搭建完整网络模型实现声纹识别

Reviewed	☐

实验报告

实验进展：

已完成基础部分和提高部分所有实验内容

实验思维导图

实验内容：

> **1）搭建网络**
>
> - 搭建适用于播音数据（2分类）的神经网络
>
> 基本任务：
>
> - transform变形图片，使其适应网络
> - CIFAR10的网络变成2分类
>
> 扩展任务一：自己写一个VGG11的网络
>
> 扩展任务二：调用预训练模型网络
>
> **2）完成康辉、海霞的声纹识别**
>
> - 搭建课上讲的模型，完成模型的训练，验证，并生成几个pth模型
> - 完成某个mel谱图输入后的测试

任务一：搭建网络

- 搭建适用于播音数据（2分类）的神经网络

基本任务1: transform变形图片，使其适应网络

transform是一个预处理的过程，主要是将输入的图像转换为适合模型输入的格式。

具体操作：修改MelSpectrogramDataset类，增加transform功能变形图片，使其适应网络。

```
class MelSpectrogramDataset(Dataset):
    '''
    该类封装了读取谱图的功能
    '''

    def __init__(self, root_dir, label_dir, transform=None):
        '''初始化类MelSpectrogramDataset'''
        self.root_dir = root_dir  # 音频谱图数据所在的根目录
        self.label_dir = label_dir  # 包含标签的子目录
        self.transform = transform  # 数据预处理的函数
        self.file_list = sorted(os.listdir(os.path.join(self.root_dir,self.label_dir)))  # 将root_dir和label_dir连接起来，组成文件夹的路径

    def __len__(self):
        '''返回谱图文件夹中文件的数量'''
        return len(self.file_list)

    def __getitem__(self, index):
        '''根据索引读取文件并返回文件和标签。'''
        #  获得完整路径，该路径指向该样本的数据文件
        file_path = os.path.join(self.root_dir,self.label_dir)
        file_path = os.path.join(file_path, self.file_list[index])
        #  将label_dir转换为整型，并将其作为该样本的标签。注意此处如果不处理则是元组类型而非tensor
        label = int(self.label_dir)
        image = cv2.imread(file_path)  # 读取图片
        if self.transform:
            image = Image.fromarray(image) # 为了后续transform处理，将numpy数组转换为PIL.Image格式
            image = self.transform(image)
            image = np.transpose(image, (1, 2, 0))
            '''
            在 PyTorch 中，CNN 模型的输入需要满足 (batch_size, channels, height, width) 的格式，
            而 transform 函数通常会将图像转换为 (height, width, channels) 的格式。
            因此，在返回结果之前，我们需要对图像数组的维度进行转换，以便与 CNN 模型的输入要求相匹配。
            '''

        return image, label  # 返回图片和标签
```

验证有效性：

```
root_dir1 = "Boyin_mel/train"
label_dir1 = '0'
root_dir2 = "Boyin_mel/val"
label_dir2 = '1'


# 无裁剪图片操作的情况 （不使用transform）
# 选取1个播音员的谱图文件夹，读取第一张谱图，显示图，打印图片的分辨率和标签，文件夹长度的读取
dataset = MelSpectrogramDataset(root_dir1,label_dir1)
image, label = dataset[0]  # 获取第一张图片和对应的标签
print(f"Image shape: {image.shape}, Label: {label}")  # 打印图片尺寸和标签
plt.imshow(image)  # 展示图片
plt.show()



# 裁剪图片操作的情况 （使用transform）
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 将图片的大小调整为 224x224
    transforms.ToTensor() # 将图片转换为 Tensor
])

# 选取1个播音员的谱图文件夹，读取第一张谱图，显示图，打印图片的分辨率和标签，文件夹长度的读取
dataset = MelSpectrogramDataset(root_dir1,label_dir1,transform=transform)
image, label = dataset[0]  # 获取第一张图片和对应的标签
print(f"Image shape: {image.shape}, Label: {label}")  # 打印图片尺寸和标签
```

```
plt.imshow(image)  # 展示图片
plt.show()
```

输出结果：

Image shape: (600, 1000, 3), Label: 0
Image shape: torch.Size([224, 224, 3]), Label: 0

上述transform过程就是将原始图像resize到适合VGG16模型输入的大小，并将其转换为张量，以便后续能够在模型中进行处理。具体来说，transforms.Resize将图片的大小调整为 224x224，因为VGG16模型的输入需要的图像大小为224x224。transforms.ToTensor将图像转换为张量。

