# 目录

[1 实验背景 2](#_Toc6855)

[2 实验目标 2](#_Toc4080)

[2.1 任务一 2](#_Toc21418)

[2.2 任务二 2](#_Toc26421)

[2.3 任务三 2](#_Toc17433)

[3 理论介绍 3](#_Toc3023)

[4 实验过程与结果 3](#_Toc27318)

[4.1 基于 K-L 变换和 MLP 的字符识别 3](#_Toc30808)

[4.2 基于多层感知机的字符识别 5](#_Toc11037)

[4.3 基于卷积神经网络的字符识别 6](#_Toc12221)

[5 实验结论 7](#_Toc15344)

[附录：关键代码注释 8](#_Toc16856)

# 字符识别

## 1 实验背景

字符识别是计算机视觉领域的一项核心任务。本实验旨在比较和研究两种字符识别方法：基于 K-L 变换与多层感知机（MLP）的特征提取与分类方法，以及基于卷积神经网络（CNN）的端到端识别方法。通过实验，我们希望深入分析不同方法在字符识别任务中的性能表现和适用场景，从而为相关应用提供参考。

本次实验采用 Chars74K 系列数据集作为数据来源。Chars74K 涵盖英文大写字母（不包含字母“I”和“O”）和数字 0~9 的图像，每张图像尺寸为 28×28 像素，字符为白色，背景为黑色。数据集分为训练集和测试集，其中训练集包含 74,700 张图像，测试集则包含 6,294 张独立图像。

## 2 实验目标

### 2.1 任务一：**基于 K-L 变换和 MLP 的字符识别**

• 特征提取： 利用 K-L 变换从原始图像中提取高效的特征向量。

• 分类方法： 使用 MLP 进行字符分类。

### 2.2 任务二：**基于多层感知机的字符识别**

• 特征提取： 利用多层感知机（MLP）自动学习图像特征。

• 分类方法： 使用 MLP 进行字符分类。

### 2.3 任务三：**基于卷积神经网络的字符识别**

• 特征提取： 利用卷积神经网络（CNN）自动学习图像特征。

• 分类方法： 使用 CNN 进行字符分类。

### 3 理论介绍

**3.1 K-L 变换**

K-L 变换是一种特征提取技术，通过线性变换将数据映射到新的坐标空间，从而实现数据方差的最大化。在字符识别任务中，K-L 变换能够将原始图像转化为更为紧凑的特征表示，这显著降低了训练过程中的计算复杂度和成本。

**3.2 多层感知机（MLP）**

多层感知机（MLP）是一种由多个层次组成的前馈神经网络，包括输入层、一个或多个隐藏层以及输出层。网络中的每个神经元与上一层的所有神经元相连接，并通过权重和激活函数进行处理和传递信息。通过训练过程，网络能够自主地学习到字符的潜在特征表达，从而实现准确的识别。

**3.3 卷积神经网络（CNN）**

卷积神经网络（CNN）是一种具有深度架构并涉及卷积运算的前馈神经网络，是深度学习领域的重要算法之一。卷积神经网络具备自动学习数据特征的能力，能够对输入信息进行平移不变的分类，因此也常被称为“平移不变的人工神经网络”。

