**《人工智能》实验报告**

实验名称 基于卷积神经网络的图像分类研究

学生姓名 黄鸿展

学　　　　号 202230441138

学生班级 计算机科学与技术2班

实验日期 2024年12月20日

实验成绩

一、实验内容

本实验旨在通过使用两种不同的卷积神经网络（CNN）模型对 CIFAR-10 数据集进行分类训练和测试，以比较不同模型的性能和优缺点。我选取了 AlexNet 和 ResNet18 作为实验的模型，并记录了它们的训练和测试过程，包括损失、准确率等关键指标。

二、实验过程

1．CIFAR10数据集预处理

CIFAR-10数据集包含60,000张32x32像素的彩色图像，分为10个类别，每个类别有6,000张图像。这些类别包括飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船和卡车。在这60,000张图像中，50,000张用作训练集，10,000张用作测试集。训练集和测试集都包含来自10个类别的图像，测试集中每个类别随机抽取1,000张图像，而训练集中每个类别有5,000张图像。CIFAR-10中的图像都是32x32像素的RGB彩色图像，这意味着每个图像由32行和32列的像素组成，每个像素包含红色、绿色和蓝色三个颜色通道。

对于 CIFAR-10 数据集，进行了以下预处理操作：

随机水平翻转-随机旋转-随机仿射变换-色彩抖动-转换为张量-归一化。

通过这一系列操作，有助于数据增强，提高模型的泛化能力。对于AlexNet数据集处理缺少了部分操作，主要目的是适配AlexNet的输入需求。

2. 模型构建

我选择了以下模型：

①AlexNet：是一个深度卷积神经网络，一个经典的 CNN 模型，具有 8 层，包括 5 个卷积层和 3 个全连接层。它的主要结构如下：

输入层：接受3通道（RGB）的224x224像素图像。

卷积层：AlexNet包含五个卷积层，其中前三个卷积层后面跟着最大池化层（Max Pooling）。

ReLU激活函数：每个卷积层后面都跟着ReLU激活函数。

归一化层： AlexNet在PyTorch实现中，使用批量归一化。

全连接层：在卷积层后，有三个全连接层，用于处理特征图并输出预测。

输出层：一个softmax层，用于输出每个类别的概率。

②ResNet18：ResNet通过引入“残差学习”来解决深度网络训练中的退化问题，即随着网络层数的增加，训练误差反而增加。ResNet-18是ResNet家族中相对较浅的一个模型，包含18层深度。它的主要结构如下：

输入层：接受3通道（RGB）的32x32像素图像。

卷积层：一个初始的卷积层，将输入图像转换为64个特征图。

残差块：ResNet-18由多个残差块（BasicBlock）组成，每个块包含两个3x3的卷积层，以及批量归一化（BatchNorm）和ReLU激活函数。每个残差块的输出通过一个跳跃连接（shortcut connection）与输入相加，以实现残差学习。

池化层：在每个残差块之后，使用平均池化层减少特征图的空间维度。

全连接层：最后，使用一个全连接层将特征图转换为最终的类别输出。

输出层：一个softmax层，用于输出每个类别的概率。

3. 具体实现

3.1 AlexNet模型

AlexNet类定义了AlexNet网络的结构，包括特征提取层和分类器层。

特征提取层（self.features）：包含五个卷积层，每个卷积层后面跟着ReLU激活函数和最大池化层（除了最后一个卷积层）。

第一个卷积层的步长为2，导致特征图尺寸减半。

最大池化层的核大小为2，进一步减少特征图的空间尺寸。

分类器层（self.classifier）：包含三个全连接层，每个全连接层后面跟着ReLU激活函数，除了最后一个全连接层。使用Dropout层来减少过拟合。

前向传播（forward方法）：

输入数据x首先通过特征提取层。然后，将特征图展平并通过分类器层。最后，应用log\_softmax函数来获取最终的输出。

|  |
| --- |
| def forward(self, x):  x = self.features(x)  x = x.view(x.size(0), 256 \* 2 \* 2) *# 展平特征图*  x = self.classifier(x)  return F.log\_softmax(x, dim=-1) |

3.2 ResNet模型：

代码中定义了两个类：BasicBlock和ResNet。

BasicBlock类：这是ResNet-18中的基本构建块，每个块包含两个卷积层，每个卷积层后面跟着批量归一化（BatchNorm）和ReLU激活函数。如果需要，还有一个shortcut连接，用于连接输入和输出，以实现残差学习。

