****基于卷积神经网络的图像识别任务

院 系： 计算机科学与工程学院

专 业： 计算机科学与技术

姓 名： 李新泽

学 号：   2201823

教 师： 肖桐

2022年12月

1. 实验目的

1998年，Yann LeCun提出了首个卷积神经网络模型LeNet；

2012年Alex Krizhevsky、Geoff Hinton和llya Sutskever提出的AlexNet模型在ILSVRC挑战赛上一骑绝尘；

2015年Kaiming He、Xiangyu Zhang、Shaoqing Ren和Jiangxi Sun提出的ResNet模型在分类任务上第一次超越人类；

自此，掀起了卷积神经网络的研究热潮，一时间，CNN在计算机视觉、自然语言处理等领域内可算得上是风头无两。

故，通过此次自然语言处理大作业，对三种经典CNN模型进行深入学习。

1. 模型简介
2. 数据集

LeNet作为CNN的开山之作，结构较为简单，在小规模的问题上取得过当时世界最好结果，但是在大规模图像上效果不好，故本次实验选择CIFAR-10作为LeNet的数据集。

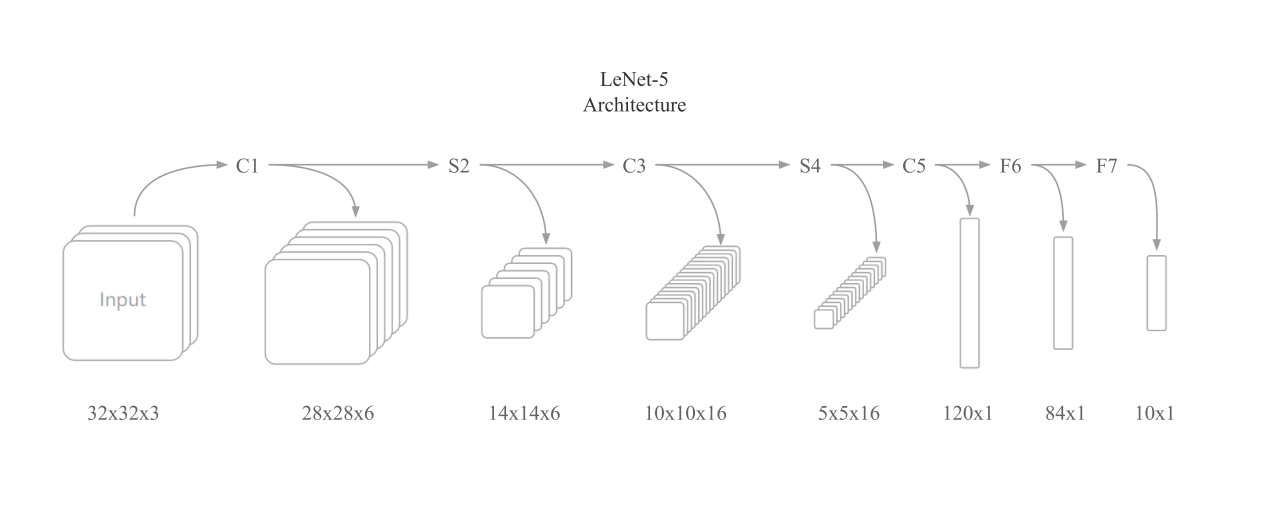
CIFAR-10数据集是由Alex Krizhevsky和 llya Sutskever整理的一个用于识别普适物体的小型数据集，一共包含10个类别的RGB彩色图片：飞机 (airplane)、汽车 (automobile)、鸟类 (bird)、猫 (cat)、鹿 (deer)、狗 (dog)、蛙类 (frog)、马 (horse)、船 (ship)和卡车 (truck)；图片尺寸为32x32像素，包含5万张图片的训练集和1万张图片的测试集。

AlexNet和ResNet作为ILSVRC挑战赛的获奖模型，理应采用比赛数据集，但是受限于个人硬件资源，无法在短时间内完成训练，故选择ImageNet的子数据集Mini-ImageNet。

Mini-ImageNet数据集是由Google DeepMind团队在ImageNet的基础上制作的小型数据集，一共包含100个类别的RGB彩色图片：知更鸟 (robin)、巨嘴鸟 (toucan)、鹅 (goose)、水母 (jellyfish)、热狗 (hotdog)、斑点狗 (dalmatian)等等；图像尺寸不固定，训练、验证、测试集共包含6万张图片，实际使用时需要对数据集进行预处理。

