HW6 Report

Models

VGG

参考 pdf 中的"VGG16 architecture"，类比构建了输入层--卷积层+ReLU--进过2~3个卷积层的池化--全连接层的结构，即：

```
self.conv_layers = nn.Sequential(
    nn.Conv2d(3, 64, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(64, 64, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2, 2),

    nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(128, 128, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2, 2),

    nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(256, 256, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(256, 256, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2, 2),

    nn.Conv2d(256, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2, 2),

    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2, 2),
        )
 self.fc_layers = nn.Sequential(
    nn.Linear(512, 256), nn.ReLU(inplace=True),
    nn.Linear(256, 128), nn.ReLU(inplace=True),
    nn.Linear(128, 10)
        )
```

ResNet

在 `ResBlock` 类实现了basic residual block, 包含两层3x3卷积且每层后都接有BatchNorm 和 ReLU；主分支完成两次卷积变换，捷径分支在输入输出通道数或空间尺寸不一致时通过1x1卷积和BatchNorm进行调整。最终，主分支和捷径分支的输出相加后再经过ReLU实现了残差连接。

```
class ResBlock(nn.Module):
    def __init__(self, in_channel, out_channel, stride):
        super().__init__()
        self.conv1=nn.Conv2d(in_channel, out_channel, kernel_size=3, stride=stride,
padding=1, bias=False)
        self.bn1=nn.BatchNorm2d(out_channel)
        self.conv2=nn.Conv2d(out_channel, out_channel, kernel_size=3, stride=1, padding=1,
bias=False)
        self.bn2=nn.BatchNorm2d(out_channel)
        if in_channel!=out_channel or stride!=1:
            self.shortcut=nn.Sequential(
                nn.Conv2d(in_channel, out_channel, kernel_size=1, stride=stride,
bias=False),
                nn.BatchNorm2d(out_channel)
            )

    def forward(self, x: torch.Tensor):
        identity=x
        out=self.conv1(x)
        out=self.bn1(out)
        out=F.relu(out)
        out=self.conv2(out)
        out=self.bn2(out)
        if hasattr(self,'shortcut'):
            identity=self.shortcut(x)
        out+=identity
        out=F.relu(out)
        return out
```

然后对于ResNet，构建一个较深层的结构，主干由3大部分组成：

1. layer 1: 7个ResBlock，输入输出通道均为16，空间尺寸保持不变;
2. layer 2: 以stride=2的ResBlock开始（实现下采样和通道扩展16→32），后面6个block通道为32，空间尺寸不变;
3. layer 3: 同理，先用stride=2的ResBlock将通道扩展到64，空间尺寸减半，后面6个block通道为64。

堆叠后特征图通过全局平均池化 `nn.AdaptiveAvgPool2d(1)`，压缩为每通道1个数。再加入 `nn.Dropout(0.2)` 进行正则化防止过拟合，最后通过fc `nn.Linear(64, 10)` 输出。

即：

```
class ResNet(nn.Module):
    '''residual network'''
    def __init__(self):
        super().__init__()
```

```
        self.conv=nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn=nn.BatchNorm2d(16)
        self.layer1=nn.Sequential(*[ResBlock(16,16,1) for _ in range(7)])
        self.layer2 = nn.Sequential(
             ResBlock(16, 32, 2),
            *[ResBlock(32, 32, 1) for _ in range(6)]
        )
        self.layer3 = nn.Sequential(
            ResBlock(32, 64, 2),
            *[ResBlock(64, 64, 1) for _ in range(6)]
        )
        self.avpool = nn.AdaptiveAvgPool2d(1)
        self.drop= nn.Dropout(0.2)
        self.fc = nn.Linear(64, 10)
    def forward(self, x: torch.Tensor):
        x = self.conv(x)
        x = self.bn(x)
        x = F.relu(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)

        x = self.avpool(x)
        x = x.view(x.size(0), -1)
        x = self.drop(x)
        out = self.fc(x)
        return out
```