基本任务2: CIFAR10的网络变成2分类

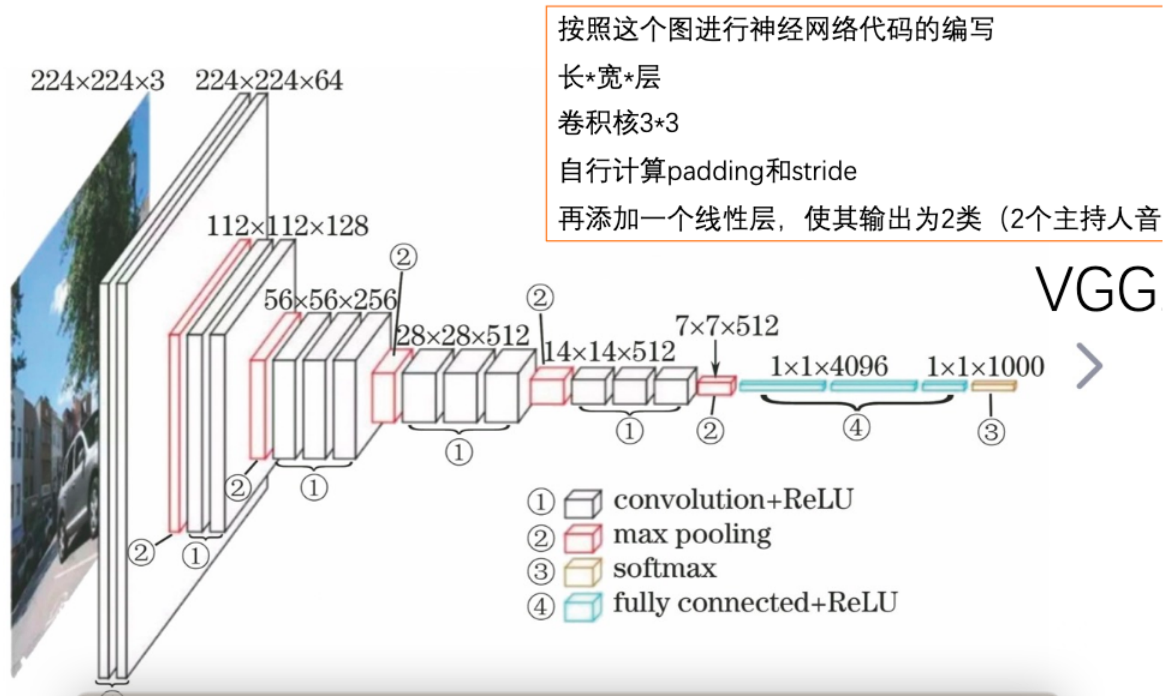
```
import torch
import torch.nn as nn
from torch.utils.tensorboard import SummaryWriter

class TuduiNet(nn.Module):
    def __init__(self):
        super(TuduiNet, self).__init__()
        self.model1 = nn.Sequential(
        nn.Conv2d(3, 32, 5, padding=2, stride=1),
        nn.MaxPool2d(2),
        nn.Conv2d(32, 32, 5, padding=2),
        nn.MaxPool2d(2),  # 最大池化层
        nn.Conv2d(32, 64, 5, padding=2),
        nn.MaxPool2d(2),  # 最大池化层
        nn.Flatten(),
        nn.Linear(1024, 64),
        nn.Linear(64, 2),  # 2分类网络
    )

    def forward(self, x):
        x = self.model1(x)
        return x

# 测试TuduiNet网络的构建是否正确
if __name__ == '__main__':
    tudui =TuduiNet()  # 实例化 TuduiNet 模型
    print(tudui)  # 打印模型结构
    input = torch.ones((64,3,32,32))  # 创建一个输入张量
    output = tudui(input)  # 将输入张量传入模型进行前向传播
    # print(output.shape)  # 打印输出张量的形状

    writer = SummaryWriter('logs')  # 实例化一个 SummaryWriter 对象，用于将日志写入 TensorBoard
    writer.add_graph(tudui, input)  # 将模型结构写入 TensorBoard
    writer.close()  # 关闭 SummaryWriter
```

网络结构：

TuduiNet(
(model1): Sequential(
(0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(4): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(6): Flatten(start_dim=1, end_dim=-1)
(7): Linear(in_features=1024, out_features=64, bias=True)
(8): Linear(in_features=64, out_features=2, bias=True))
)