1. LeNet模型

LeNet是一个7层的神经网络，包含3个卷积层，2个池化层，2个全连接层，具体结构如下图所示。



细节参数如下：

·C1：卷积层，num\_kernels=6, kernel\_size=5×5, stride=1, padding=0

·S2：均值池化层，kernel\_size=2×2, stride=2, padding=0

·C3：卷积层，num\_kernels=16, kernel\_size=5×5, stride=1, padding=0

·S4：均值池化层，kernel\_size=2×2, stride=2, padding=0

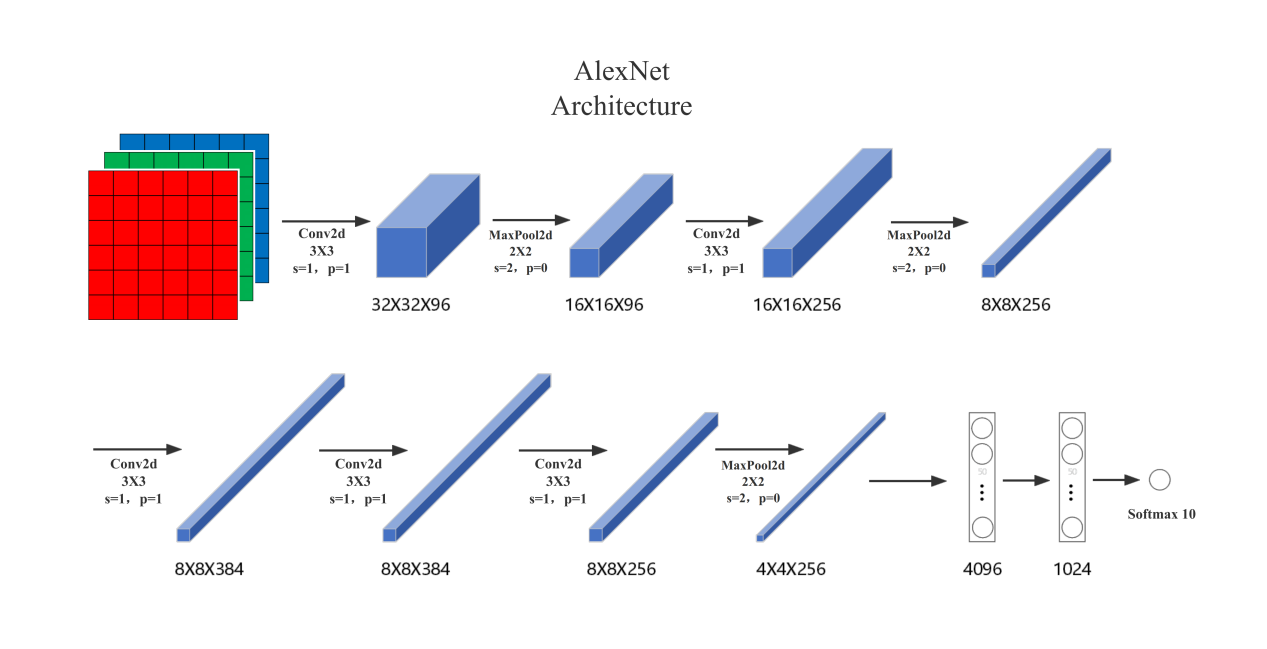
·C5：卷积层，num\_kernels=120,kernel\_size=5×5, stride=1, padding=0

