哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

# 《模式识别》课程实验报告

## 实验三：高光谱图像语义分割


院　　系：机电工程学院自动化系

姓　　名：朱颖聪

学　　号：200320708

班　　级：自动化 7 班


日　　期：2023 年 12 月 30 日

# 目录

# 高光谱语义分割

## 1 项目背景

### 1.1 背景介绍

高光谱图像是对某一区域内的若干波段内的光谱能量的记录，波段宽度往往在 10nm 左右，而其记录的波长范围可达 400 至 2500nm，远超可见光谱范围。因而相较于常见的三通道 RGB 图像，针对同一场景和同一幅画，得益于高光谱图像的宽感知范围和窄波段（更多的通道数），高光谱图像包含更多的信息。这使得我们可以通过对高光谱图像进行数据分析，获得传统三通道 RGB 图像无法提供给我们的实用信息。目前高光谱图像在遥感、医学、农学、食品安全等领域得到了光感的研究和应用。

### 1.2 数据集说明

本项目使用的高光谱图像来自 AVIRIS 传感器，采集于印第安纳州西北部的印度松测试点。该图像包含了 145×145 像素，每个像素对应 224 个光谱反射带，覆盖波长范围为 400 至 2500nm。印度松树景观中包含了农田、森林、道路、建筑等多种地物，且标签矩阵将图像分为 16 个地物类别，不考虑背景类像素。

## 2 项目目标

本项目的主要目标是实现高光谱图像的语义分割，即将图像中的每个像素精准地分配到相应的地物类别中。具体而言，项目要求对图像中的 16 类像素进行训练和测试，划分训练集和测试集时，每一类像素的 70%作为训练集，剩余 30%像素作为测试集。要求分类器的测试准确率达到 80%以上。为降低分类难度，背景类像素将被忽略。

## 3 理论介绍

**1. K-L 变换：**

K-L 变换是一种特征提取方法，通过线性变换将数据映射到一个新的坐标系，使得在新坐标系中数据的方差最大化。K-L 变换可用于将原始图像转换为更紧凑的特征表示，有助于提高后续分类器的性能，并且可以大大简化训练成本。通过 K-L 变换，可以将原始高维光谱数据转换为一组新的互相不相关的特征，称为主

成分。这些主成分按照其对数据方差的贡献递减的顺序排列，允许保留数据中的主要信息并减小数据的维度。在高光谱图像语义分割项目中，K-L 变换可用于减少输入特征的维度，提高分类器的效率，并有助于消除特征之间的冗余信息，从而更好地捕捉数据的关键特征。

**2. 多层感知机（MLP）：**

多层感知器（MLP）是一种人工神经网络结构，具有多个层（包括输入层、隐藏层和输出层）和非线性激活函数。通过前向传播和反向传播算法，MLP 能够学习输入数据的复杂模式和特征，适用于分类和回归等任务。在高光谱图像语义分割项目中，MLP 可用于将每个像素准确地分配到相应的地物类别，通过训练和优化提高分类准确率，实现对图像的精准分割和地物分类。

# 4 实验步骤

本次实验思路为：加载图像数据—>>特征降维——>>分类器设计。

特征降维的方法主要使用 K-L 变换，分类方法使用人工神经网络，主要为多层感知机（MLP）。

**（1）加载数据**

由于数据集为 mat 文件，需要进行预处理，将数据中每个类别的 70%作为训练集、30%作为测试集。由于不同类别数目不同，需要对训练集进行扩充，使得不同类别在训练集中的占比一致。除此之外，对数据各维度的特征进行归一化处理。

**（2） K-L 变换**

对训练集的图像计算协方差矩阵，计算特征值和特征向量，根据累计贡献率选取前 m 大的特征向量（代码见附录）。

$$\eta_m = \frac{\lambda_1 + \lambda_2 + \cdots + \lambda_m}{\lambda_1 + \lambda_2 + \cdots + \lambda_p}$$

求得的特征向量将保存到 json 文件中，为后续的降维处理保存数据。

**（3） 获得降维训练集和测试集**

将步骤 1 中获得的 200 维度的训练集和测试集与特征向量相乘降维，获得降维的训练集与测试集，并转换为 Pytorch 的 Dataset 类（代码见附录）。

**（4） MLP 模型构建**

设计 MLP 模型，选择合适的层数、每层神经元数量、激活函数和损失函数。考虑到语义分割任务，输出层的神经元数量为 16，输入层的神经元个数则跟 K-L 变换后的特征维数一致。