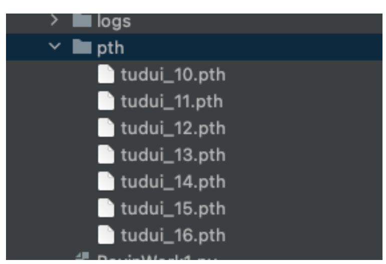
扩展任务一：自己写一个VGG16的网络

按照下图自己写一个VGG16的网络




```
import torch
import torch.nn as n

class VGG16(nn.Module):
    def __init__(self):
        super(VGG16, self).__init__()

        # 定义卷积层部分
        self.conv_layers = nn.Sequential(
            # 第1个卷积层：输入通道为3，输出通道为64，卷积核大小为3，填充为1，添加BatchNorm层和ReLU激活函数，后面紧跟一个最大池化层，池化核大小为2，步长为2。
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),  # 添加BatchNorm2d，通过对每个小批量的数据在每个神经元的输出上做归一化，可以加速网络的训练，并且使得网络对初始参数的选择更加鲁
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # 第2个卷积层：输入通道为64，输出通道为128，卷积核大小为3，填充为1，添加BatchNorm层和ReLU激活函数，后面紧跟一个最大池化层，池化核大小为2，步长为2。
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # 第3个卷积层：输入通道为128，输出通道为256，卷积核大小为3，填充为1，添加BatchNorm层和ReLU激活函数，后面紧跟一个最大池化层，池化核大小为2，步长为2。
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
```

```
            # 第4个卷积层：输入通道为256，输出通道为512，卷积核大小为3，填充为1，添加BatchNorm层和ReLU激活函数，后面紧跟一个最大池化层，池化核大小为2，步长为2。
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # 第5个卷积层：输入通道为512，输出通道为512，卷积核大小为3，填充为1，添加BatchNorm层和ReLU激活函数，后面紧跟一个最大池化层，池化核大小为2，步长为2。
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),  # 添加BatchNorm层
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )


        # 全连接层：三个线性层，中间添加了ReLU激活函数和Dropout正则化。
        self.fc_layers = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096), # 线性层

            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096), # 线性层

            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 2), # 线性层

        )

    # 前向传播函数
    def forward(self, x):
        x = self.conv_layers(x) # 输入x通过卷积层部分得到特征图
        x = x.reshape(x.size(0), -1) # 压缩成1维向量
        x = self.fc_layers(x) # 通过全连接层输出
        return x

vgg16=VGG16() # 创建了VGG16模型的实例
print(vgg16)
image=torch.randn(1,3,224,224) # 创建一张随机的输入图像
vgg16.eval() # 将模型设置为评估模式，并关闭梯度计算
with torch.no_grad(): # 对输入图像进行推理，并输出模型的预测结果
    output=vgg16(image)
print(output) # 输出模型的预测结果
print(output.argmax(1)) # 输出预测结果的类别
```



输出结果：

网络结构：

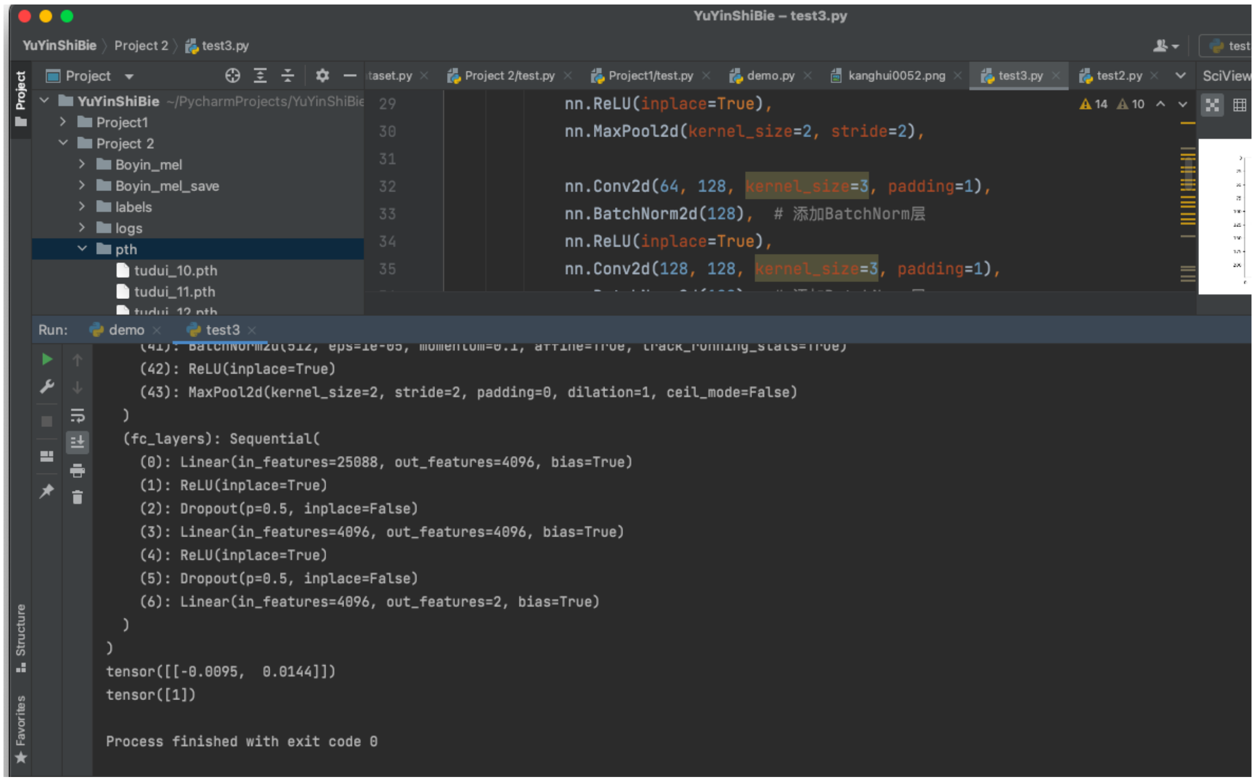
VGG16(
(conv_layers): Sequential(
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

```
(2): ReLU(inplace=True)
(3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(9): ReLU(inplace=True)
(10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(12): ReLU(inplace=True)
(13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(16): ReLU(inplace=True)
(17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(19): ReLU(inplace=True)
(20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(22): ReLU(inplace=True)
(23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(24): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(26): ReLU(inplace=True)
(27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(29): ReLU(inplace=True)
(30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(32): ReLU(inplace=True)
(33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(36): ReLU(inplace=True)
(37): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(39): ReLU(inplace=True)
(40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(42): ReLU(inplace=True)
(43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(fc_layers): Sequential(
(0): Linear(in_features=25088, out_features=4096, bias=True)
(1): ReLU(inplace=True)
(2): Dropout(p=0.5, inplace=False)
(3): Linear(in_features=4096, out_features=4096, bias=True)
(4): ReLU(inplace=True)
(5): Dropout(p=0.5, inplace=False)
(6): Linear(in_features=4096, out_features=2, bias=True)
))
```