·F6：全连接层，out\_features=84

·F7：全连接层，out\_features=10

1. AlexNet模型

AlexNet是一个8层的神经网络，包含5个卷积层，3个全连接层，具体结构如下图所示。



但网络结构并不是AlexNet成功的关键，一些新的方法和技术在卷积神经网络中的应用使得AlexNet不同于LeNet等之前的卷积模型：

·ReLU(Rectified Linear Units)：AlexNet使用ReLU而不是tanh作为激活函数，这使得卷积网络在训练时间上大大减少；

·Local Response Normalization(LRN)：受生物神经元启发，在ReLU层后添加LRN增加网络的泛化能力；



·DropOut：以预定的概率关闭神经元，每一次迭代都使用不同的神经元参与训练，这迫使每个神经元具有更健壮的特征；

·Multiple GPUs：通过将模型的神经元分别放在两个GPU上进行训练，在硬件资源受限的时代训练更大的模型；

细节参数如下：

·卷积层C1

·卷积：num\_kernels=96, kernel\_size=11×11, stride=4, padding=0

·ReLU：将卷积结果输入到ReLU函数，不改变特征图参数

·局部响应归一化：使用LRN函数进行局部归一化，不改变特征图参数

·池化：kernel\_size=3×3, stride=2, padding=0,

·卷积层C2

·卷积：num\_kernels=256, kernel\_size=5×5, stride=1, padding=2

·ReLU：将卷积结果输入到ReLU函数，不改变特征图参数

·局部响应归一化：使用LRN函数进行局部归一化，不改变特征图参数

·池化：kernel\_size=3×3, stride=2, padding=0,

·卷积层C3

·卷积：num\_kernels=384, kernel\_size=3×3, stride=1, padding=1

·ReLU：将卷积结果输入到ReLU函数，不改变特征图参数

·卷积层C4

·卷积：num\_kernels=384, kernel\_size=3×3, stride=1, padding=1

·ReLU：将卷积结果输入到ReLU函数，不改变特征图参数

·卷积层C5

·卷积：num\_kernels=256, kernel\_size=3×3, stride=1, padding=1

·ReLU：将卷积结果输入到ReLU函数，不改变特征图参数

·池化：kernel\_size=3×3, stride=2, padding=0,

·全连接层F6

·全连接：out\_features=4096

·ReLU：将卷积结果输入到ReLU函数，不改变特征图参数

·Dropout：抑制过拟合，以50%的概率不激活某些神经元

·全连接层F7

·全连接：out\_features=1024

·ReLU：将卷积结果输入到ReLU函数，不改变特征图参数

·Dropout：抑制过拟合，以50%的概率不激活某些神经元

·全连接层F8

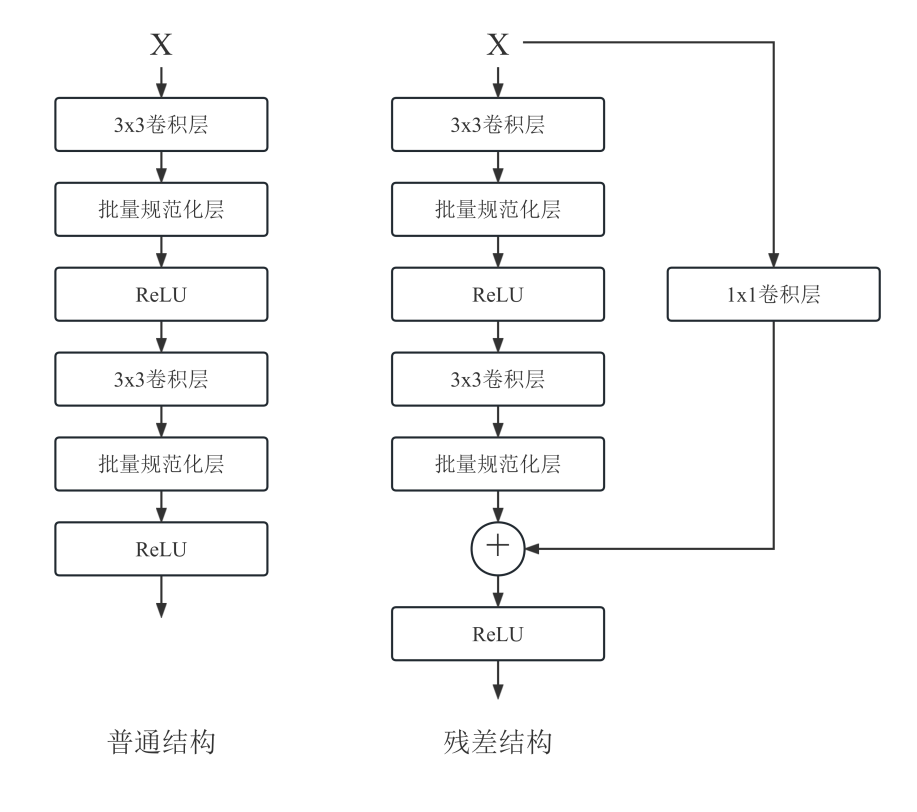
·全连接：out\_features=100

1. ResNet模型

在ResNet提出之前，卷积神经网络都是通过卷积层和池化层的叠加组成的，但是随着卷积和池化的层数增多，反而出现了两种问题：

一是退化问题，随着层数的增加，预测效果反而越来越差；

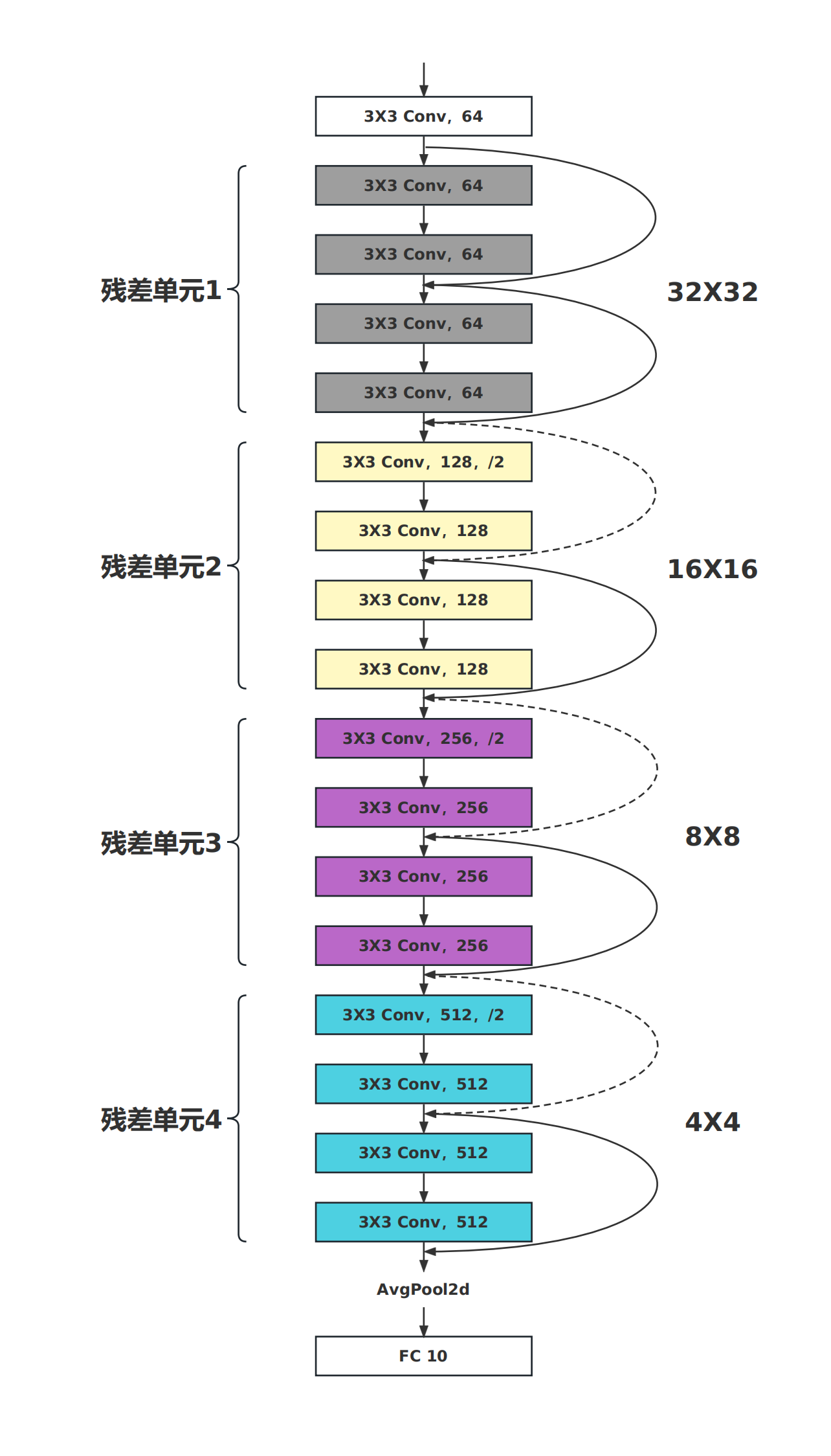
为了解决退化问题，ResNet论文提出了一种残差结构(Residual)，通过隔层相连，弱化每层之间的强联系，如下图所示：



二是梯度消失或梯度爆炸：单层的误差梯度小于(大于)1，反向传播时，网络越深，梯度越趋于0(趋于无穷)

为了解决梯度消失或梯度爆炸问题，ResNet论文提出通过数据的预处理以及在网络中使用BN(Batch Normalization)层来解决；

受限于个人硬件资源，本次实验选取ResNet-18模型进行学习，具体结构如下图所示。

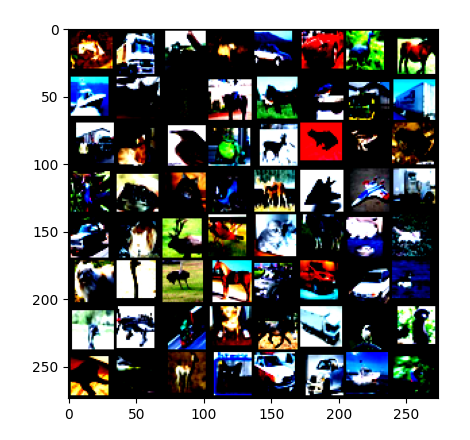
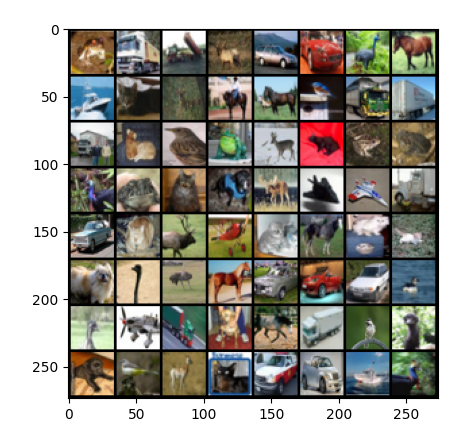