初始网络结构选择为

```
def __init__(self, num):
    super(HyperSpectralModel, self).__init__()
    self.output = nn.Sequential(
        nn.Linear(num, 250),
        nn.ReLU(True),
        nn.Linear(250, 100),
        nn.ReLU(True),
        nn.Linear(100, 50),
        nn.ReLU(True),
        nn.Linear(50, 16),
    )
```

后根据实验效果进行调整。

**（5） 选取合适的优化器和损失函数。**

由于这个是多分类问题，所以损失函数选择为交叉熵损失函数、而优化器选择 Adadelta，其可以自动调节步长，实现快速优化。

**（6） 调节参数，训练模型，评估模型。（代码见附录）**

# 5 调参过程及结果

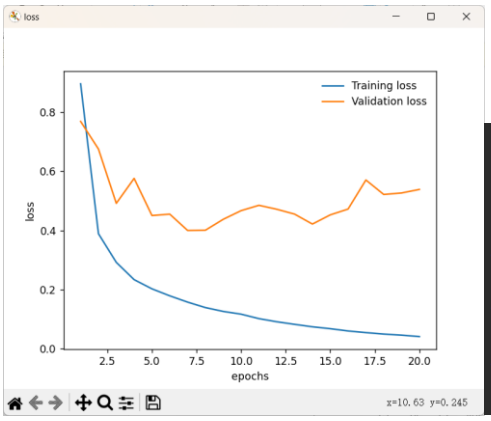
**（1）调节 K-L 变换的主成分数量**

选择累计贡献率为$\eta_m = 0.99$和$\eta_m = 0.999$，其主成分数量分别为 39 和 108。选择 batch_size = 128，epochs=20，分别降维后的数据训练，对比结果。

网络结构为

```
def __init__(self, num):
    super(HyperSpectralModel, self).__init__()
    self.output = nn.Sequential(
        nn.Linear(num, 250),
        nn.ReLU(True),
        nn.Linear(250, 100),
        nn.ReLU(True),
        nn.Linear(100, 50),
        nn.ReLU(True),
        nn.Linear(50, 16),
    )
```

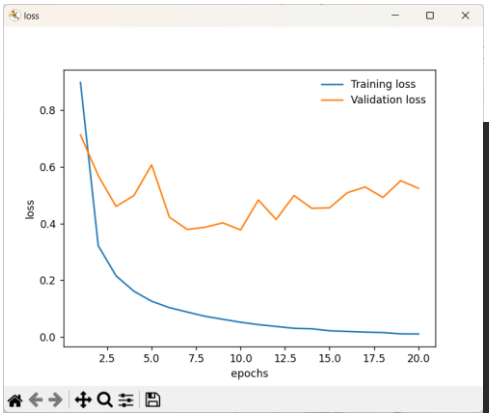
$\eta_m = 0.99$ ：




```
e: 0.99
num: 39
开始测试.....
平均损失：  0.3584103218454402
准确率：  0.8695228821811101
```

由于数据量过小，网络随着训练进行很快达到过拟合，其测试损失最低点在第 8 个回合，故选取第 6 个回合作为最佳模型进行测试，最终准确率为 0.8695。

$\eta_m = 0.999$ ：




```
e: 0.999
num: 108
开始测试.....
平均损失：  0.18632197506995
准确率：  0.943524829600779
```