模型的预测结果：
tensor([[-0.0095, 0.0144]])

预测结果的类别：
tensor([1])

运行截图：



扩展任务二：调用预训练模型网络

方法**1、调用VGG16预训练模型网络，并添加了一个额外的线性层作为分类器**

```
import torch
import torch.nn as nn
import torchvision
#在数据集上取得比较好的效果的网络参数
vgg16_true=torchvision.models.vgg16(pretrained=True)
#添加一层add_module()：
vgg16_true.classifier.add_module('add_linear',nn.Linear(1000,2)) #名称,神经网络线性层(放在classifier层中），从1000变到10
#可以打印模型，看一下模型结构
print(vgg16_true)

image=torch.randn(1,3,224,224) # 创建一张随机的输入图像
vgg16_true.eval() # 将模型设置为评估模式，并关闭梯度计算
with torch.no_grad(): # 对输入图像进行推理，并输出模型的预测结果
    output=vgg16_true(image)
print(output) # 输出模型的预测结果
print(output.argmax(1)) # 输出预测结果的类别
```

输出结果：

网络结构：

VGG(
(features): Sequential(

```
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU(inplace=True)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU(inplace=True)
(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(6): ReLU(inplace=True)
(7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): ReLU(inplace=True)
(9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU(inplace=True)
(12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(13): ReLU(inplace=True)
(14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): ReLU(inplace=True)
(16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(18): ReLU(inplace=True)
(19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(20): ReLU(inplace=True)
(21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(22): ReLU(inplace=True)
(23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(25): ReLU(inplace=True)
(26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(27): ReLU(inplace=True)
(28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29): ReLU(inplace=True)
(30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
(classifier): Sequential(
(0): Linear(in_features=25088, out_features=4096, bias=True)
(1): ReLU(inplace=True)
(2): Dropout(p=0.5, inplace=False)
(3): Linear(in_features=4096, out_features=4096, bias=True)
(4): ReLU(inplace=True)
(5): Dropout(p=0.5, inplace=False)
(6): Linear(in_features=4096, out_features=1000, bias=True)
(add_linear): Linear(in_features=1000, out_features=2, bias=True)
))
```

