
# 特征学习

## 神经网络与深度学习课程实验报告

班级：110

姓名：耿翊中

学号：2021213382

2023 年 12 月

目录

## 1. 实验要求

A7：特征学习

任务 1. PCA 或 kernel PCA (10 points)
使用 PCA 或 kernel PCA 对手写数字数据集 MINST 进行降维。观察前两个特征向量所对应的图像，即将数据嵌入到 R2 空间。绘制降维后的数据，并分析二维特征是否能够足以完成对输入的分类，对结果进行分析和评价。

任务 2. Autoencoder (10 points)
使用自动编码器学习输入的特征表示。尝试设计一个全链接前馈神经网络或卷积神经网络。尝试使用不同的损失函数和正则化方法。

附加题 (BONUS: 10 points)
模型训练中，你可以尝试任何可以提升模型性能的合理的方法。例如其它的网络结构、设计多个隐藏层、引入降噪自动编码器等任何你能想到的方法。计算模型在训练集和测试集上的损失，并对结果进行讨论。

# 2. 任务一：PCA 与 kernel PCA

## 2.1 实验原理

PCA（主成分分析）：PCA 是一种线性降维技术，它通过正交变换将数据投影到一个新的坐标系统中，使得最大方差的数据成为第一个新坐标（第一个主成分），第二大方差的数据成为第二个新坐标，依此类推。

Kernel PCA（核主成分分析）：Kernel PCA 是 PCA 的一种扩展，适用于非线性数据集。它使用核技巧映射数据到高维空间，在这个高维空间中应用线性 PCA。

## 2.2 实验过程

1.数据加载与预处理

代码中首先通过 fetch_openml 函数从 MNIST 数据集中加载前 4000 个样本。

将目标标签（数字）转换为整型。

```
# 加载 MNIST 数据集的一部分
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"][:4000], mnist["target"][:4000]  # 只使用前 3000 个样本
# 将数据类型转换为整型
y = y.astype(np.uint8)
```

2.降维处理

使用 PCA 将数据降至 2 个维度。代码中通过 PCA(n_components=2)实现，然后使用 fit_transform 方法对数据 X 进行转换，得到 X_pca。

对于 Kernel PCA，选择 RBF 核，并设置 gamma 值为 0.1。通过 KernelPCA(n_components=2, kernel="rbf", gamma=0.1)实现，并对数据 X 使用 fit_transform 方法，得到 X_kernel_pca。

```
# 使用 PCA 进行降维
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# 调整 Kernel PCA 参数
kernel_pca = KernelPCA(n_components=2, kernel="rbf", gamma=0.1)  # 调整 gamma 值
X_kernel_pca = kernel_pca.fit_transform(X)
```

3. 结果可视化

利用 matplotlib 绘图。先创建一个大小为 12x6 英寸的图形窗口。

在第一个子图中绘制 PCA 结果，使用 scatter 方法，x 轴为 X_pca[:, 0]，y 轴为 X_pca[:, 1]，颜色根据标签 y 变化。

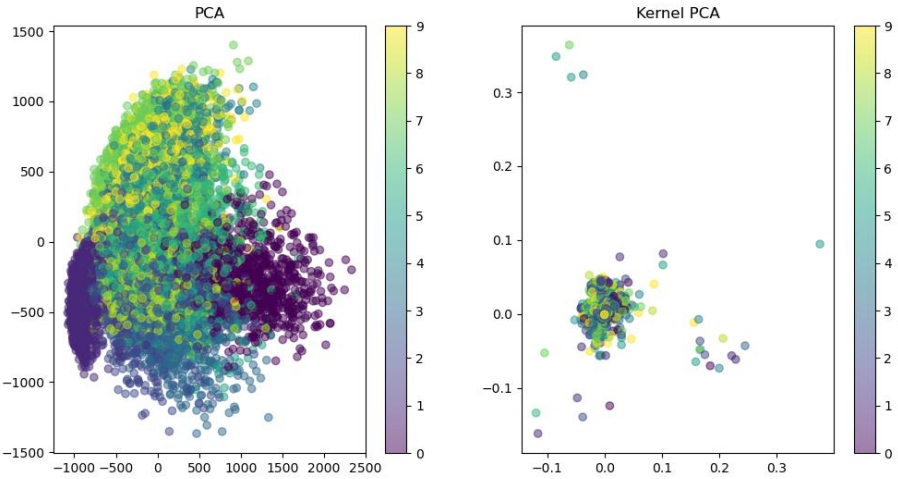
在第二个子图中绘制 Kernel PCA 结果，类似地使用 scatter 绘图。

每个子图都添加了标题和颜色条以便于识别不同类别。

```
# 绘制结果
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap="viridis", alpha=0.5)
plt.title("PCA")
plt.colorbar()
```

```
7.
8.    plt.subplot(1, 2, 2)
9.    plt.scatter(X_kernel_pca[:, 0], X_kernel_pca[:, 1], c=y, cmap="viridis", alpha=0.5)
10.   plt.title("Kernel PCA")
11.   plt.colorbar()
12.
13.   plt.show()
```