最佳模型在第 9 回合，选择其作为测试模型，最终准确率为 0.9435。

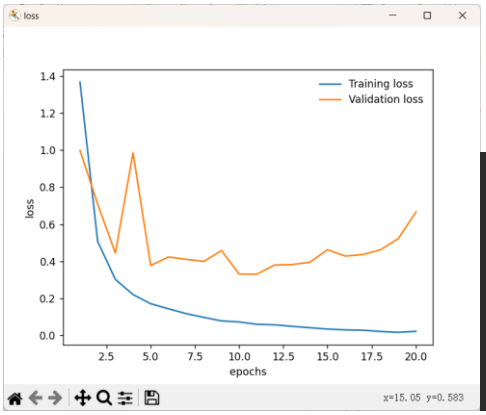
**（2）调节网络结构**

选择$\eta_m$ = 0.999，batch_size = 128，epochs=20。

- **增加网络深度，**结构变为

```
def __init__(self, num):
    super(HyperSpectralModel, self).__init__()
    self.output = nn.Sequential(
        nn.Linear(num, 250),
        nn.ReLU(True),
        nn.Linear(250, 200),
        nn.ReLU(True),
        nn.Linear(200, 150),
        nn.ReLU(True),
        nn.Linear(150, 100),
        nn.ReLU(True),
        nn.Linear(100, 50),
        nn.ReLU(True),
        nn.Linear(50, 16),
    )
```

结果：best_epoch = 10

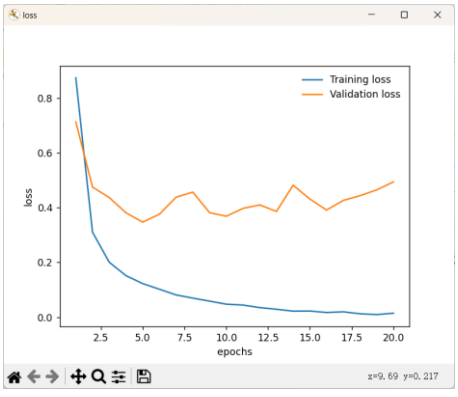



```
e: 0.999
num: 108
开始测试.....
平均损失:  0.15737711413763464
准确率:  0.9503407984420643
```

- **不改变深度，增加宽度，**结构变为

```
def __init__(self, num):
    super(HyperSpectralModel, self).__init__()
    self.output = nn.Sequential(
        nn.Linear(num, 400),
        nn.ReLU(True),
        nn.Linear(400, 200),
        nn.ReLU(True),
        nn.Linear(200, 100),
        nn.ReLU(True),
        nn.Linear(100, 16),
    )
```

结果：best_epoch = 4




```
e: 0.999
num: 108
开始测试.....
平均损失:  0.2827636918239296
准确率:  0.8938656280428432
```

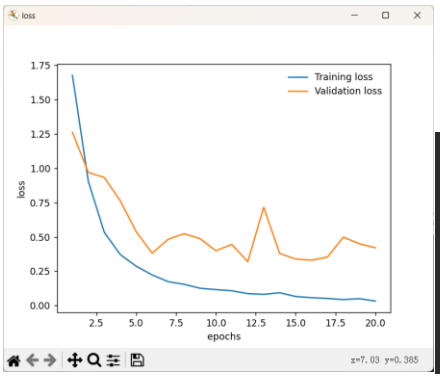
（3） 调节 batch size

选择$\eta_m$ = 0.999，epochs=20，网络结构为

```
def __init__(self, num):
    super(HyperSpectralModel, self).__init__()
    self.output = nn.Sequential(
        nn.Linear(num, 250),
        nn.ReLU(True),
        nn.Linear(250, 200),
        nn.ReLU(True),
        nn.Linear(200, 150),
        nn.ReLU(True),
        nn.Linear(150, 100),
        nn.ReLU(True),
        nn.Linear(100, 50),
        nn.ReLU(True),
        nn.Linear(50, 16),
    )
```

- batch_size = 256

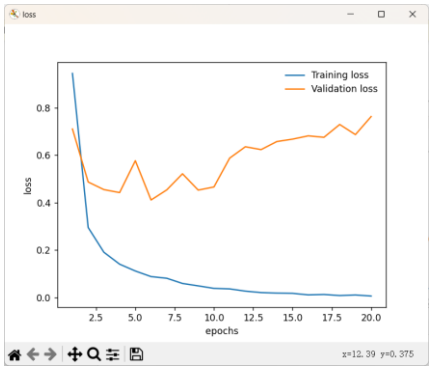
结果：best_epoch = 11




```
e: 0.999
num: 108
开始测试.....
平均损失： 0.30539489343048515
准确率： 0.881207400194742
```