ResNeXt

采用 bottleneck+group 卷积结构，每个 ResNextBlock 由三层卷积组成： 1x1 卷积降维，接着用 3x3 group cnvolution 进行特征提取，最后再用 1x1 卷积恢复通道数，主分支完成上述三次卷积变换。shortcut在输入输出通道数或空间尺寸不一致时通过 1x1 卷积和 BatchNorm 进行调整，否则直接传递输入。主分支和捷径分支的输出相加后再经过 ReLU，实现了残差连接。

```
class ResNextBlock(nn.Module):
    def __init__(self, in_channel, out_channel, bottle_neck, group, stride):
        super().__init__()
        hidden_channel = int(out_channel / bottle_neck)
        self.block = nn.Sequential(
            nn.Conv2d(in_channel, hidden_channel, kernel_size=1, stride=1, bias=False),
            nn.BatchNorm2d(hidden_channel),
            nn.ReLU(inplace=True),
            nn.Conv2d(hidden_channel, hidden_channel, kernel_size=3, stride=stride,
padding=1, groups=group, bias=False),
            nn.BatchNorm2d(hidden_channel),
            nn.ReLU(inplace=True),
            nn.Conv2d(hidden_channel, out_channel, kernel_size=1, stride=1, bias=False),
            nn.BatchNorm2d(out_channel)
        )
```

```
        if in_channel != out_channel or stride != 1:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channel, out_channel, kernel_size=1, stride=stride,
bias=False),
                nn.BatchNorm2d(out_channel)
            )
    def forward(self, x):
        identity = x
        if hasattr(self, 'shortcut'):
            identity = self.shortcut(x)
        out = self.block(x)
        out += identity
        out = F.relu(out)
        return out
```

对于 ResNeXt 主体网络，结构与 ResNet 类似但每个 stage 由 ResNextBlock 组成，也加入Dropout 和 Global Average Pooling, 即：

```
class ResNext(nn.Module):
    def __init__(self):
        super().__init__()
        bottleneck = 4
        group = 8
        num_blocks = [3, 3, 3]
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            self._make_stage(64, 64, num_blocks[0], bottleneck, group, stride=1),
            self._make_stage(64, 128, num_blocks[1], bottleneck, group, stride=2),
            self._make_stage(128, 256, num_blocks[2], bottleneck, group, stride=2)
        )
        self.avpool = nn.AdaptiveAvgPool2d(1)
        self.drop = nn.Dropout(0.2)
        self.fc = nn.Linear(256, 10)

    def _make_stage(self, in_channel, out_channel, num_blocks, bottleneck, group, stride):
        layers = []
        layers.append(ResNextBlock(in_channel, out_channel, bottleneck, group, stride))
        for _ in range(1, num_blocks):
            layers.append(ResNextBlock(out_channel, out_channel, bottleneck, group,
stride=1))
        return nn.Sequential(*layers)

    def forward(self, x: torch.Tensor):
        x = self.features(x)
        x=self.avpool(x)
        x = x.view(x.size(0), -1)
        x=self.drop(x)
        out = self.fc(x)
        return out
```

训练过程

结合上次 HW5 对optimizer 和 scheduler的分析，沿用在作业五中的组合选择：

```
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.07)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=40, eta_min=0.0)
```