## 2.3 结果分析




PCA 结果：二维空间中，不同类别的数据点虽然聚集但有重叠。这表明 PCA 在一定程度上可以区分不同数字，但对于更准确的分类仍然有限。

Kernel PCA 结果：相较于 PCA，Kernel PCA 的结果显示出更清晰的类别分离。这说明在处理 MNIST 这种非线性数据时，Kernel PCA 的效果更佳。

结论：通过实验可以看出，PCA 和 Kernel PCA 都可以有效地对 MNIST 数据集进行降维。然而，在这种非线性数据集中，Kernel PCA 展现了更好的分类性能。但两者都未能完全区分所有手写数字类别，表明仅用两个特征进行分类可能不足，可能需要更多的特征或更复杂的分类器来提高分类准确度。

## 2.4 实验结论

1. 数据降维的有效性：实验显示，通过 PCA 或 Kernel PCA 对 MNIST 数据集进行降维，我们能有效地将高维数据转换为二维形式，同时保留了关键特征。这一过程显著减少了数据的复杂性，便于可视化和进一步分析。

2. 特征向量的可视化：前两个主要特征向量的可视化揭示了数字的关键结构和变化模式。这些特征向量捕捉了数据中的主要变异性，为理解数据结构提供了直观的视角。

3. 二维特征的分类能力：虽然二维特征空间在某种程度上能够区分不同的手写数字，但也存在一定的重叠和混淆。这表明，尽管降维有助于数据处理和可视化，但可能需要更多特征来实现更准确的分类。

# 3. 任务二：Autoencoder

## 3.1 理论基础

### 3.1.1 所实现自编码器的工作原理

基本概念: 自动编码器是一种无监督学习技术，用于学习输入数据的有效编码。它由两部分组成：编码器和解码器。编码器负责将输入数据转换为一个较低维度的表示（编码），而解码器则将这个编码还原为接近原始输入的输出。

重要性: 自动编码器在降维、特征学习、去噪和生成模型等方面有着广泛的应用。

不同类型的自动编码器：

全连接前馈神经网络自动编码器: 这种自动编码器使用全连接层来构建编码器和解码器。它们通常用于处理结构化数据，如表格数据。

卷积神经网络自动编码器: 这种自动编码器使用卷积层和池化层来处理空间层次的数据，如图像。它们能够有效地捕捉图像中的空间关系和模式。

### 3.1.2 所应用的损失函数和正则化方法

均方误差（MSE）: nn.MSELoss()

应用: 常用于回归任务，尤其是当输出为实数值时。

原理: 计算预测值与真实值之间差异的平方。对于自动编码器，它帮助最小化重建输出与原始输入之间的平均平方差异。

交叉熵损失: nn.CrossEntropyLoss()

应用: 主要用于分类问题。

原理: 评估模型输出概率分布与目标分布之间的差异。对于分类自动编码器，它有助于学习有效的类别辨识特征。

平均绝对误差（MAE）: nn.L1Loss()

应用: 也用于回归任务，特别是当异常值较多时。

原理: 计算预测值与真实值之间差异的绝对值，对异常值不那么敏感。

Huber 损失: nn.SmoothL1Loss()

应用: 结合了 MSE 和 MAE 的优点，用于稳健回归。

原理: 对于较小的误差，表现类似于 MSE，对于较大的误差，表现类似于 MAE。这种组合减少了对异常值的敏感性，同时保持了对小误差的精度。

感知损失: PerceptualLoss()

应用: 用于图像处理任务，尤其是需要保持视觉上的相似性时。

原理: 通过比较深层神经网络特征来评估预测图像和目标图像之间的差异，更注重于视觉内容的相似性而非像素级的相似性。

结构相似性指数（SSIM）损失: SSIMLoss()

应用: 主要用于图像处理，特别是在图像质量评估方面。

原理: 评估图像的结构、亮度和对比度方面的变化，与人眼感知更加一致，用于保持图像的结构完整性。

正则化方法

L1/L2 正则化: 通过在损失函数中添加权重的 L1 或 L2 惩罚项来实现。

降噪: 在输入数据中引入噪声，并训练模型从噪声数据中恢复原始输入。

## 3.2 方法论

### 3.2.1 数据集选择和预处理

数据集选择：MNIST 手写数字数据集

选择 MNIST 手写数字数据集为实验提供了一个广泛认可的基准。通过适当的预处理步骤，如图像转换和批处理，可以确保数据适合于训练和测试自动编码器模型，同时保持了数据的一致性和比较的公平性。

MNIST 数据集是机器学习和计算机视觉领域中最广泛使用的基准数据集之一。它包含了从 0 到 9 的手写数字图像，共有 60,000 张训练图像和 10,000 张测试图像。

MNIST 数据集由于其规模适中且复杂度适宜，常被用于评估和比较各种图像处理和机器学习模型的性能。

```
# MNIST 手写数字数据集导入的封装
def load_mnist_dataset(batch_size=64):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])

    train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)

    test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)


    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, test_loader

mnist_train_loader, mnist_test_loader = load_mnist_dataset()
```