1. 实验过程
2. 数据预处理

PyTorch对CIFAR-10数据集提供了支持，可以通过官方代码自动加载；但是由于图片尺寸只有32x32像素，且训练集只有5万张图片，故需要对数据进行增强，包括随机缩放裁剪(RandomResizedCrop)和水平翻转(RandomResizedCrop)，并在转化为Tensor格式时进行进行归一化处理，详细参数如下：

*# 定义Transform*data\_transform = {  
 'train': transforms.Compose([transforms.RandomCrop(32, padding=4),  
 transforms.RandomHorizontalFlip(),  
 transforms.ToTensor(),  
 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])]),  
 'val': transforms.Compose([transforms.ToTensor(),  
 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])}

最终获得的数据集部分展示如下图：



处理前 处理后

Mini-ImageNet数据集主要是针对小样本学习领域，通过pandas对标签文件进行分析，获得了如下信息：

·train.csv包含38400张图片标签，共计64个类别

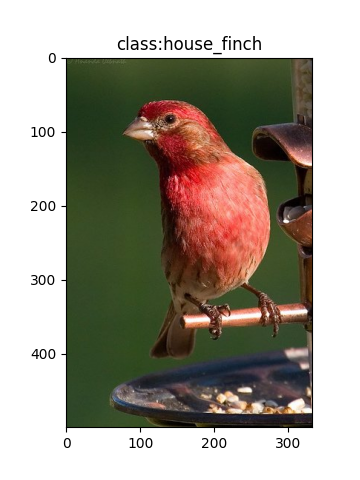
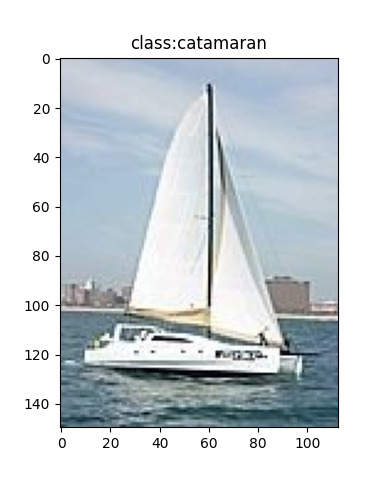
·val.csv包含9600张图片标签，共计16个类别

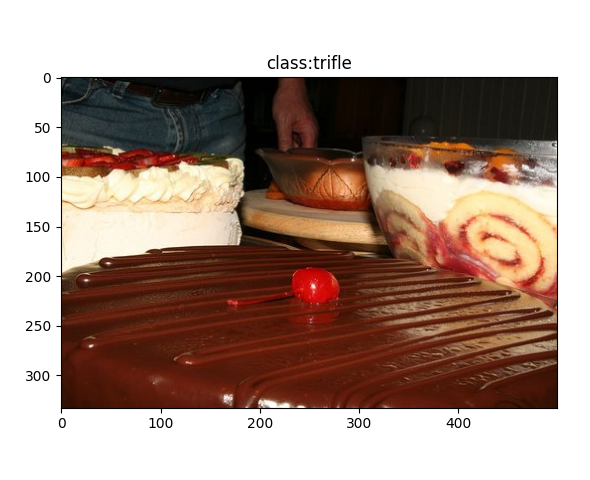
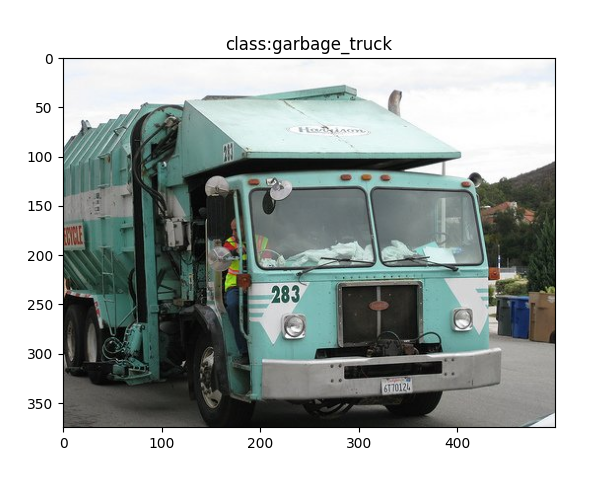
·test.csv包含12000张图片，共计20个类别

所以，直接使用Mini-ImageNet数据集训练AlexNet和ResNet是不可行的，我们需要重新构建一个新的train.csv、val.csv标签文件，最终获得的数据集结构如下：

·train.csv包含48000张图片标签，共计100个类别

·val.csv包含12000张图片标签，共计100个类别



由上述样图可知Mini-ImageNet数据集中图片尺寸并不统一，因此需要将图像裁剪、压缩为统一尺寸224x224x3，在转化为Tensor格式时进行进行归一化处理，详细参数如下：

*# 定义Transform*data\_transform = {  
 'train': transforms.Compose([transforms.RandomResizedCrop(224),  
 transforms.RandomHorizontalFlip(),  
 transforms.ToTensor(),  
 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])]),  
 'val': transforms.Compose([transforms.Resize(256),  
 transforms.CenterCrop(224),  
 transforms.ToTensor(),  
 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])}

1. 模型构建
   1. LeNet

class CNN\_1LeNet(nn.Module):  
 *# 重写构造函数* def \_\_init\_\_(self):  
 super(CNN\_1LeNet, self).\_\_init\_\_()  
 self.conv1 = nn.Sequential(  
 nn.Conv2d( *# 32x32 ——> 28x28* in\_channels=3, *# 输入通道数* out\_channels=16, *# 输出通道数* kernel\_size=5, *# 卷积核的大小* stride=1, *# 步长* padding=0 *# 填充* ),  
 nn.ReLU(),  
 nn.MaxPool2d( *# 28x28 ——> 14x14* kernel\_size=2, *# 卷积核的大小* stride=2, *# 步长* padding=0 *# 填充* )  
 )  
 self.conv2 = nn.Sequential(  
 nn.Conv2d( *# 14x14 ——> 10x10* in\_channels=16, *# 输入通道数* out\_channels=32, *# 输出通道数* kernel\_size=5, *# 卷积核的大小* stride=1, *# 步长* padding=0 *# 填充* ),  
 nn.ReLU(),  
 nn.MaxPool2d( *# 10x10 ——> 5x5* kernel\_size=2, *# 卷积核的大小* stride=2, *# 步长* padding=0 *# 填充* )  
 )  
 self.classifier = nn.Sequential(  
 nn.Flatten(), *# 降维* nn.Linear(32 \* 5 \* 5, 120), *# 全连接层* nn.ReLU(),  
  
 nn.Linear(120, 84), *# 全连接层* nn.ReLU(),  
  
 nn.Linear(84, 10) *# 全连接层* )  
  
 def forward(self, x):  
 x = self.conv1(x)  
 x = self.conv2(x)  
 x = self.classifier(x)  
 return x