### 4 实验过程与结果

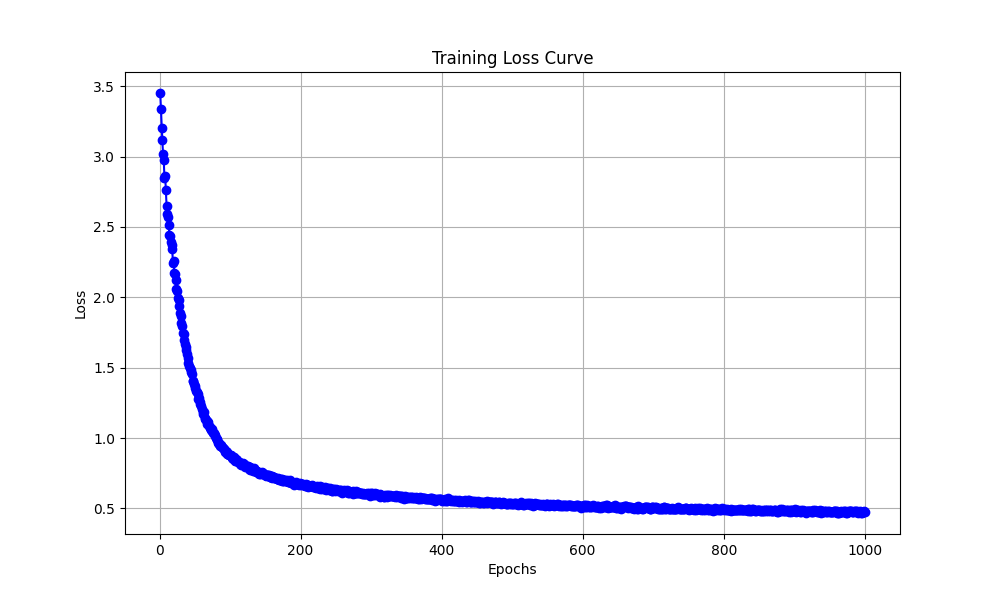
### 4.1 基于 K-L 变换和 MLP 的字符识别

首先，对训练集的图像构造去均值样本矩阵，随后计算协方差矩阵，并计算协方差矩阵的特征值和特征向量，根据预先设定的重构精确度选取累计贡献率达到要求的前大的特征向量，对图像进行降维处理。（代码见附录）

实验设计网络结构为：

1. **class MLPModel(nn.Module):**
2. **def \_\_init\_\_(self, input\_dim, num\_classes):**
3. **super(MLPModel, self).\_\_init\_\_()**
4. **self.fc1 = nn.Linear(input\_dim, 128)**
5. **self.fc2 = nn.Linear(128, 256)**
6. **self.fc3 = nn.Linear(256, 516)**
7. **self.fc4 = nn.Linear(516, 1024)**
8. **self.fc5 = nn.Linear(1024, 512)**
9. **self.fc6 = nn.Linear(512, 256)**
10. **self.fc7 = nn.Linear(256, 128)**
11. **self.fc8 = nn.Linear(128, num\_classes)**
12. **self.relu = nn.ReLU(True)**
13. **self.dropout = nn.Dropout(0.5)**

实验细节：设定重构精确度，重构后的特征向量维数以变量的方式传入模型进行初始化，选择优化器为 Adadelta[[1]](#footnote-0)，损失函数为 CrossEntropyLoss[[2]](#footnote-1)，设置 batch\_size = 8192, 训练 1000 epochs（训练具体环境见附录），训练过程损失函数如下所示：



随着训练次数的增加，训练损失和测试损失逐渐下降，并最终收敛到一个稳定的值，表明模型已达到收敛状态，因此可以适当减少训练的 epoch 数量。选取最后一个回合的模型进行测试，结果为：

1. **Using device: cuda**
2. **Evaluating the model...**
3. **Test Accuracy: 92.72%**

可见使用 K-L 降维之后可以在降低特征向量维度的同时基本保留特征向量的所有信息，这样可以提高训练的效率，节省时间。但是效果比不上不使用降维的原始数据训练的结果。

创新点：由于对于设定的重构精确度和确定的训练数据，降维后的图片特征和均值等均是固定的，因此不必每次实验都计算一次，可以使用缓存的方法将计算好的相关信息存到本地，后续实验可以直接使用，从而提高效率。相关代码为：

1. **if \_\_name\_\_ == "\_\_main\_\_":**
2. **# 尝试从缓存加载数据**
3. **e = 0.99  # 重构精确度**
4. **eigenfaces, mean, labels, coordinates = load\_from\_cache(e)**
5. **if eigenfaces is None:**
6. **train\_path = "./dataset/train.txt"**
7. **print("构造样本矩阵……")**
8. **X, mean, labels = getX(train\_path)**
9. **print("构造本征变换矩阵……")**
10. **eigenfaces = getEigenfaces(X, e)**
11. **print("可视化本征字符……")**
12. **drawEigenfaces(eigenfaces)  # 绘制可视化本征向量**
13. **print("计算训练集图像在特征脸空间中的线性表示……")**
14. **coordinates = np.dot(X, eigenfaces.T)**
15. **# 保存计算结果到缓存**
16. **save\_to\_cache(e, eigenfaces, mean, labels, coordinates)**