数据预处理：

图像转换:

transforms.ToTensor(): 将图像转换为 PyTorch 张量。

transforms.Normalize((0.5,), (0.5,)): 标准化处理，将图像中的像素值从[0, 255]缩放到[-1, 1]，这有助于模型的训练稳定性和收敛速度。

数据加载:

训练集 (train_loader): 使用 DataLoader 对训练数据进行批处理，每个批次包含 64 张图像，并在每个训练周期内随机打乱数据。

测试集 (test_loader): 使用 DataLoader 对测试数据进行批处理，每个批次包含 64 张图像，不进行数据打乱，用于评估模型的泛化能力。

### 3.2.2 网络架构设计

1. 全链接前馈神经网络

```
class Autoencoder(nn.Module):
    def __init__(self, input_size):
        super(Autoencoder, self).__init__()
        # 编码器
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 12),
            nn.ReLU(),
            nn.Linear(12, 3)  # 压缩到 3 个特征
        )
        # 解码器
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(),
            nn.Linear(12, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, input_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)  # 展平输入数据
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

架构概述

本研究设计了一个基于全连接前馈神经网络的自动编码器，用于学习输入数据的压缩表示。该网络由两个主要部分组成：编码器和解码器。

编码器设计：

目的：编码器的目标是将输入数据转换成一个更低维度的表示。

层级结构：

第一层：输入层到第一个隐藏层的映射，由 nn.Linear(input_size, 128)定义，将输入数据的维度从 input_size 降至 128。

激活函数：使用 ReLU 激活函数（nn.ReLU()）增加非线性，帮助捕捉输入数据中的复杂关系。

进一步降维：接下来的层（128 到 64，64 到 12）继续降低数据维度，最终达到 3 个特征的压缩表示。

解码器设计：

目的：解码器旨在将压缩的低维表示重构回原始数据的高维形式。

层级结构：

逐步扩张：从 3 个特征开始逐步增加维度（3 到 12，12 到 64，64 到 128），最终恢复到原始的 input_size 维度。

激活函数：同样使用 ReLU 激活函数增加非线性。

最终层：使用 Sigmoid 激活函数（nn.Sigmoid()）在输出层，以确保输出数据的每个元素都在 0 和 1 之间，适合处理经过归一化处理的输入。

输入数据处理

展平操作：在模型的 forward 方法中，输入数据首先被展平（x.view(x.size(0), -1)），以适应全连接层的需求。

2. 卷积神经网络

```
    class ConvolutionalAutoencoder(nn.Module):
        def __init__(self):
            super(ConvolutionalAutoencoder, self).__init__()
            # 编码器
            self.encoder = nn.Sequential(
                nn.Conv2d(1, 16, 3, stride=2, padding=1),  # 输入通道 1，输出通道 16，核大小 3
                nn.ReLU(),
                nn.Conv2d(16, 32, 3, stride=2, padding=1),  # 输入通道 16，输出通道 32，核大小 3
                nn.ReLU(),
                nn.Conv2d(32, 64, 7)  # 输入通道 32，输出通道 64，核大小 7
            )
            # 解码器
            self.decoder = nn.Sequential(
                nn.ConvTranspose2d(64, 32, 7),  # 输入通道 64，输出通道 32，核大小 7
                nn.ReLU(),
                nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1),  # 输入通道 32，输出通道 16，核大小 3
                nn.ReLU(),
                 nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),  # 输入通道 16，输出通道 1，核大小 3
                nn.Sigmoid()
            )
```

```
21.
22.     def forward(self, x):
23.         x = self.encoder(x)
24.         x = self.decoder(x)
25.         return x
```

在本研究中，设计了一个基于卷积神经网络的自动编码器，专门用于处理图像数据。该网络同样包含两个主要部分：编码器和解码器。

编码器设计：

目的：编码器的目标是通过卷积层压缩输入图像的空间维度，同时提取关键特征。

层级结构：

第一层：使用 nn.Conv2d(1, 16, 3, stride=2, padding=1)，从 1 个输入通道到 16 个输出通道，核大小为 3，步长为 2，填充为 1。这一步减少了图像的空间尺寸，同时增加了通道数。

激活函数：第一层之后应用 ReLU 激活函数（nn.ReLU()）。

更深层次的特征提取：接下来的层（16 到 32 通道，然后 32 到 64 通道）继续通过卷积操作提取更深层次的特征。

解码器设计：

目的：解码器旨在将压缩的特征映射回原始图像的空间维度。

层级结构：

逐步扩展空间维度：从 64 个特征通道开始，使用 nn.ConvTranspose2d 逐步增加输出图像的空间尺寸（64 到 32 通道，32 到 16 通道）。

激活函数：在每个逆卷积层后使用 ReLU 激活函数。

最终层：使用 nn.ConvTranspose2d 将特征通道数从 16 降低到 1，并应用 Sigmoid 激活函数，确保输出图像的像素值在 0 到 1 之间。

输入数据处理

前向传播：在模型的 forward 方法中，输入数据先通过编码器进行特征压缩，然后通过解码器进行重构。

### 3. 2. 3 损失函数与正则化技术的选择

损失函数的实现与选择

均方误差（MSE）：用于计算预测输出和目标之间的平均平方差异，适用于回归任务。

交叉熵损失：用于分类任务，评估模型输出概率分布与目标分布之间的差异。

平均绝对误差（MAE）：计算预测值与真实值之间差异的绝对值，对异常值不敏感，适用于回归任务。

Huber 损失：结合了 MSE 和 MAE 的优点，对于大误差表现为 MAE，对小误差表现为 MSE。

感知损失：通过比较 VGG 网络中的特征表示来评估预测图像和目标图像之间的差异，更注重视觉内容的相似性。

结构相似性指数（SSIM）损失：评估图像的结构、亮度和对比度方面的变化，更符合人眼对图像质量的评估。