* 1. AlexNet

class CNN\_2AlexNet(nn.Module):  
 *# 重写构造函数* def \_\_init\_\_(self):  
 super(CNN\_2AlexNet, self).\_\_init\_\_()  
 self.conv1 = nn.Sequential(  
 nn.Conv2d( *# 227x227 ——> 55x55* in\_channels=3, *# 输入通道数* out\_channels=96, *# 输出通道数* kernel\_size=11, *# 卷积核的大小* stride=4, *# 步长* padding=0 *# 填充* ),  
 nn.ReLU(inplace=True),  
 nn.LocalResponseNorm(size=5, alpha=0.0001, beta=0.75, k=2),  
 nn.MaxPool2d( *# 55x55 ——> 27x27* kernel\_size=3, *# 卷积核的大小* stride=2, *# 步长* padding=0 *# 填充* )  
 )  
 self.conv2 = nn.Sequential(  
 nn.Conv2d( *# 27x27 ——> 27x27* in\_channels=96, *# 输入通道数* out\_channels=256, *# 输出通道数* kernel\_size=5, *# 卷积核的大小* stride=1, *# 步长* padding=2 *# 填充* ),  
 nn.ReLU(inplace=True),  
 nn.LocalResponseNorm(size=5, alpha=0.0001, beta=0.75, k=2),  
 nn.MaxPool2d( *# 27x27 ——> 13x13* kernel\_size=3, *# 卷积核的大小* stride=2, *# 步长* padding=0 *# 填充* )  
 )  
 self.conv3 = nn.Sequential(  
 nn.Conv2d( *# 13x13 ——> 13x13* in\_channels=256, *# 输入通道数* out\_channels=384, *# 输出通道数* kernel\_size=3, *# 卷积核的大小* stride=1, *# 步长* padding=1 *# 填充* ),  
 nn.ReLU(inplace=True),  
 )  
 self.conv4 = nn.Sequential(  
 nn.Conv2d( *# 13x13 ——> 13x13* in\_channels=384, *# 输入通道数* out\_channels=384, *# 输出通道数* kernel\_size=3, *# 卷积核的大小* stride=1, *# 步长* padding=1 *# 填充* ),  
 nn.ReLU(inplace=True),  
 )  
 self.conv5 = nn.Sequential(  
 nn.Conv2d( *# 13x13 ——> 13x13* in\_channels=384, *# 输入通道数* out\_channels=256, *# 输出通道数* kernel\_size=3, *# 卷积核的大小* stride=1, *# 步长* padding=1 *# 填充* ),  
 nn.ReLU(inplace=True),  
 nn.MaxPool2d( *# 13x13 ——> 6x6* kernel\_size=3, *# 卷积核的大小* stride=2, *# 步长* padding=0 *# 填充* )  
 )  
 self.classifier = nn.Sequential(  
 nn.Flatten(), *# 降维* nn.Linear(256 \* 6 \* 6, 4096), *# 全连接层* nn.ReLU(inplace=True),  
 nn.Dropout(),  
  
 nn.Linear(4096, 4096), *# 全连接层* nn.ReLU(inplace=True),  
 nn.Dropout(),  
  
 nn.Linear(4096, 100) *# 全连接层* )  
  
 def forward(self, x):  
 x = self.conv1(x)  
 x = self.conv2(x)  
 x = self.conv3(x)  
 x = self.conv4(x)  
 x = self.conv5(x)  
 x = self.classifier(x)  
 return x

* 1. ResNet

class CNN\_3ResNet(nn.Module):  
 *# 重写构造函数* def \_\_init\_\_(self, block):  
 super(CNN\_3ResNet, self).\_\_init\_\_()  
 self.conv1 = nn.Sequential(  
 nn.Conv2d( *# 224x224 ——> 112x112* in\_channels=3, *# 输入通道数* out\_channels=64, *# 输出通道数* kernel\_size=7, *# 卷积核的大小* stride=2, *# 步长* padding=3, *# 填充* bias=False *# 取消偏执* ),  
 nn.BatchNorm2d(64), *# 批归一化* nn.ReLU(inplace=True),  
 nn.MaxPool2d( *# 112x112 ——> 56x56* kernel\_size=3, *# 卷积核的大小* stride=2, *# 步长* padding=1 *# 填充* )  
 )  
 self.in\_channels = 64  
 self.layer1 = self.make\_layer(block, 64, 2, 1) *# 56x56 ——> 28x28* self.layer2 = self.make\_layer(block, 128, 2, 2) *# 28x28 ——> 14x14* self.layer3 = self.make\_layer(block, 256, 2, 2) *# 14x14 ——> 7x7* self.layer4 = self.make\_layer(block, 512, 2, 2) *# 7x7 ——> 1x1* self.classifier = nn.Sequential(  
 *#* nn.AdaptiveAvgPool2d((1, 1)), *# 池化  
 #* nn.Flatten(), *# 降维  
 #* nn.Linear(512, 100) *# 全连接层* )  
  
 def make\_layer(self, block, out\_channels, num\_blocks, stride):  
 strides = [stride] + [1] \* (num\_blocks - 1)  
 layers = []  
 for s in strides:  
 layers.append(block(self.in\_channels, out\_channels, s))  
 self.in\_channels = out\_channels  
 return nn.Sequential(\*layers)  
  
 def forward(self, x):  
 x = self.conv1(x)  
 x = self.layer1(x)  
 x = self.layer2(x)  
 x = self.layer3(x)  
 x = self.layer4(x)  
 x = self.classifier(x)  
 return x  
  