ResNet类：初始化时，会创建一个卷积层、四个残差层（layer1到layer4），以及一个全连接层。

\_make\_layer方法，接受残差块类型、输出通道数、块的数量和步长作为参数。为每个残差块指定了一个步长，并且除了第一个残差块之外，其余残差块的步长都被设置为1。然后，使用nn.Sequential将这些残差块序列化，形成一个完整的层。

|  |
| --- |
| def \_make\_layer(self, block, planes, num\_blocks, stride):  strides = [stride] + [1]\*(num\_blocks-1)  layers = []  for stride in strides:  layers.append(block(self.in\_planes, planes, stride))  self.in\_planes = planes \* block.expansion  return nn.Sequential(\*layers) |

前向传播（forward方法）定义了数据如何通过模型流动，包括卷积、残差块、池化和全连接层。

首先，数据通过初始的卷积层、批量归一化层和ReLU激活函数。

然后，数据依次通过四个由\_make\_layer创建的残差层（layer1到layer4）。

在每个残差层之后，特征图的尺寸会减半（因为步长为2），通道数会增加。

最后，经过平均池化层（F.avg\_pool2d）将特征图的空间尺寸减少到1x1，然后展平（view）并传递给全连接层，最终通过softmax函数输出类别概率。

|  |
| --- |
| def forward(self, x):  out = F.relu(self.bn1(self.conv1(x)))  out = self.layer1(out)  out = self.layer2(out)  out = self.layer3(out)  out = self.layer4(out)  out = F.avg\_pool2d(out, 4)  out = out.view(out.size(0), -1)  out = self.linear(out)  return F.log\_softmax(out, dim=-1) |

3.3 模型训练流程如下：

①导入必要的库：PyTorch、torchvision、transforms、matplotlib、numpy等库

②环境设置：在kaggle中将设备设置为GPU100，然后设备改为用GPU训练。

③定义数据预处理：定义GetTransforms类，其中包含trainparams和testparams方法，用于定义训练和测试数据的预处理步骤。

④下载和加载数据集：

使用GetCIFAR10\_TrainData类下载CIFAR-10数据集，并应用预处理步骤。

通过torch.utils.data.DataLoader创建数据加载器trainloader和testloader，用于在训练和测试时批量加载数据。

⑤定义模型结构

⑥查看模型：使用torchsummary库的summary函数打印模型的结构摘要，以查看每层的输出形状和参数数量。

⑦定义训练和测试函数：

定义model\_training函数，用于执行模型的训练过程。

定义model\_testing函数，用于执行模型的测试过程，并计算测试准确率。

⑧设置优化器和学习率调度器：

设置交叉熵损失函数nn.CrossEntropyLoss。

设置SGD优化器optim.SGD，用于更新模型权重。

设置学习率调度器，用于根据验证损失调整学习率。初始学习率设置为0.05.

⑨训练模型：

初始化用于记录训练准确率和损失的列表。

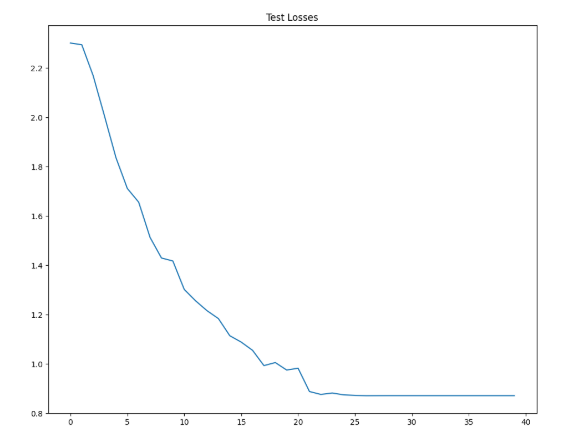
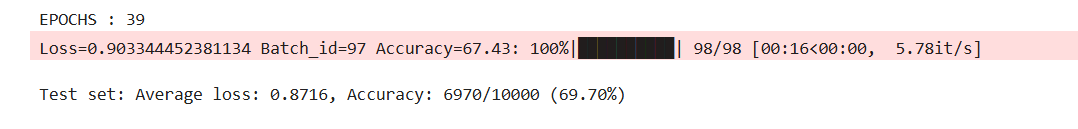
进行40个epoch的训练，每个epoch都会调用model\_training函数，并在每个epoch结束后使用model\_testing函数评估模型性能。