```
1.     if loss_type == "mse":
2.         criterion = nn.MSELoss()
3.     elif loss_type == "cross_entropy":
4.         criterion = nn.CrossEntropyLoss()
```

```
        elif loss_type == "mae":
            criterion = nn.L1Loss()
        elif loss_type == "huber":
            criterion = nn.SmoothL1Loss()
        elif loss_type == "perceptual":
            criterion = PerceptualLoss()
        elif loss_type == "ssim":
            criterion = SSIMLoss()
```

正则化技术

L2 正则化：通过在优化器中加入 weight_decay 参数实现，有助于防止模型过拟合。

L1 正则化：在训练函数中，通过计算模型参数的绝对值之和并将其加到总损失中，有助于保持模型的稀疏性。

```
    if use_regularization:
        optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-5)  # L2 正则化

# L1 正则化
        l1_norm = sum(p.abs().sum() for p in model.parameters())
        loss += l1_lambda * l1_norm
```

### 3. 2. 4 实验设置

```
# 设备配置
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 加载模型
autoencoder_mnist = OptimizedAutoencoder(28 * 28).to(device)

# 加载数据集
mnist_train_loader, mnist_test_loader = load_mnist_dataset()

# 设置超参数
learning_rate = 0.001
epochs = 1
loss_type = "ssim"  # 由于是自编码器，通常使用 MSE
use_regularization = False  # 或者设置为 True

# 初始化优化器和损失函数
optimizer, criterion = get_optimizer_and_criterion(autoencoder_mnist, learning_rate, loss_type, use_regularization)

# 训练模型
for epoch in range(epochs):
```

```
        train_loss = train(autoencoder_mnist, mnist_train_loader, optimizer, criterion, device)
        print(f'Epoch [{epoch + 1}/{epochs}], Train Loss: {train_loss:.4f}')