  
class ResBlock(nn.Module):  
 *# 重写构造函数* def \_\_init\_\_(self, in\_channels, out\_channels, stride):  
 super(ResBlock, self).\_\_init\_\_()  
 self.conv1 = nn.Sequential(  
 nn.Conv2d(  
 in\_channels=in\_channels, *# 输入通道数* out\_channels=out\_channels, *# 输出通道数* kernel\_size=3, *# 卷积核的大小* stride=stride, *# 步长* padding=1, *# 填充* bias=False *# 偏执* ),  
 nn.BatchNorm2d(out\_channels),  
 nn.ReLU(inplace=True)  
 )  
 self.conv2 = nn.Sequential(  
 nn.Conv2d(  
 in\_channels=out\_channels, *# 输入通道数* out\_channels=out\_channels, *# 输出通道数* kernel\_size=3, *# 卷积核的大小* stride=1, *# 步长* padding=1, *# 填充* bias=False *# 偏执* ),  
 nn.BatchNorm2d(out\_channels)  
 )  
 if stride != 1 or in\_channels != out\_channels:  
 self.shortcut = nn.Sequential(  
 nn.Conv2d(  
 in\_channels=in\_channels, *# 输入通道数* out\_channels=out\_channels, *# 输出通道数* kernel\_size=1, *# 卷积核的大小* stride=stride, *# 步长* padding=0, *# 填充* bias=False *# 偏执* ),  
 nn.BatchNorm2d(out\_channels)  
 )  
 else:  
 self.shortcut = None  
  
 def forward(self, x):  
 out = self.conv1(x)  
 out = self.conv2(out)  
 if self.shortcut is not None:  
 out = out + self.shortcut(x)  
 out = F.relu(out)  
 return out

1. 迭代训练
   1. 参数设置

数据、模型、损失函数与优化器被称为神经网络的四要素，目前数据和模型已经构建完毕，要想获得良好的结果，合适的损失函数和优化器也是必不可少的。

损失函数在深度学习领域是用来计算搭建模型预测的输出值和真实值之间的误差。具体流程为：在一个批次(batch)前向传播完成后，得到模型预测值，然后损失函数计算出预测值和真实值之间的差值。

常见的损失函数有：BCELoss(二分类交叉熵损失)、CrossEntropyLoss(交叉熵损失)、MSELoss(均方误差损失)、L1Loss(平均绝对值误差损失)等等，由于本次实验的任务是多分类场景，因此选择CrossEntropyLoss作为损失函数。

*# 损失函数*loss\_fun = nn.CrossEntropyLoss().to(device)

优化器在深度学习领域是用来更新模型中的参数。通过损失函数计算得到误差后，优化器一般采用梯度下降算法，通过反向传播去更新权值和偏置等参数，以降低差值，不断向真实值接近，最终得到效果良好的模型。

常见的优化器有：SGD(随机梯度下降)、Adam(自适应矩估计)、Adagrad(自适应梯度)、RMSprop(均方根传递)等等，由于SGD在训练前期迭代效果卓越，Adam在训练后期可以很快收敛，所以本次实验在训练前期采用SGD作为优化器，后期采用Adam作为优化器。

*# 优化器*optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight\_decay=5e-4)

*# 优化器*optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

此外，优化器中的学习率也是一个重要的参数，选择一个合适的学习率即可以缩短训练时间，也可以提供训练效果，但是一个合适的学习率需要不断实验调整，在复杂任务中一次训练就需要十几甚至几十个小时，因此反复调参是不可取的，故PyTorch提供了多种学习率衰减策略，有LambdaLR、StepLR、ExponentialLR、CosineAnnealingLR等等，本次实验采用衰减曲线较为平滑的CosineAnnealingLR。

*# 学习率衰减*scheduler = lr\_scheduler.CosineAnnealingLR(optimizer, T\_max=100)

* 1. 断点续训

深度学习训练非常耗费时间，如果中间中止训练，则会非常麻烦。所以为了节省时间，通常会让模型从断点中继续训练。为此，需要模型在训练的过程中保存一些关键信息，比如模型的参数、优化器的配置，epoch等，并在下次训练时进行调用。故断点续训分为两部分，模型保存和断点恢复：

·模型保存

def save\_checkpoint(model, model\_name, loss\_fun, optimizer, epoch, doc):  
 *# 模型保存路径* model\_path = './data/model/' + model\_name + f'{epoch}.pth'  
 *# 字典* state = {  
 'epoch': epoch,  
 'model': model,  
 'loss': loss\_fun,  
 'optim': optimizer.state\_dict()}  
 *# 保存模型* torch.save(state, model\_path)  
 *#* print('Checkpoint saved to {}'.format(model\_path))  
 print('Checkpoint saved to {}'.format(model\_path), file=doc)

·断点恢复

def load\_checkpoint(model\_path, model, optimizer, doc):  
 *# 判断断点是否存在* if os.path.isfile(model\_path):  
 checkpoint = torch.load(model\_path)  
 epoch = checkpoint['epoch'] + 1  
 model.load\_state\_dict(checkpoint['model'])  
 optimizer.load\_state\_dict(checkpoint['optim'])  
 loss\_fun = checkpoint['loss']  
 print('Success load checkpoint {}'.format(model\_path))  
 print('Success load checkpoint {}'.format(model\_path), file=doc)  
 return model, loss\_fun, optimizer, epoch  
 else:  
 print('There is no checkpoint {}'.format(model\_path))  
 print('There is no checkpoint {}'.format(model\_path), file=doc)  
 return None

* 1. 日志记录

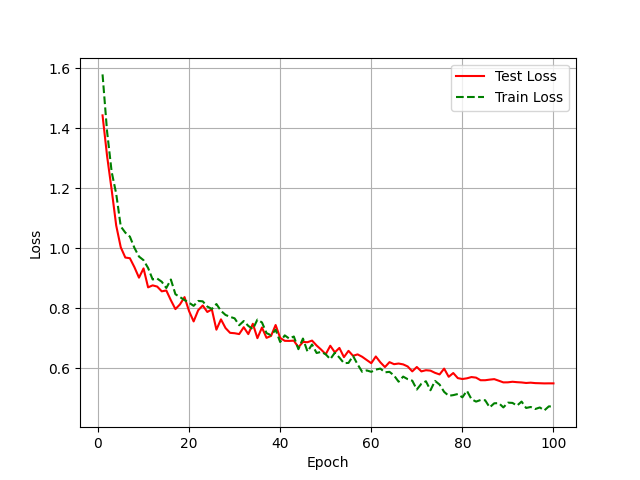
良好的日志记录是实验成功的关键，对于模型学习训练过程中的参数变化、指标波动，仅仅打印在控制台是不够的，一次完整的训练可能耗费数个小时，在训练完成后要留下足够多的数据用以分析和复盘，所以在整个项目中，要对关键数据进行存档，形成良好的日志记录。实验中部分日志记录如下：