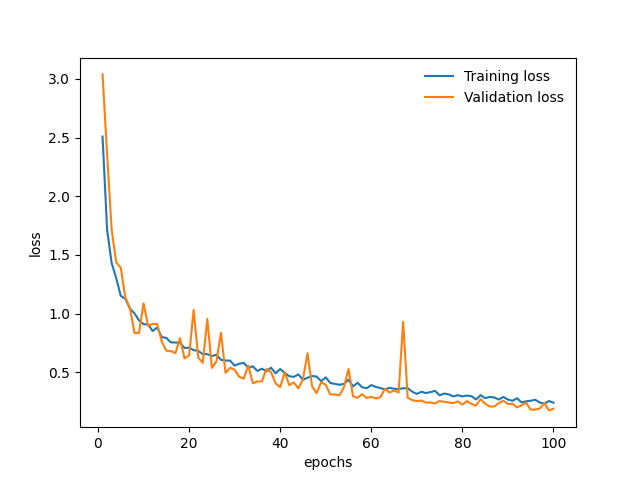
### 4.2 基于多层感知机的字符识别

在这个实验里，不使用 K-L 等方法进行数据降维，而是直接使用原始图片进行训练，以对比效果。

网络设置为：

1. **class Classification(nn.Module):**
2. **# 定义构造函数**
3. **def \_\_init\_\_(self, input\_size=784, hidden\_sizes=[256, 128], dropout\_rate=0.5):**
4. **super(Classification, self).\_\_init\_\_()**
5. **# 定义网络结构**
6. **self.output = nn.Sequential(**
7. **nn.Flatten(),  # 1x28x28 -> 1x784**
8. **nn.Linear(input\_size, hidden\_sizes[0]),  # 1x784 -> hidden\_sizes[0]**
9. **nn.BatchNorm1d(hidden\_sizes[0]),  # 批归一化**
10. **nn.ReLU(True),  # 激活函数**
11. **nn.Linear(hidden\_sizes[0], hidden\_sizes[1]),**
12. **nn.BatchNorm1d(hidden\_sizes[1]),  # 批归一化**
13. **nn.ReLU(True),  # 激活函数**
14. **nn.Dropout(dropout\_rate),  # 随机失活**
15. **nn.Linear(hidden\_sizes[1], number\_classes),**
16. **)**

训练细节：选择优化器为 Adadelta，损失函数为 CrossEntropyLoss，类别数量为 34，设置 batch\_size = 10240, 训练 100 epochs，训练过程损失函数如下所示：



可以观察到，在训练的初期，损失函数（loss）下降较为显著，表明模型在初始阶段能够快速地拟合数据，学习到较为有效的特征。在训练的中期，验证集的损失出现较大幅度的波动，这通常意味着模型仍处于优化过程中，可能在局部最优解附近徘徊，尚未找到最佳的参数配置。此时，模型对数据的拟合度仍有提升空间，可能存在过拟合或欠拟合的风险。到了训练的后期，损失值趋于平稳，基本不再发生显著变化，表明模型已经收敛，进一步的训练对损失的改善效果较小，模型的学习效果已经接近最优。因此，训练基本完成，损失的稳定性标志着模型的训练过程已经达到收敛。

训练结果：

1. **测试图像数量: 6294, 误分类数量: 207, 分类准确率: 96.71%**

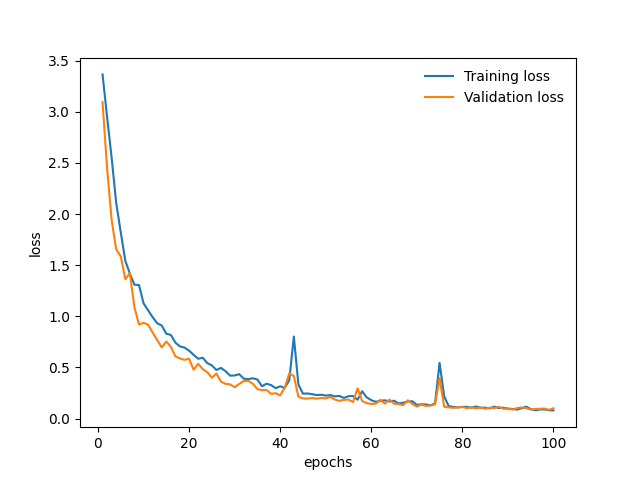
可以看出，使用原始图片进行训练所需的 epoch 数量显著少于使用处理后图像的情况，并且最终能够达到更好的效果。这表明原始图片包含了更多的有效信息，能够更快地帮助模型学习到数据的关键特征。