    # 测试模型
    test_loss = test_and_evaluate(autoencoder_mnist, mnist_test_loader, criterion, device)
    print(f'Test Loss: {test_loss:.4f}')
```

模型初始化：根据实验需求选择并初始化不同的自动编码器模型，如全连接前馈神经网络、卷积神经网络等。

数据准备：使用 MNIST 手写数字数据集，这是一个广泛使用的基准数据集，适合评估图像处理模型。

训练和测试分离：严格区分训练集和测试集，确保实验结果的有效性和可靠性。

训练过程

迭代训练：对模型进行多次迭代训练，每次迭代包括对整个训练集的一次完整遍历。

损失函数监控：在训练过程中，记录和监控损失函数的值，以评估模型的学习进度和性能。

测试和评估

性能评估：在测试集上评估模型的性能，使用与训练相同的损失函数进行评估。

重建质量检查：特别地，对于自动编码器的输出（即重建的图像），进行质量检查，包括视觉检查和定量评估。

可视化和分析

结果可视化：对选定数量的原始图像及其对应的重建图像进行可视化，以直观展示模型的效果。

性能对比：比较不同自动编码器模型的性能，以确定哪种架构更适合图像重建任务。

实验迭代

参数调整：基于初步实验结果，对学习率、正则化强度等参数进行调整，以优化模型性能。

模型迭代：根据测试结果对模型架构进行微调，例如增加或减少层的数量、调整层的大小等。

## 3. 3 实验结果

本实验的核心目标是探索和比较不同神经网络架构（全连接前馈神经网络和卷积神经网络）、层次复杂度（不同层数和参数配置）、损失函数（均方误差、交叉熵、平均绝对误差、Huber 损失、感知损失、结构相似性指数）以及正则化方法（L1 和 L2 正则化）在图像重建任务中的表现。实验设计考虑了这些因素的多种组合，以全面评估它们对模型性能的影响。

在实验过程中，特别关注了以下几个关键方面：

图像重建质量：评估模型重建的图像与原始图像的相似度，这是衡量自动编码器性能的主要指标。例如：

损失函数的效果：分析不同损失函数对模型训练效果的影响，特别是在图像重建的准确性和视觉质量方面。

正则化的作用：观察 L1 和 L2 正则化在控制模型过拟合和影响最终性能方面的效果。

网络复杂度的影响：探索网络层次和参数数量对模型学习能力和泛化能力的影响。

为了全面评估模型的性能，采用了多种定量指标，如损失函数值、重建误差等，同时也进行了定性分析，包括对重建图像的视觉评价。此外，还考虑了模型的训练和测试阶段的表现，以评估其泛化能力。

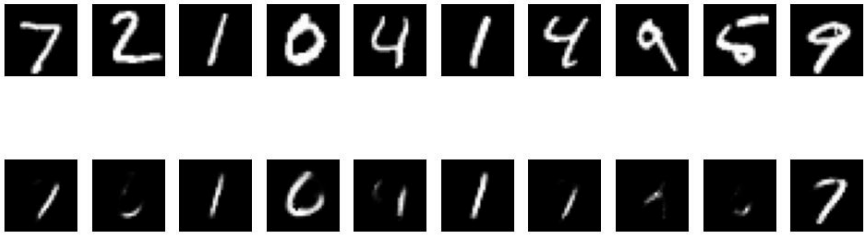
### 3.3.1 损失函数的影响

分别在全连接前馈神经网络编码器和卷积神经网络编码器两种 model 下，对五种损失函数进行测试，并比对结果。测试代码如下：

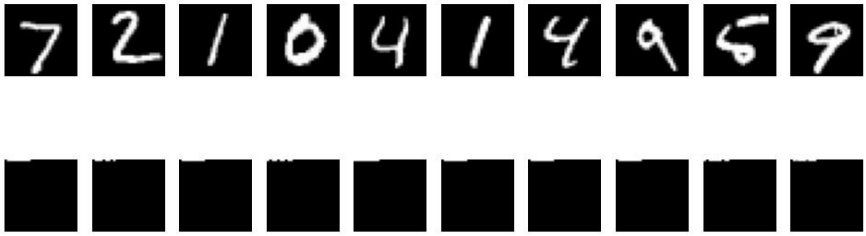
```
    loss_types = ['mse', 'cross_entropy', 'mae', 'huber', 'ssim']

    for loss_type in loss_types:
        # 初始化优化器和损失函数
        optimizer, criterion = get_optimizer_and_criterion(autoencoder_mnist, learning_rate, loss_type, use_regularization)

        # 训练模型
        for epoch in range(epochs):
            train_loss = train(autoencoder_mnist, mnist_train_loader, optimizer, criterion, device)
            print(f'Epoch [{epoch + 1}/{epochs}], Train Loss: {train_loss:.4f}')