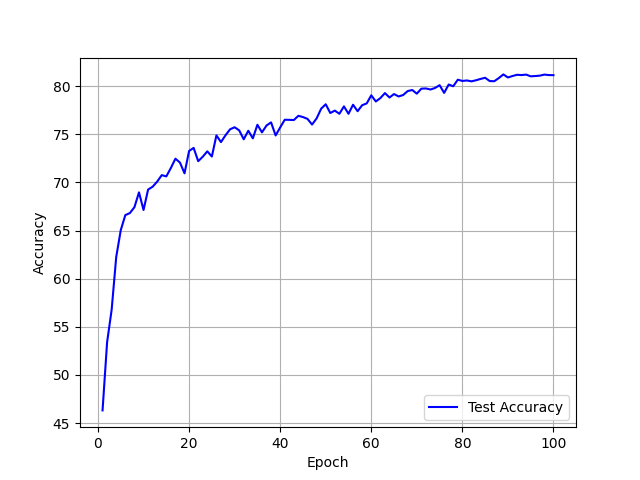
四、实验结果

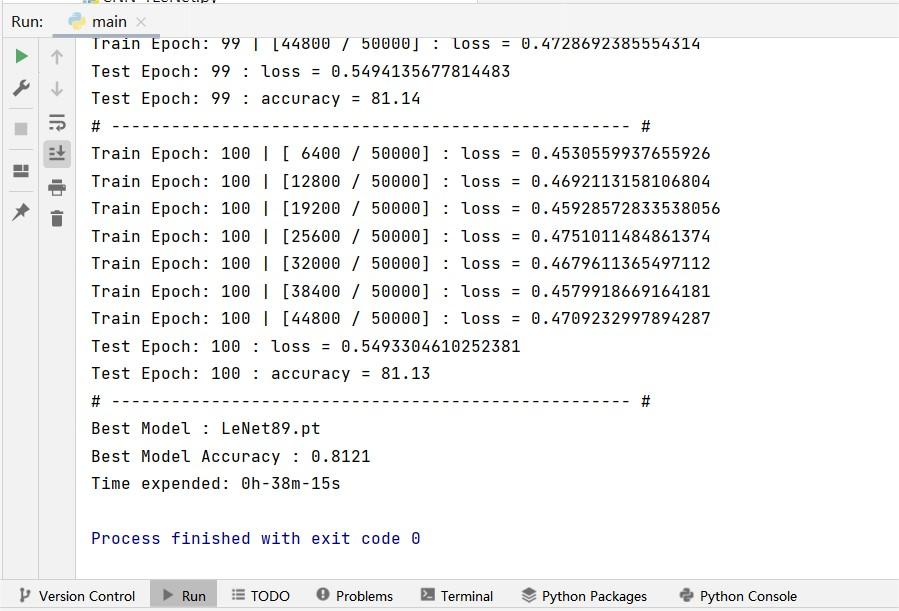
在模型训练中，均采用100个epoch、学习率余弦退火衰减的参数，最后选择其中测试成绩最高的作为最终模型。