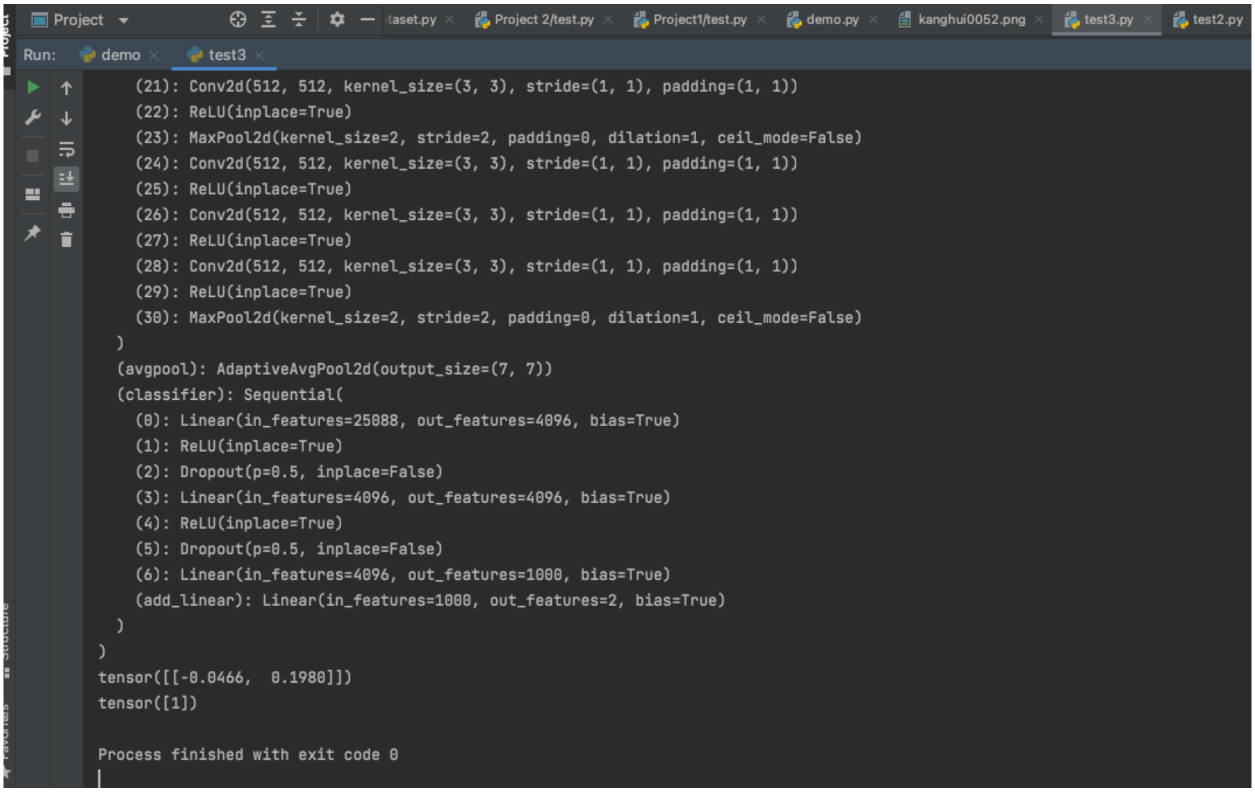
模型的预测结果：

```
tensor([[-0.0466,  0.1980]])
```

预测结果的类别：

```
tensor([1])
```

运行截图：



方法2、直接修改预训练网络最后一层，也就是classifier的[6]层，使模型不会输出1000类，而是输出2类

```
import torch
import torch.nn as nn
import torchvision

vgg16_true=torchvision.models.vgg16(pretrained=True)
vgg16_true.classifier[6]=nn.Linear(4096,2) # 直接修改预训练网络最后一层，使模型不会输出1000类，而是输出2类
print(vgg16_true) #可以打印模型，看一下模型结构

image=torch.randn(1,3,224,224) # 创建一张随机的输入图像
vgg16_true.eval() # 将模型设置为评估模式，并关闭梯度计算
with torch.no_grad(): # 对输入图像进行推理，并输出模型的预测结果
    output=vgg16_true(image)
print(output) # 输出模型的预测结果
print(output.argmax(1)) # 输出预测结果的类别
```

输出结果：

网络结构：

VGG(
(features): Sequential(
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU(inplace=True)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU(inplace=True)
(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(6): ReLU(inplace=True)
(7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): ReLU(inplace=True)

```
(9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU(inplace=True)
(12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(13): ReLU(inplace=True)
(14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): ReLU(inplace=True)
(16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(18): ReLU(inplace=True)
(19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(20): ReLU(inplace=True)
(21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(22): ReLU(inplace=True)
(23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(25): ReLU(inplace=True)
(26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(27): ReLU(inplace=True)
(28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29): ReLU(inplace=True)
(30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
(classifier): Sequential(
(0): Linear(in_features=25088, out_features=4096, bias=True)
(1): ReLU(inplace=True)
(2): Dropout(p=0.5, inplace=False)
(3): Linear(in_features=4096, out_features=4096, bias=True)
(4): ReLU(inplace=True)
(5): Dropout(p=0.5, inplace=False)
(6): Linear(in_features=4096, out_features=2, bias=True)
))
```