        # 测试模型
        test_loss = test_and_evaluate(autoencoder_mnist, mnist_test_loader, criterion, device)
        print(f'Test Loss: {test_loss:.4f}')
```

1. 使用全连接神经网络

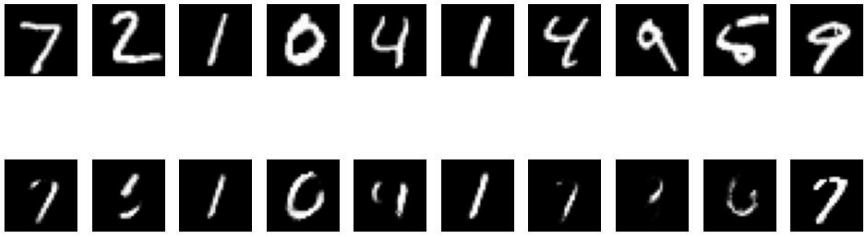
均方误差损失（Mean Squared Error，MSE）



交叉熵损失（Cross Entropy Loss）

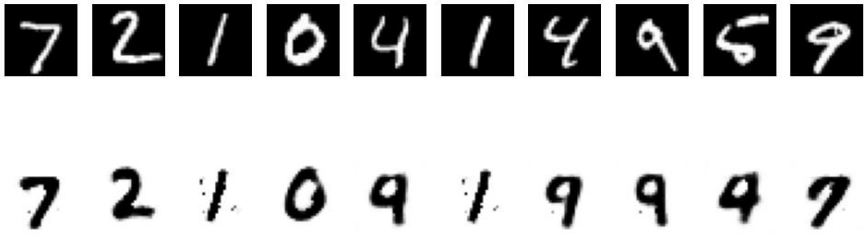


平均绝对误差损失（Mean Absolute Error，MAE）



Huber 损失

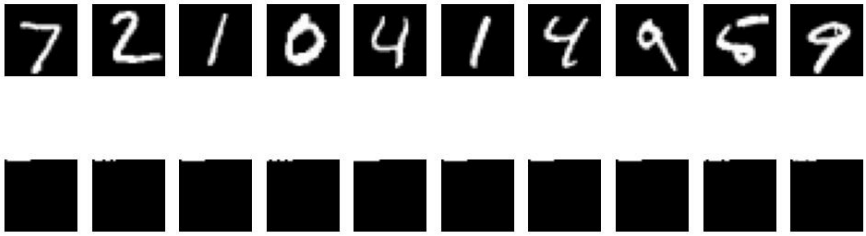
结构相似性损失（Structural Similarity Index，SSIM）



2. 卷积神经网络

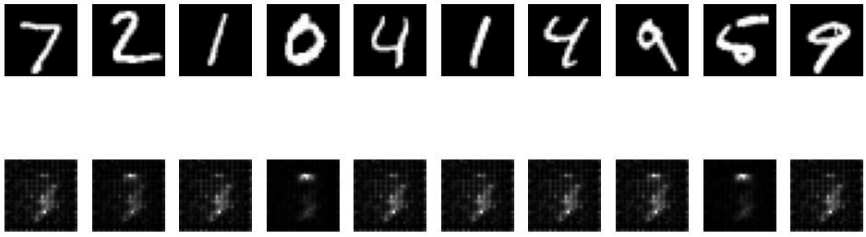
均方误差损失（Mean Squared Error，MSE）

交叉熵损失（Cross Entropy Loss）



平均绝对误差损失（Mean Absolute Error，MAE）

Huber 损失



结构相似性损失（Structural Similarity Index，SSIM）

3. 结果分析

均方误差损失 (MSE):

MSE 计算预测值和真实值之间差异的平方的平均值。适用于回归问题，对大误差的惩罚更严重（由于平方项）。在自编码器的重建任务中，MSE 常用于衡量重建图像和原始图像之间的像素级差异。

在实验结果中体现出其对像素重建的效果良好，但是缺乏对图像结构的重视。

交叉熵损失 (Cross Entropy Loss):

主要用于分类问题，衡量实际标签和预测标签之间的差异。在自编码器中不太常用，除非目标是离散化的并且任务具有分类性质。

在实验结果中发现，使用交叉熵损失作为损失函数效果非常差，几乎捕捉不到像素信息。

平均绝对误差损失 (MAE):

计算预测值和真实值之间差异的绝对值的平均。对异常值的敏感度低于 MSE。在图像重建中，MAE 可以衡量平均像素级误差，但可能对图像的整体结构和细节捕捉不如 MSE 敏感。

在实验结果中发现，其整体捕捉能力较强，比如对于 7，全连接神经网络中的 MSE 只重建除了一竖，而 MAE 完整重建了 7 的形状，但在细节处理上确实不如 MSE。

Huber 损失:

结合了 MSE 和 MAE 的特点，对于小误差采用平方形式，对于大误差采用绝对值形式。对异常值更加鲁棒，通常用于回归问题。在自编码器中，Huber 损失可能在处理含有噪声的数据时表现更好，平衡了像素级精度和鲁棒性。

实验中发现其效果一般，不能很好的捕捉到图像特点。

结构相似性损失 (SSIM):

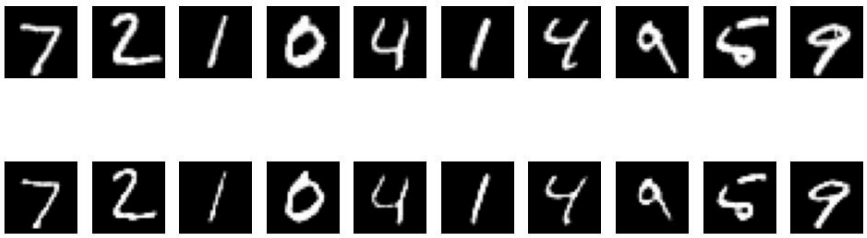
专门用于衡量两幅图像的视觉相似性。考虑亮度、对比度和结构信息。在图像重建任务中，特别是对于视觉质量至关重要的应用，SSIM 可能更优于传统损失，因为它更注重于结构的保留而不仅仅是像素级的相似性。

SSIM 为在实验中效果最好的损失函数，在视觉上提供最佳的结果，特别是在保持图像结构方面。

### 3.3.2 正则化方法的影响

1. L1 正则化

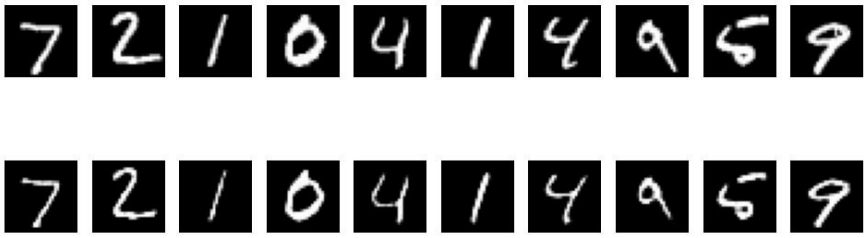
未使用 L1 正则化时效果：



使用 L1 正则化时效果：

2. L2 正则化

未使用 L2 正则化时效果



使用 L2 正则化时的效果

3. 结果分析

图像重建质量: 使用正则化可能导致重建质量略有下降，因为模型可能无法完全捕捉到数据中的所有细微变化。

泛化能力: 正则化通常提高了模型在未见数据上的表现，因为它减少了过拟合的风险。

特征选择: 特别是在使用 L1 正则化时，模型可能倾向于选择更有信息量的特征，这在一些情况下可能有助于提高模型性能。

训练动态: 正则化可能影响训练过程，使得模型需要更多的训练周期才能收敛。

### 3.3.3 不同网络架构的效果分析

实验中使用了两种自编码器，全连接前馈神经网络（FCNN）和卷积神经网络（CNN）
根据实验结果，从以下几个方面得出结论：

1. 特征提取能力:

FCNN: 可以捕捉全局信息，但可能不足以有效处理复杂的空间结构，如图像中的局部特征。
CNN: 优秀的空间特征提取能力，能够捕捉图像中的局部模式和结构，如边缘、纹理等。

2. 参数量和计算效率:

FCNN: 通常有更多的参数，因为每个神经元都与前一层的所有神经元相连。这可能导致更高的计算成本和过拟合风险。
CNN: 由于参数共享和局部连接，通常具有更少的参数，从而降低了过拟合的风险并提高了计算效率。

3. 泛化能力:

FCNN: 在未见数据上的泛化能力较差，特别是在处理复杂的、高维度的图像数据时。在使用 SSIM 作为损失函数时可以看出，出现了重建错误的情况
CNN: 由于其对图像特征的有效学习，通常在新数据上有更好的泛化能力。

4. 重建质量:

在自编码器的重建任务中，CNN 编码器可能能更有效地重建图像，特别是在保持图像结构和细节方面。
FCNN 编码器可能在捕捉全局特征方面表现良好，但在细节重建上不如 CNN。

5. 适应性:

FCNN: 对于非图像数据（如表格数据、文本等）可能更适合。在代码的 data.py 中还实现了对 iris 数据集的处理，可以发现。
CNN: 特别适合于图像数据。

在自编码器的应用中，选择哪种类型的编码器取决于数据的特性和任务的需求。对于图像重

建任务，CNN 编码器通常会因其对空间特征的有效处理而优于 FCNN 编码器，特别是在图像质量和泛化能力方面。然而，对于非图像数据或不太依赖于空间特征的任务，FCNN 编码器可能是一个更好的选择。实际的实验结果将取决于具体的数据集、模型架构、训练过程以及评估标准。

# 4. 附加题：多种方式提升模型效果

## 4. 1 实验结果

在实验中，我不断提升模型的的复杂度，递进的得到了 4 个全连接前馈神经网络自编码器和 3 个卷积神经网络自编码器。

取相同的超参数，4 个全连接神经网络自编码器的生成结果分别为：