* 1. LeNet

LeNet在100个epoch中，测试集最大Accuracy出现在第epoch，损失函数、正确率及部分日志如下图所示：

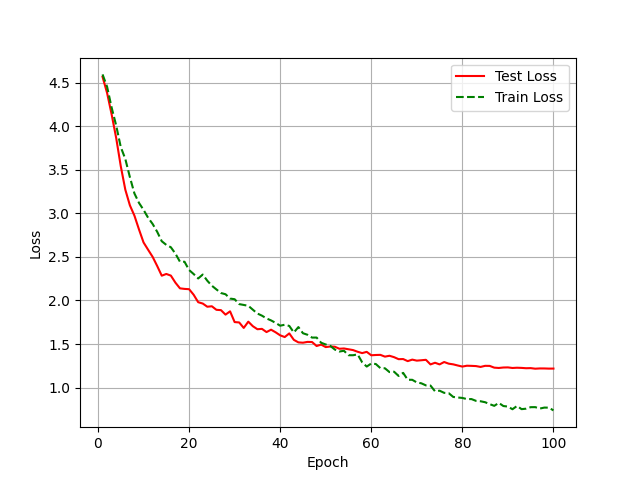


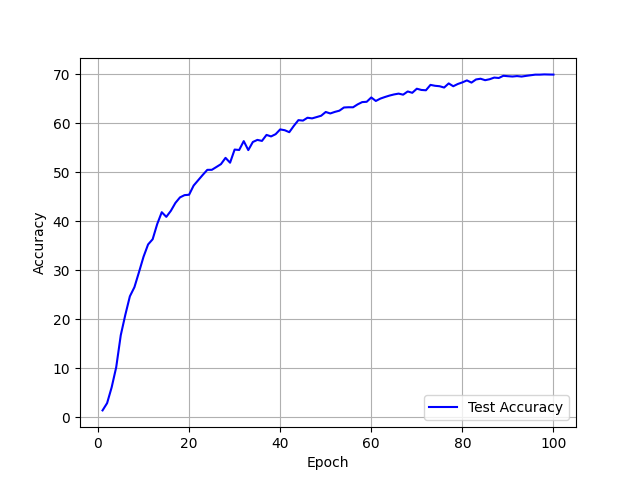


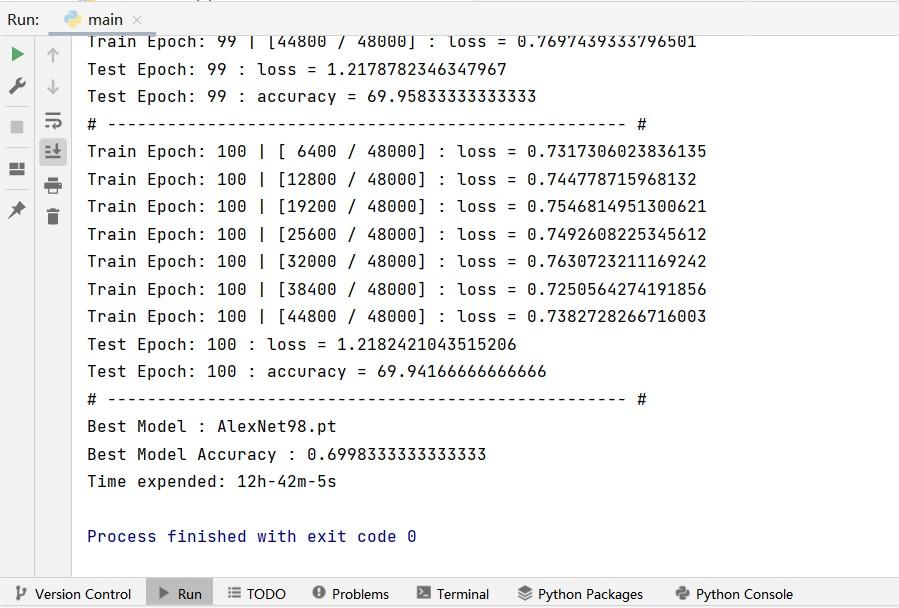


* 1. AlexNet

AlexNet在100个epoch中，测试集最大Accuracy出现在第epoch，损失函数、正确率及部分日志如下图所示：

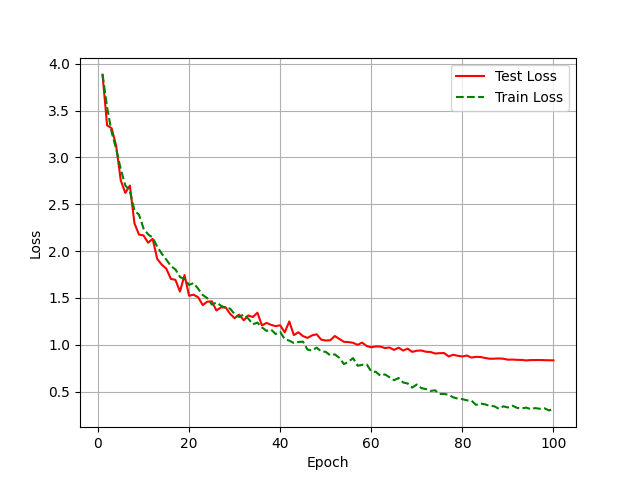


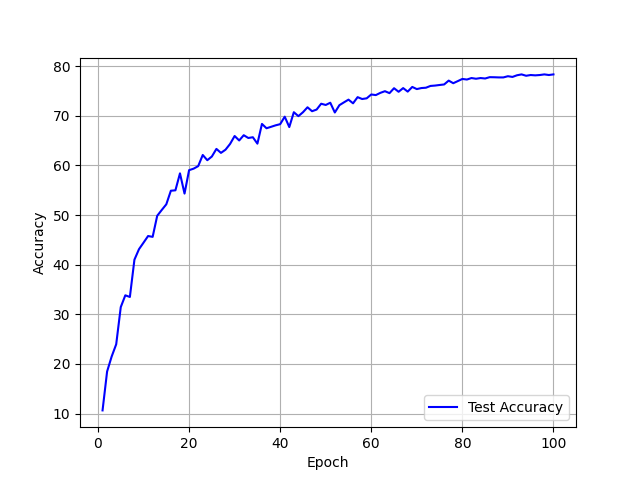


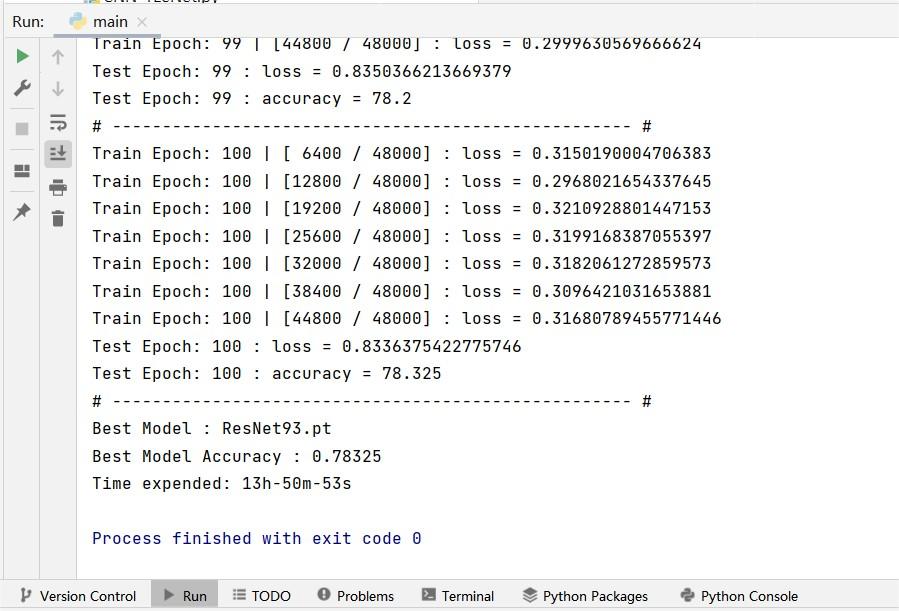


* 1. ResNet

ResNet在100个epoch中，测试集最大Accuracy出现在第epoch，损失函数、正确率及部分日志如下图所示：







五、总结展望

通过本次实验，我对卷积神经网络有了更加深入的了解，通过学习一些经典的卷积神经网络模型，如LeNet、AlexNet、ResNet，对卷积、池化、局部感知、视野等概念有了自己的认识，不再局限于照搬照本宣科，能够根据任务需要搭建合适的网络模型。

同时，我也认识到了自身的不足，由于本科期间没有学习过机器学习相关内容，NLP、CV等方向只是有所了解，对Encoder-Decoder、Transformer、Self-Attention等知识不够熟悉，无法完成一些进阶的项目，需要在今后的工作和学习中加倍努力，向优秀的同学看齐。

最后，感谢几位老师和学长的谆谆教导，让我获益良多，预祝大家新年快乐，身体健康，万事如意！