- batch_size = 64

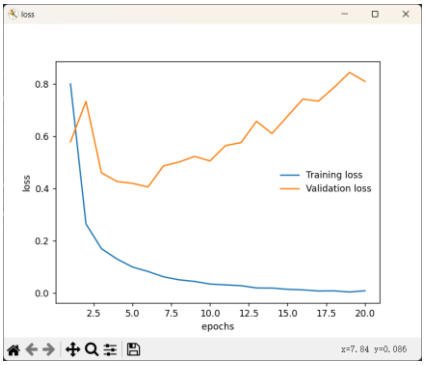
结果：best_epoch = 5




```
e: 0.999
num: 108
开始测试.....
平均损失： 0.40000617229474744
准确率： 0.8980850373255437
```

- batch_size = 32

结果：best_epoch = 5

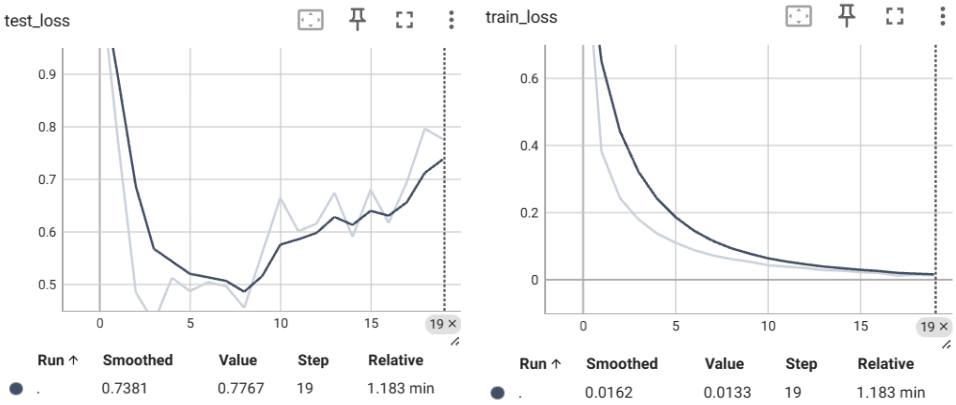



```
e: 0.999
num: 108
开始测试.....
平均损失： 0.23647375284071664
准确率： 0.9279454722492697
```

**（4）增加 Dropout 层**

```
class HyperSpectralModel(nn.Module):
    def __init__(self, num):
        super(HyperSpectralModel, self).__init__()
        self.output = nn.Sequential(
            nn.Linear(num, 250),
            nn.ReLU(True),
            nn.Linear(250, 200),
            nn.ReLU(True),
            nn.Linear(200, 150),
            nn.ReLU(True),
            nn.Dropout(0.5),
            nn.Linear(150, 100),
            nn.ReLU(True),
            nn.Linear(100, 50),
            nn.ReLU(True),
            nn.Linear(50, 16),
        )
```

调节 batch_size=64, $\eta_m = 0.999$




Best_epoch = 9,测试准确率为 0.9555

```
e: 0.999
num: 108
开始测试……
平均损失：  0.17136491192015724
准确率：  0.955533917559234
```

# 实验结果

经过一系列的参数调整，最佳模型准确率达到 0.9555，模型为

```
class HyperSpectralModel(nn.Module):
    def __init__(self, num):
        super(HyperSpectralModel, self).__init__()
        self.output = nn.Sequential(
            nn.Linear(num, 250),
            nn.ReLU(True),
            nn.Linear(250, 200),
            nn.ReLU(True),
            nn.Linear(200, 150),
            nn.ReLU(True),
            nn.Dropout(0.5),
            nn.Linear(150, 100),
            nn.ReLU(True),
            nn.Linear(100, 50),
            nn.ReLU(True),
            nn.Linear(50, 16),
        )
```