### 4.3 基于卷积神经网络的字符识别

网络结构：

1. **class ClassificationModel(nn.Module):**
2. **# 定义构造函数**
3. **def \_\_init\_\_(self):**
4. **super(ClassificationModel, self).\_\_init\_\_()**
5. **# 定义网络结构**
6. **self.output = nn.Sequential(**
7. **nn.Conv2d(1, 32, kernel\_size=3, stride=1, padding=1),  # 1x28x28 -> 32x28x28**
8. **nn.ReLU(True),**
9. **nn.Conv2d(32, 64, kernel\_size=3, stride=1, padding=1),  # 32x28x28 -> 64x28x28**
10. **nn.ReLU(True),**
11. **nn.MaxPool2d(kernel\_size=2, stride=2),  # 64x28x28 -> 64x14x14**
12. **nn.Flatten(),  # 64x14x14 -> 12544**
13. **nn.Linear(64 \* 14 \* 14, 128),  # 12544 -> 128**
14. **nn.ReLU(True),**
15. **nn.Dropout(0.3),  # 以默认的概率 0.3 随机地将输入张量中的一些元素设置为 0**
16. **nn.Linear(128, number\_classes),  # 128 -> 34**
17. **)**

训练细节：选择优化器为 Adadelta，损失函数为 CrossEntropyLoss，类别数量为 34，设置 batch\_size = 10240, 训练 100 epochs，训练过程损失函数如下：



### 可以看出，使用 CNN 训练的过程与 MLP 相似，都是通过反向传播和梯度更新来优化模型。然而，CNN 在训练过程中通常更加稳定。这是因为 CNN 通过卷积层和池化层有效地提取了局部特征和空间层次信息，能够更好地捕捉图像中的结构化模式，减少了梯度消失或爆炸的风险。

训练结果：

1. **测试图像数量: 6294, 误分类数量: 189, 分类准确率: 97.00%**

# **5 实验结论**

实验结果表明，对于字符识别的任务，使用原始图像进行训练能够显著减少所需的 epoch 数量，并且在较短时间内达到更好的性能。而使用 K-L 进行图片信息降维后训练可以减少训练过程中的计算复杂度和成本。此外，CNN 网络在训练过程中表现出较 MLP 神经网络更加稳定的收敛性，得益于其强大的局部特征提取能力和空间结构学习能力。这三种方法都能不错地完成字符识别任务。

### 附录：

1. **实验环境与配置**
2. **- Platform: Linux-5.4.0-150-generic-x86\_64-with-glibc2.27**
3. **- Python version: 3.10.16**
4. **- Huggingface\_hub version: 0.27.0**
5. **- Safetensors version: 0.4.5**
6. **- Accelerate version: not installed**
7. **- Accelerate config: not found**
8. **- PyTorch version (GPU?): 2.5.1+cu124 (True)**
9. **- Tensorflow version (GPU?): not installed (NA)**
10. **- Flax version (CPU?/GPU?/TPU?): not installed (NA)**
11. **- Jax version: not installed**
12. **- JaxLib version: not installed**
13. **基于 K-L 变换和 MLP 的字符识别**

K-L变换部分的实现细节如下

1. **def getX(train\_path: str) -> tuple[np.ndarray, np.ndarray, np.ndarray]:**
2. **"""**
3. **构造去均值样本矩阵 X (m, w\*h), m 为样本数，w、h 为样本宽度和高度。**
4. **参数:**
5. **train\_path (str): 训练集文件路径**
6. **返回:**
7. **tuple[np.ndarray, np.ndarray, np.ndarray]: 去均值样本矩阵 X, 均值向量 mean, 标签向量 labels**
8. **"""**
9. **image\_paths, labels = load\_image\_paths\_and\_labels(train\_path)**
10. **X: list[int] = []**
11. **for image\_path in tqdm(image\_paths, desc="Reading images"):**
12. **image = np.array(Image.open(image\_path)).flatten()  # 读取图像，转换为灰度图，一维化**
13. **X.append(image)  # 构造图像矩阵**
14. **mean: torch.Tensor = np.mean(X, axis=0)**
15. **X = np.array(X) - mean  # 去均值**
16. **return X, mean, np.array(labels)**
17. **def getEigenfaces(X, e: float) -> np.ndarray:**
18. **"""**
19. **通过 KL 变换获取本征向量**
20. **参数:**
21. **去均值样本矩阵 X (m, w\*h), 重构精确度 e**
22. **返回:**
23. **np.ndarray: 本征向量矩阵，每一行为一个本征向量，维度为 w\*h**
24. **"""**
25. **# 样本矩阵降维：m\*wh \* wh\*m -> m\*m 计算样本矩阵 X 的协方差矩阵 E**
26. **E = np.dot(np.transpose(X), X)**
27. **eigenvalue, featurevector\_E = np.linalg.eig(E)  # 求解特征值和特征向量**
28. **featurevector\_E = np.transpose(featurevector\_E)  # 将特征向量转置一下**
29. **eigen = []**
30. **for i in range(0, eigenvalue.shape[0]):  # 将特征值和特征向量绑到一起，方便排序**
31. **eigen.append([eigenvalue[i], featurevector\_E[i]])**
32. **eigen = sorted(eigen, key=lambda eigen: eigen[0], reverse=True)  # 根据特征值的大小排序**
33. **# 根据重构精确度 e 确定需要的特征脸数量。通过累加特征值，直到累加和占总特征值和的比例达到或超过 e**
34. **eigenfaces\_num, eigenvalue\_sum = 0, 0**
35. **for i in range(0, eigenvalue.shape[0]):**
36. **eigenvalue\_sum += eigen[i][0]**
37. **if eigenvalue\_sum / eigenvalue.sum() >= e:**
38. **eigenfaces\_num = i + 1**
39. **break**
40. **eigenfaces = []  # 提取前 eigenfaces\_num 个特征向量作为特征脸**
41. **for i in range(0, eigenfaces\_num):**
42. **eigenfaces.append(eigen[i][1])**
43. **return np.array(eigenfaces)**
44. **基于卷积神经网络的字符识别**