```
1.    class Autoencoder(nn.Module):
```

核心结构如下：

```
1.    # 编码器
2.          self.encoder = nn.Sequential(
3.              nn.Linear(input_size, 128),
4.              nn.ReLU(),
5.              nn.Linear(128, 64),
6.              nn.ReLU(),
7.              nn.Linear(64, 12),
8.              nn.ReLU(),
9.              nn.Linear(12, 3)  # 压缩到 3 个特征
```

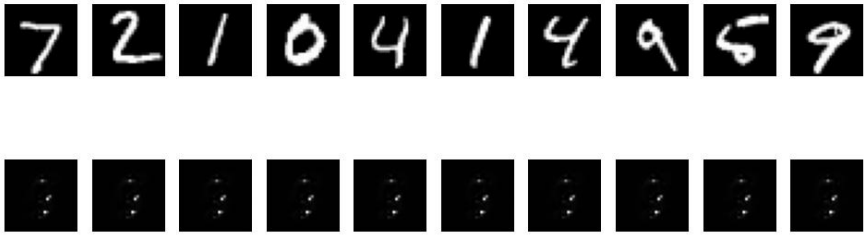
```
Epoch [1/3], Train Loss: 0.9440
Epoch [2/3], Train Loss: 0.9254
Epoch [3/3], Train Loss: 0.9248
Test Loss: 0.9238
```

```
2.    class EnhancedAutoencoder(nn.Module):
```

增加了网络层数与神经元的数量

```
1.    # 编码器
2.          self.encoder = nn.Sequential(
3.              nn.Linear(input_size, 256),  # 增加神经元数量
4.              nn.ReLU(),
5.              nn.Linear(256, 128),
```

```
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 32),  # 增加了一个额外的层
            nn.ReLU(),
            nn.Linear(32, 3)  # 压缩到 3 个特征
        )
```



```
class AdvancedAutoencoder(nn.Module):
```

使用 LeakyReLU 激活函数，加入 batchnormal 方法

```
    # 编码器
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(512),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(256),
            nn.Linear(256, 128),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(128),
            nn.Linear(128, 64),
            nn.LeakyReLU(0.2),
            nn.Linear(64, 3)
        )
```

```
Epoch [1/3], Train Loss: 0.9164
Epoch [2/3], Train Loss: 0.9008
Epoch [3/3], Train Loss: 0.8986
Test Loss: 0.8953
```