## 附录：关键代码注释

K-L 变换求取前 N 大特征向量

```python
def getEigenVector(train_data, e):
    X = train_data - train_data.mean(axis=0)  # m*wh
    Y = np.dot(np.transpose(X), X) / X.shape[0]  # 样本矩阵降维，m*wh * wh*m -> m*m
    eigenvalue, feature_vector = np.linalg.eig(Y)  # 计算特征值和特征向量(得到的是列向量构成的特征矩阵)
    feature_vector = np.transpose(feature_vector)  # 将特征向量转置一下
    eigen = []
    for i in range(0, eigenvalue.shape[0]):  # 将特征值和特征向量两列一起，方便排序
        eigen.append([eigenvalue[i], feature_vector[i]])
    eigen = sorted(eigen, key=lambda eigen: eigen[0], reverse=True)  # 根据特征值大小，从大到小排序
    eigen_num = 0  # 根据重构精度确定特征指数目
    eigenvalue_sum = 0
    for i in range(0, eigenvalue.shape[0]):
        eigenvalue_sum += eigen[i][0]
        if eigenvalue_sum / eigenvalue.sum() >= e:
            eigen_num = i + 1
            break
    eigen_vector = []
    for i in range(0, eigen_num):
        vec = eigen[i][1]
        eigen_vector.append(vec)
    return eigen_num, np.array(eigen_vector)
```

```python
if __name__ == "__main__":
    data_path = './data/Indian_pines_corrected.mat'
    label_path = './data/Indian_pines_gt.mat'
    data = scio.loadmat(data_path)['indian_pines_corrected'].reshape(-1, 200)
    label = scio.loadmat(label_path)['indian_pines_gt'].flatten()
    train_datas, train_labels, test_datas, test_labels = Preprocesssing(data, label)
    e = 0.999
    eigen_num, eigen_vec = getEigenVector(train_datas, e)
    print('eigen_num:', eigen_num)
    json_file = {"e": e, "num": eigen_num, "vec": eigen_vec.tolist()}
    # 保存为json文件
    os.makedirs('./json_file', exist_ok=True)
    json_path = './json_file/feature_vector.json'  # 生成json存储路径
    # 存储为json文件
    with open(json_path, 'w') as file_json:
        json.dump(json_file, file_json, indent=2)  # 自动换行，缩进2个空格
        print('保存完毕！')
```

获取降维训练集和测试集

```python
class HyperSpectralDataset(Dataset):
    def __init__(self, data, label):
        self.data = np.array(data, dtype='float32')
        self.label = label

    def __getitem__(self, index):
        return torch.from_numpy(self.data[index]), torch.from_numpy(np.array(self.label[index]))

    def __len__(self):
        return len(self.label)


def Hyperspectral(data_path='./data/Indian_pines_corrected.mat', label_path='./data/Indian_pines_gt.mat'):
    data = scio.loadmat(data_path)['indian_pines_corrected'].reshape(-1, 200)
    label = scio.loadmat(label_path)['indian_pines_gt'].flatten()
    train_datas, train_labels, test_datas, test_labels = Preprocesssing(data, label)
    # 进行pca降维
    with open('./json_file/feature_vector.json', 'r') as file_json:
        file_python = json.load(file_json)
    e = file_python['e']
    num = file_python['num']
    eigen_vector = torch.tensor(file_python['vec'])
    print('e:', e)
    print('num:', num)
    train_datas = np.dot(train_datas, np.transpose(eigen_vector))
    test_datas = np.dot(test_datas, np.transpose(eigen_vector))
    train_dataset = HyperSpectralDataset(train_datas, train_labels)
    test_dataset = HyperSpectralDataset(test_datas, test_labels)
    return train_dataset, test_dataset, num
```

训练模型

```python
print("开始训练……")
for epoch in range(epochs):
    model.train()
    loss_train = []
    for index, (data, label) in enumerate(train_loader):
        data = data.to(device)
        label = label.to(device=device, dtype=torch.long)
        output = model(data)
        loss = lossFun(output, label)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        loss_train.append(loss.data.item())
        total_train_step += 1
        print('epoch:{}, batch:{}, total_train_step:{}, loss:{:.6f}'.format(epoch + 1, index + 1,
                                                                            total_train_step, loss.data.item()))
        torch.save(model.state_dict(), './models/model_' + str(epoch + 1) + '.pth')
        torch.save(optimizer.state_dict(), './models/model_optimizer_' + str(epoch + 1) + '.pth')
    model.eval()
    loss_test = []
    print('开始验证……')
    for index, (data, label) in enumerate(test_loader):
        data = data.to(device)
        label = label.to(device=device, dtype=torch.long)
        output = model(data)
        loss = lossFun(output, label)
        loss_test.append(loss.data.item())
    print('平均损失：', np.mean(loss_test))
    #记录损失
    loss_epochs_train.append(np.mean(loss_train))
    loss_epochs_test.append(np.mean(loss_test))
    writer.add_scalar("train_loss", np.mean(loss_train), epoch)
    writer.add_scalar("test_loss", np.mean(loss_test), epoch)
    epochs_x.append(epoch + 1)
```