训练的具体过程，包括实现在GPU上训练，以及训练时记录所需数据，方便后续进行训练过程与结果的分析

1. **print("开始训练……")**
2. **for epoch in range(epochs):**
3. **CC.train()**
4. **loss\_train = []**
5. **for batch\_idx, (images, labels) in enumerate(data\_loader\_train):**
6. **start\_time = time.time()**
7. **images, labels = images.cuda(), labels.cuda()**
8. **CC\_output = CC(images)**
9. **loss: torch.Tensor = lossFun(CC\_output, labels)**
10. **optimizer.zero\_grad()**
11. **loss.backward()**
12. **optimizer.step()**
13. **loss\_data = loss.data.item()**
14. **loss\_train.append(loss\_data)**
15. **batch\_time = time.time() - start\_time**
16. **print(**
17. **"epoch:{}, batch:{}, loss:{:.3f}, time:{:.2f}s".format(**
18. **epoch + 1, batch\_idx + 1, loss\_data, batch\_time**
19. **)**
20. **)**
21. **if (epoch + 1) % 10 == 0:**
22. **os.makedirs("./models", exist\_ok=True)**
23. **torch.save(CC.state\_dict(), "./models/CC\_" + str(epoch + 1) + ".pth")**
24. **torch.save(optimizer.state\_dict(), "./models/CC\_optimizer\_" + str(epoch + 1) + ".pth")**
25. **save\_loss\_plot\_and\_data(**
26. **epochs\_x,**
27. **loss\_epochs\_train,**
28. **loss\_epochs\_test,**
29. **plot\_filename=f"loss\_plot\_epoch\_{epoch + 1}.png",**
30. **data\_filename=f"loss\_data\_epoch\_{epoch + 1}.pkl",**
31. **)**
32. **# 模型测试（每个epoch使用测试集进行一次验证）**
33. **CC.eval()**
34. **loss\_test = []**
35. **for batch\_idx, (images, labels) in enumerate(data\_loader\_test):**
36. **images, labels = images.cuda(), labels.cuda()**
37. **CC\_output = CC(images)**
38. **loss = lossFun(CC\_output, labels)**
39. **loss\_test.append(loss.data.item())**
40. **print("平均损失：", np.mean(loss\_test))**
41. **# 记录损失**
42. **loss\_epochs\_train.append(np.mean(loss\_train))**
43. **loss\_epochs\_test.append(np.mean(loss\_test))**
44. **epochs\_x.append(epoch + 1)**

1. Adadelta 是一种自适应学习率的优化算法，旨在解决传统梯度下降算法中学习率选择和调整的问题。它是对 Adagrad 的改进，通过引入一个滑动窗口来限制累积梯度的大小，从而避免了 Adagrad 在训练过程中学习率过早衰减的问题。 [↑](#footnote-ref-0)
2. CrossEntropyLoss 是一种常用于分类任务中的损失函数，尤其是在多类分类问题中。它是通过计算目标类别的概率分布与模型预测的概率分布之间的差异来评估模型的性能。其核心思想是衡量真实标签的分布与预测分布之间的交叉熵（cross-entropy） [↑](#footnote-ref-1)