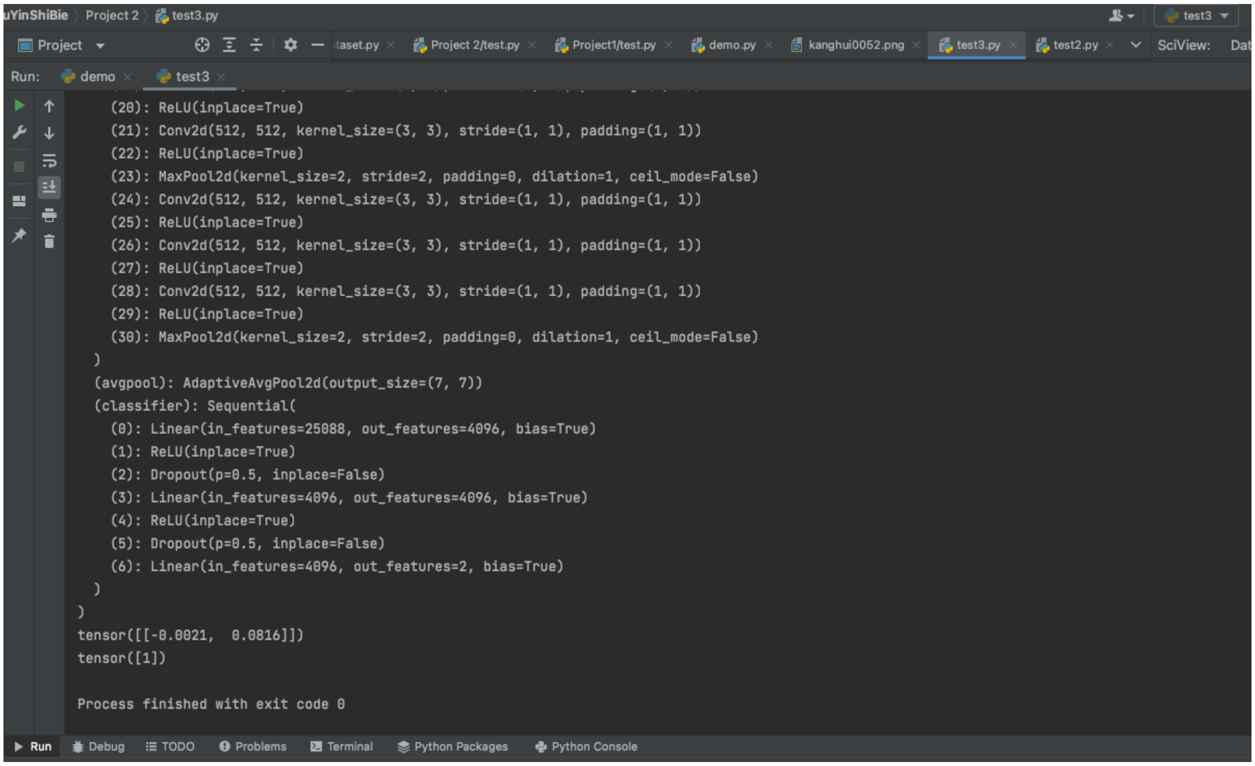
模型的预测结果：

```
tensor([[-0.0021,  0.0816]])
```

预测结果的类别：

```
tensor([1])
```

运行截图：



任务二：完成康辉、海霞的声纹识别

- **对刚刚搭建的讲义中的vgg16模型，完成模型的训练，验证，并生成几个pth模型**

```
# 创建变换函数
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 将图片的大小调整为 224x224
    transforms.ToTensor() # 将图片转换为 Tensor
])

# 加载数据集
root_dir1 = "Boyin_mel/train"
label_dir1 = '0'
root_dir2 = "Boyin_mel/val"
label_dir2 = '1'

# 定义数据集路径和目录
kanghui_train_dataset = MelSpectrogramDataset(root_dir1, label_dir2, transform=transform)
haixia_train_dataset = MelSpectrogramDataset(root_dir1, label_dir1, transform=transform)
kanghui_val_dataset = MelSpectrogramDataset(root_dir2, label_dir2, transform=transform)
haixia_val_dataset = MelSpectrogramDataset(root_dir2, label_dir1, transform=transform)

train_dataset = kanghui_train_dataset + haixia_train_dataset
val_dataset = kanghui_val_dataset + haixia_val_dataset

# 测试数据集有效性
test_data_size = len(val_dataset)
image, label = train_dataset[0]  # 获取第一张图片和对应的标签
print(f"Image shape: {image.shape}, Label: {label}")  # 打印图片尺寸和标签
print("Tensor shape:", transform(Image.fromarray(np.uint8(image))).shape)
print(f" kanghui_trainset length: {len(kanghui_train_dataset)}, kanghui_validationset length: {len(kanghui_val_dataset)}")
print(f" haixia_trainset length: {len(kanghui_train_dataset)}, haixia_validationset length: {len(kanghui_val_dataset)}")
print(f" trainset length: {len(train_dataset)}, validationset length: {test_data_size}")
```

