同数字的傅里叶描述子特征', fontsize=16, fontweight='bold')

    for digit in range(10):
        # 找到该数字的第一个样本
        idx = np.where(labels == digit)[0][0]
        image = images[idx]

        # 提取特征
        features = recognizer.fd.extract_features(image)

        row = digit // 5
        col = digit % 5

        axes[row, col].stem(range(len(features)), features, basefmt=' ')
        axes[row, col].set_title(f'数字 {digit}', fontsize=11)
        axes[row, col].set_xlabel('频率索引', fontsize=9)
        axes[row, col].set_ylabel('幅度', fontsize=9)
        axes[row, col].grid(True, alpha=0.3)
        axes[row, col].set_ylim([0, 1.1])

    plt.tight_layout()
    plt.savefig('fourier_features_comparison.png', dpi=300, bbox_inches='tight')
    plt.show()

    print("特征对比已保存: fourier_features_comparison.png")

```

```
def plot_confusion_matrix(y_test, y_pred):
    """绘制混淆矩阵"""
    print("\n 正在生成混淆矩阵...")

    cm = confusion_matrix(y_test, y_pred)

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=range(10), yticklabels=range(10),
                cbar_kws={'label': '数量'})
    plt.title('数字识别混淆矩阵', fontsize=16, fontweight='bold')
    plt.xlabel('预测标签', fontsize=12)
    plt.ylabel('真实标签', fontsize=12)
    plt.tight_layout()
    plt.savefig('confusion_matrix.png', dpi=300,
bbox_inches='tight')
    plt.show()

    print("混淆矩阵已保存: confusion_matrix.png")


def plot_recognition_examples(recognizer, X_test, y_test, y_pred,
n_examples=10):
    """展示识别示例"""
    print("\n 正在生成识别示例...")

    # 找出正确和错误的预测
    correct_idx = np.where(y_pred == y_test)[0]
    wrong_idx = np.where(y_pred != y_test)[0]

    n_correct = min(5, len(correct_idx))
    n_wrong = min(5, len(wrong_idx))

    fig, axes = plt.subplots(2, 5, figsize=(14, 6))
    fig.suptitle('识别示例（上：正确，下：错误）', fontsize=16,
fontweight='bold')

    # 正确的示例
    for i in range(n_correct):
        idx = correct_idx[i]
        axes[0, i].imshow(X_test[idx], cmap='gray')
        axes[0, i].set_title(f'真实:{y_test[idx]}\n 预
测:{y_pred[idx]}',
```

```
                                fontsize=10, color='green')
            axes[0, i].axis('off')

        # 填充空白
        for i in range(n_correct, 5):
            axes[0, i].axis('off')

        # 错误的示例
        for i in range(n_wrong):
            idx = wrong_idx[i]
            axes[1, i].imshow(X_test[idx], cmap='gray')
            axes[1, i].set_title(f'真实:{y_test[idx]}\n预测:{y_pred[idx]}',
                                fontsize=10, color='red')
            axes[1, i].axis('off')

        # 填充空白
        for i in range(n_wrong, 5):
            axes[1, i].axis('off')

        plt.tight_layout()
        plt.savefig('recognition_examples.png', dpi=300, bbox_inches='tight')
        plt.show()

        print("识别示例已保存: recognition_examples.png")


def main():
    """主函数"""
    print("=" * 70)
    print("基于傅里叶描述子的数字识别实验")
    print("=" * 70)

    # 加载数据
    images, labels = load_mnist_data()

    # 可视化样本
    visualize_samples(images, labels)

    # 划分训练集和测试集
    X_train, X_test, y_train, y_test = train_test_split(
```

```
        images, labels, test_size=0.3, random_state=42, stratify=labels
    )

    print(f"\n 训练集大小: {len(X_train)}")
    print(f"测试集大小: {len(X_test)}")

    # 创建识别器（使用 KNN）
    print("\n" + "=" * 70)
    print("使用 K 近邻分类器")
    print("=" * 70)

    recognizer_knn = DigitRecognizer(n_descriptors=30, classifier_type='knn')

    # 训练模型
    recognizer_knn.train(X_train, y_train)

    # 可视化傅里叶特征
    visualize_fourier_features(recognizer_knn, images, labels)

    # 评估模型
    accuracy_knn, y_pred_knn = recognizer_knn.evaluate(X_test, y_test)

    print(f"\n{'=' * 70}")
    print(f"K 近邻分类器准确率: {accuracy_knn:.4f} ({accuracy_knn*100:.2f}%)")
    print("=" * 70)

    # 详细分类报告
    print("\n 分类报告:")
    print(classification_report(y_test, y_pred_knn,
                                target_names=[str(i) for i in range(10)]))

    # 绘制混淆矩阵
    plot_confusion_matrix(y_test, y_pred_knn)

    # 展示识别示例
    plot_recognition_examples(recognizer_knn, X_test, y_test, y_pred_knn)

```

```
    # 使用 SVM 分类器
    print("\n" + "=" * 70)
    print("使用支持向量机分类器")
    print("=" * 70)

    recognizer_svm = DigitRecognizer(n_descriptors=30, classifier_type='svm')
    recognizer_svm.train(X_train, y_train)

    accuracy_svm, y_pred_svm = recognizer_svm.evaluate(X_test, y_test)

    print(f"\n{'=' * 70}")
    print(f"支持向量机准确率: {accuracy_svm:.4f} ({accuracy_svm*100:.2f}%)")
    print("=" * 70)

    # 对比结果
    print("\n" + "=" * 70)
    print("分类器性能对比")
    print("=" * 70)
    print(f"K 近邻 (KNN):      {accuracy_knn:.4f} ({accuracy_knn*100:.2f}%)")
    print(f"支持向量机 (SVM): {accuracy_svm:.4f} ({accuracy_svm*100:.2f}%)")

    if accuracy_knn > accuracy_svm:
        print("\n最佳分类器: K 近邻 (KNN)")
    else:
        print("\n最佳分类器: 支持向量机 (SVM)")

    print("\n" + "=" * 70)
    print("实验完成！")
    print("=" * 70)

    print("\n生成的文件:")
    print("  - digit_samples.png (数字样本)")
    print("  - fourier_features_comparison.png (傅里叶特征对比)")
    print("  - confusion_matrix.png (混淆矩阵)")
    print("  - recognition_examples.png (识别示例)")
if __name__ == "__main__":
    main()
```