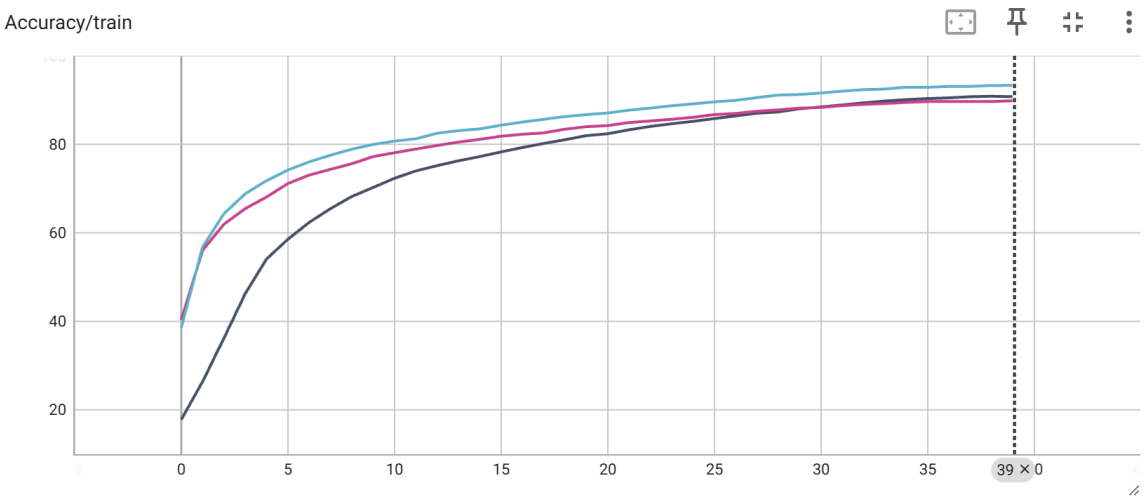
初始加载数据时，对训练集的数据做数据增强，加入颜色调、随机旋转、随机翻转：

```
transform_train = transforms.Compose([
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
        transforms.RandomRotation(15),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

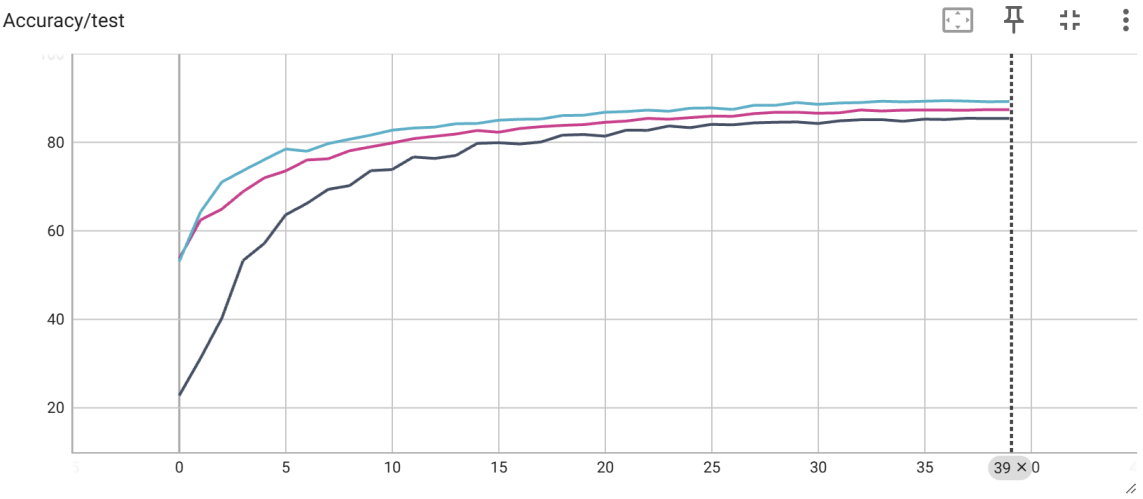
其余保存训练、测试、加载checkpoint、记录每轮的acc loss等过程类比hw5，即可正常实现。

结果对比

在训练集和测试集上每轮的正确率：

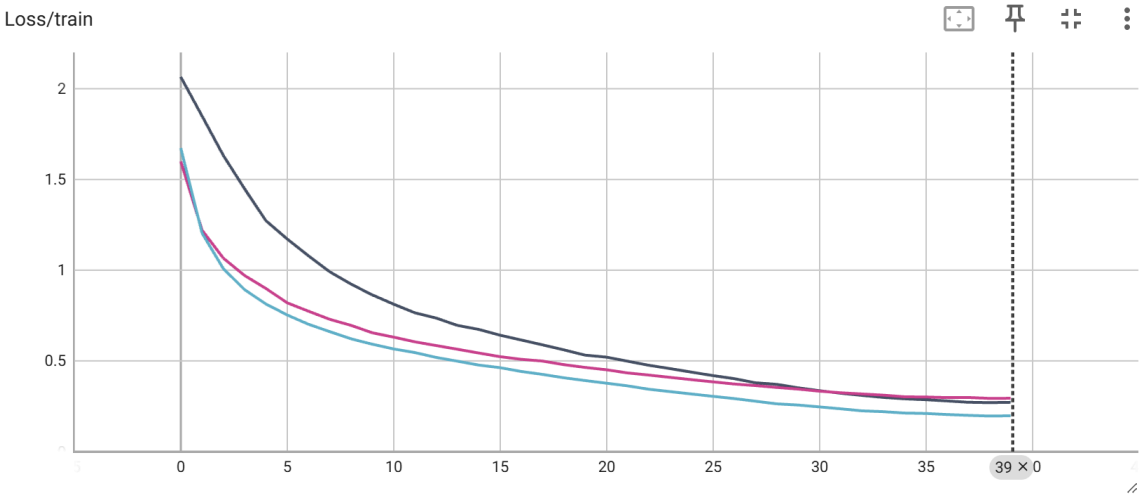


Run ↑	Value	Step	Relative
resnet_adamw_cosine	93.302	39	8.579 min
resnext_adamw_cosine	89.824	39	5.999 min
vgg_adamw_cosine	90.748	39	5.919 min

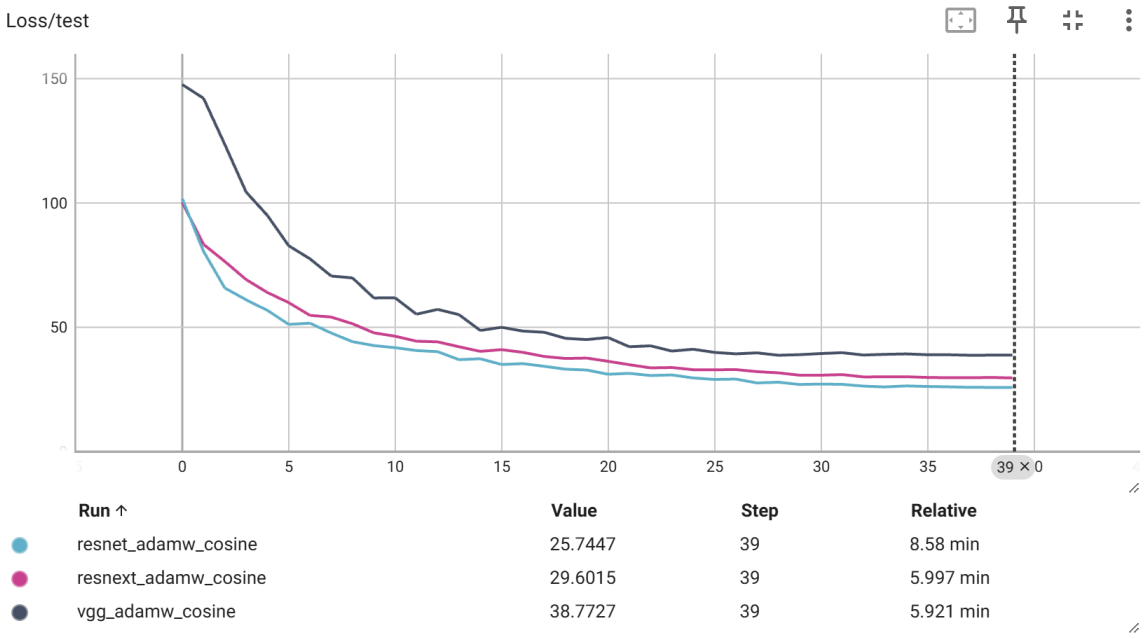


Run ↑	Value	Step	Relative
resnet_adamw_cosine	89.21	39	8.58 min
resnext_adamw_cosine	87.35	39	5.997 min
vgg_adamw_cosine	85.38	39	5.921 min

在训练集和测试集上每轮的loss：



Run ↑	Value	Step	Relative
resnet_adamw_cosine	0.1976	39	8.579 min