输出结果：

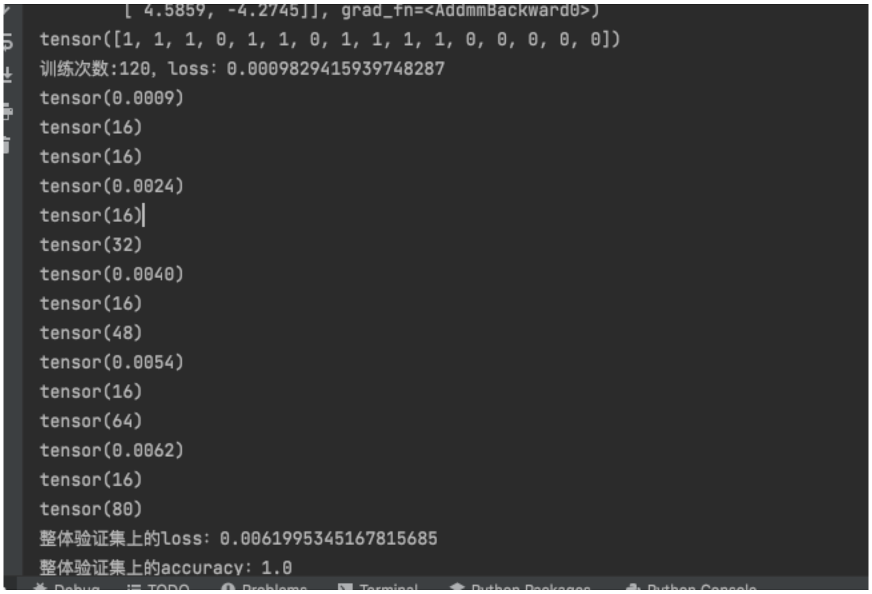
Image shape: torch.Size([224, 224, 3]), Label: 1
Tensor shape: torch.Size([3, 224, 224])
kanghui_trainset length: 160, kanghui_validationset length: 40
haixia_trainset length: 160, haixia_validationset length: 40
trainset length: 320, validationset length: 80

表明数据集构建成功。

训练和测试模型：

```
# 定义训练参数
batch_size = 16
num_epochs = 6
learning_rate = 0.01

# 创建训练集和验证集的 DataLoader
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)


# 损失函数
loss = nn.CrossEntropyLoss()
# 创建随机梯度下降法优化器（神经网络参数，学习率）
optim = torch.optim.SGD(vgg16.parameters(), lr=learning_rate)

# 记录训练次数
total_train_step=0
# 记录测试次数
total_test_step=0

# 训练模型
for i in range(num_epochs): # 对于每个 epoch，循环训练模型
    print('-----第{}轮训练开始-----'.format(i+1))
    for data in train_loader: # 遍历训练集 train_loader，依次读取每个数据样本 data
        imgs, targets = data # 从 data 中分别取出图像数据 imgs 和标签数据 targets
        # 将 imgs 转换成 float 类型，并按照 CWH 的格式重新排列通道，
        # 使得每个数据的形状为 (batch_size, 3, 224, 224)
        imgs = imgs.float()
        imgs=imgs.permute(0, 3, 1, 2)
        # print(imgs.shape)

        outputs = vgg16(imgs) # 通过调用 vgg16 模型的 forward 方法，传入 imgs，得到输出结果 outputs
        print(outputs)
        print(targets)

        result_loss = loss(outputs, targets) # 计算输出结果 outputs 和标签数据 targets 之间的损失函数值 result_loss
        optim.zero_grad() # 调用优化器 optim 来更新模型参数
        result_loss.backward()
        optim.step()
        total_train_step+=1
        print("训练次数:{}, loss：{}".format(total_train_step,result_loss)) #打印训练次数 total_train_step 和损失函数值 result_loss

    torch.save(vgg16,'pth/tudui_{}.pth'.format(i+10)) # 在每轮结束后，保存训练好的模型，以便后续使用

# 测试步骤开始
    total_test_loss = 0
    total_accuracy = 0
    with torch.no_grad():
        for data in test_loader: # 遍历测试集 train_loader，依次读取每个数据样本 data
            imgs, targets = data # 从 data 中分别取出图像数据 imgs 和标签数据 targets
            # 将 imgs 转换成 float 类型，并按照 CWH 的格式重新排列通道，
            # 使得每个数据的形状为 (batch_size, 3, 224, 224)
            imgs = imgs.float()
            imgs = imgs.permute(0, 3, 1, 2)

            outputs = vgg16(imgs) # 使用训练好的模型对输入图片进行预测。
            result_loss = loss(outputs, targets) # 计算模型预测值和实际标签之间的loss
            total_test_loss = total_test_loss + result_loss # 累加当前批次的loss值到总loss中
            print(total_test_loss)
            accuracy = (outputs.argmax(1) == targets).sum() # 计算当前批次中预测正确的样本数。
            print(accuracy)
            total_accuracy = total_accuracy + accuracy # 累加当前批次中预测正确的样本数到总数中。
            print(total_accuracy)

    # 打印出整个测试集上的loss和accuracy
```

```
    print('整体验证集上的loss：{}'.format(total_test_loss))
    print('整体验证集上的accuracy：{}'.format(total_accuracy / test_data_size))
```