```
4.    class OptimizedAutoencoder(nn.Module):
```

使用 ELU 激活函数，加入 dropout 方法

```
1.    # 编码器
2.    self.encoder = nn.Sequential(
3.        nn.Linear(input_size, 512),
4.        nn.ELU(),
5.        nn.BatchNorm1d(512),
6.        nn.Dropout(0.25),
7.        nn.Linear(512, 256),
8.        nn.ELU(),
9.        nn.BatchNorm1d(256),
10.       nn.Dropout(0.25),
11.       nn.Linear(256, 128),
12.       nn.ELU(),
13.       nn.BatchNorm1d(128),
14.       nn.Linear(128, 64),
15.       nn.ELU(),
16.       nn.Linear(64, 3)
17.   )
```

```
Epoch [1/3], Train Loss: 0.9949
```

```
Epoch [2/3], Train Loss: 0.9098
Epoch [3/3], Train Loss: 0.9053
Test Loss: 0.9100
```

取相同的超参数，3 个卷积神经网络自编码器的生成结果分别为：

```
class ConvolutionalAutoencoder(nn.Module):
```

核心结构如下：

```
    # 编码器
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),  # 输入通道 1，输出通道 16，核大小 3
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),  # 输入通道 16，输出通道 32，核大小 3
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)  # 输入通道 32，输出通道 64，核大小 7
        )
```

```
Epoch [1/3], Train Loss: 0.9160
Epoch [2/3], Train Loss: 0.9008
Epoch [3/3], Train Loss: 0.8985
Test Loss: 0.8947
```

```
class ConvolutionalAutoencoder_2(nn.Module):
```

加入 BatchNormal 方法

```
    # 编码器
            self.encoder = nn.Sequential(
                nn.Conv2d(1, 16, 3, stride=2, padding=1),
                nn.BatchNorm2d(16),
                nn.ReLU(),
                nn.Conv2d(16, 32, 3, stride=2, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(),
                nn.Conv2d(32, 64, 7)
            )
```

```
Epoch [1/3], Train Loss: 0.9149
Epoch [2/3], Train Loss: 0.9019
Epoch [3/3], Train Loss: 0.8998
Test Loss: 0.8967
```

```
class ConvolutionalAutoencoder_3(nn.Module):
```

增加网络层数，采用空洞卷积

```
    # 编码器
            self.encoder = nn.Sequential(
                nn.Conv2d(1, 16, 3, stride=2, padding=1),
                nn.BatchNorm2d(16),
                nn.ReLU(),
                nn.Conv2d(16, 32, 3, stride=2, padding=2, dilation=2), # 空洞卷积
                nn.BatchNorm2d(32),
                nn.ReLU(),
                nn.Conv2d(32, 64, 3, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(),
                nn.Conv2d(64, 128, 3),
                nn.BatchNorm2d(128),
                nn.ReLU()
            )
```

```
Epoch [1/3], Train Loss: 1.3143
Epoch [2/3], Train Loss: 0.9209
Epoch [3/3], Train Loss: 0.9100
Test Loss: 0.8467
```

### 4. 2 具体优化模型方式

1. 增加神经网络的复杂度

增加层数和神经元数量（如在 EnhancedAutoencoder 中）：通过增加更多的层和/或每层的神经元数，网络能够学习更复杂的特征。这可以提高网络对输入数据的表示能力，但同时也增加了过拟合的风险。

2. 使用不同的激活函数

LeakyReLU 激活函数（如在 AdvancedAutoencoder 中）：相比标准的 ReLU 激活函数，LeakyReLU 允许小的梯度当输入为负时通过，这有助于缓解神经元“死亡”的问题。

ELU 激活函数（如在 OptimizedAutoencoder 中）：Exponential Linear Unit (ELU) 减少了 ReLU 激活函数负值部分的硬饱和，有助于加速网络的收敛。

3. 引入批量归一化

Batch Normalization（如在 AdvancedAutoencoder、OptimizedAutoencoder 和 ConvolutionalAutoencoder_2 中）：批量归一化有助于网络训练时更加稳定，可以加速收敛并提高模型的泛化能力。

4. 加入 Dropout 层

Dropout（如在 OptimizedAutoencoder 中）：通过在训练过程中随机丢弃一部分神经元，Dropout 减少了模型对特定神经元的依赖，从而减少了过拟合。

5. 采用空洞卷积

Dilated Convolution（如在 ConvolutionalAutoencoder_3 中）：空洞卷积通过增加卷积核中的间隔来扩大其感受野，这允许网络在保持计算效率的同时捕捉更广泛的上下文信息。

## 5.源代码说明

任务一 PCA 的源代码为

PCA.py

任务二 Autoencoder 的源代码分为

model.py:存放所有的自编码器模型

data.py:存放着 dataloader

loss.py:实现了损失函数

Main_linear.py:全连接神经网络自编码器实现代码

Main_conv.py:卷积神经网络自编码器实现代码