resnext_adamw_cosine	0.294	39	5.999 min
vgg_adamw_cosine	0.2717	39	5.919 min



Run ↑	Value	Step	Relative
resnet_adamw_cosine	25.7447	39	8.58 min
resnext_adamw_cosine	29.6015	39	5.997 min
vgg_adamw_cosine	38.7727	39	5.921 min

可见，三种卷积神经网络都有较快的收敛速度和较好的最终效果，总体上在相同epoch和学习率等参数的情况下，vgg效果略逊色于另外两个模型。最终在测试集上的test正确率与终端输出为： **VGG--85.37% ResNet--89.82% ResNeXt--88.17%**

```
PS E:\cv\Problem_Set_5\hw5> python main.py --model vgg --run test
Test Loss: 76.7590, Test Accuracy: 85.3700
PS E:\cv\Problem_Set_5\hw5> python main.py --model resnet --run test
Test Loss: 47.7961, Test Accuracy: 89.8200
PS E:\cv\Problem_Set_5\hw5> python main.py --model resnext --run test
Test Loss: 54.9678, Test Accuracy: 88.1700
```

（权重文件：`checkpoint_epoch_40_VGG_2.0.pth`、`checkpoint_epoch_40_ResNet_2.0.pth`、`checkpoint_epoch_40_ResNeXt_2.0.pth`，分别在命令行调用不同的model即可加载各自保存的参数值，得到上述终端截图的输出正确率）