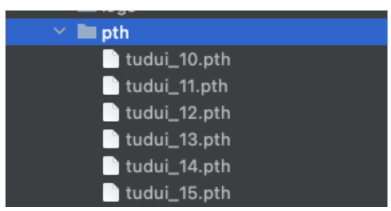
输出结果：

训练次数:120，loss：0.0009829415939748287

整体验证集上的loss：0.0061995345167815685
整体验证集上的accuracy：1.0

从输出结果可以看出，该代码训练了120个epoch，在训练集上表现良好，整体验证集上的loss和accuracy也表现很好，loss仅为0.0062，accuracy为1.0，说明该模型在验证集上的表现非常好。因此，可以认为该模型在分类任务上具有很高的准确性和鲁棒性。

在每轮结束后，保存训练好的模型如下：



注意，进行了6个num_epoch，命名的时候i+10：`torch.save(vgg16,'pth/tudui_{}.pth'.format(i+10))`，因此最新的为tudui_15.pth。

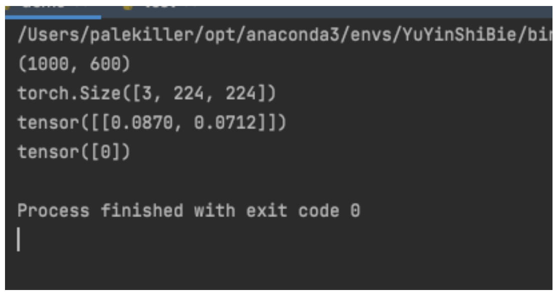
该模型在后续任务中使用。

- **完成某个mel谱图输入后的测试**

分别输入海霞和康辉的mel谱图测试，注意康辉的mel谱图的label设置为0.，海霞为1.

```
file_path = 'kanghui.png'
image = Image.open(file_path)
print(image.size)
image = image.convert('RGB')
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 将图片的大小调整为 224x224
    transforms.ToTensor() # 将图片转换为 Tensor
])
```

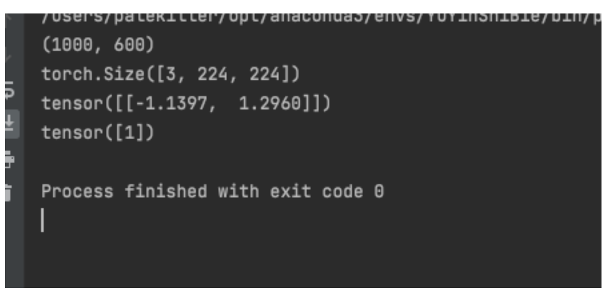
```
image = transform(image)
print(image.shape)

model = torch.load('pth/tudui_15.pth')
#print(model)
image=torch.reshape(image,(1,3,224,224))
model.eval()
with torch.no_grad():
    output=model(image)
print(output)
print(output.argmax(1))
```



(1000, 600)
torch.Size([3, 224, 224])
tensor([[0.0870, 0.0712]])
tensor([0])

output.argmax(1)为0，是康辉的label，表明康辉mel谱图分类正确

```
file_path = 'haixia.png'
image = Image.open(file_path)
print(image.size)
image = image.convert('RGB')
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 将图片的大小调整为 224x224
    transforms.ToTensor() # 将图片转换为 Tensor
])
image = transform(image)
print(image.shape)

model = torch.load('pth/tudui_15.pth')
#print(model)
image=torch.reshape(image,(1,3,224,224))
model.eval()
with torch.no_grad():
    output=model(image)
print(output)
print(output.argmax(1))
```



(1000, 600)
torch.Size([3, 224, 224])

tensor([[-1.1397, 1.2960]])
tensor([1])

output.argmax(1)为1，是海霞的label，表明海霞mel谱图分类正确

实验遇到的问题及解决办法：

1、处理数据集图片后的格式与网络模型输入要求不匹配。

在 PyTorch 中，CNN 模型的输入需要满足 (batch_size, channels, height, width) 的格式，
而 transform 函数通常会将图像转换为 (height, width, channels) 的格式。
因此，在返回结果之前，我们需要对图像数组的维度进行转换，以便与 CNN 模型的输入要求相匹配。

image = np.transpose(image, (1, 2, 0))

2、训练自己的vgg16模型出现梯度爆炸。

在网络中添加batchnorm层。

3、训练速度太慢

调整epoch_num和batchsize。

4、在class MelSpectrogramDataset(Dataset)，需要image = Image.fromarray(image) 再进行transform处理，将numpy数组转换为PIL.Image格式。

5、在class MelSpectrogramDataset(Dataset)，需要label = int(self.label_dir), 将label_dir转换为整型，并将其作为该样本的标签。注意此处如果不处理则是元组类型而非tensor。