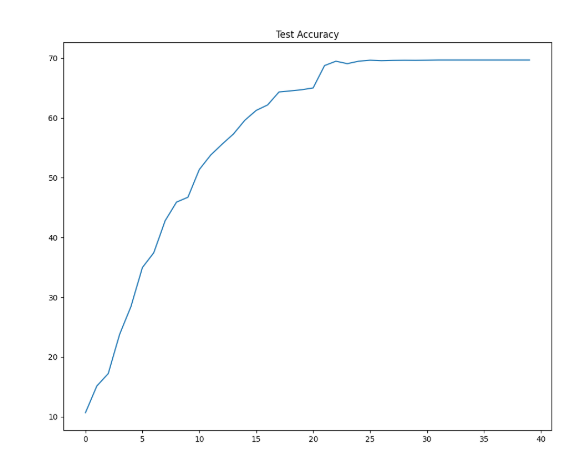
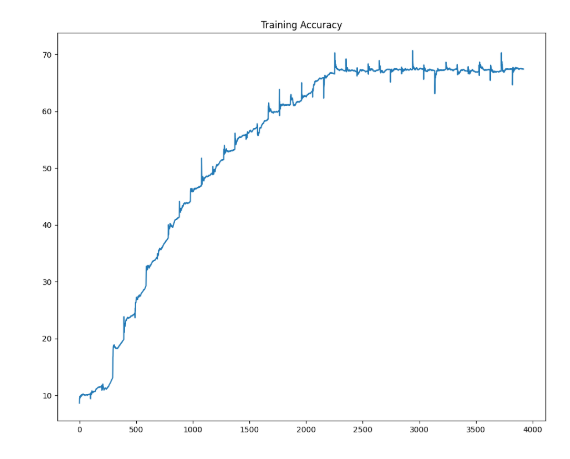
⑩结果可视化：使用matplotlib库绘制训练和测试过程中的损失和准确率曲线。

输出训练和测试结果：打印出模型在测试集上的平均损失和准确率。

三、实验结果

AlexNet:

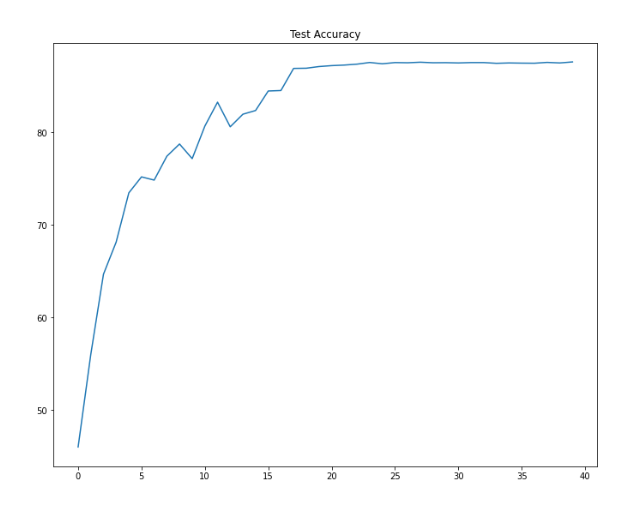
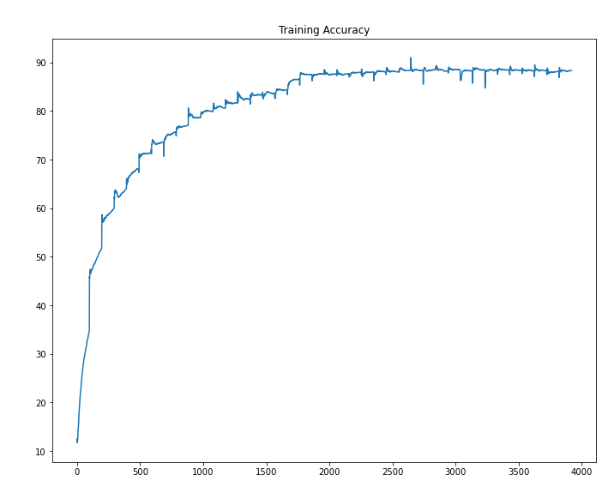


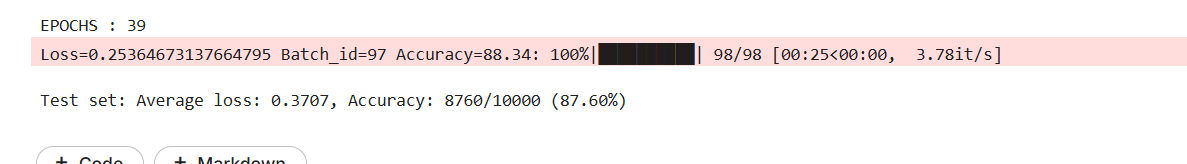


训练准确率在每个 epoch 中波动较大，尤其是在早期 epoch 中，准确率较低（接近 0）。随着 epoch 的增加，训练准确率逐渐上升，但仍然存在较大的波动,这表明 AlexNet 在 CIFAR-10 上的训练可能存在过拟合或训练不稳定的问题。

测试准确率在 40 个 epoch 后达到了 69.70%，这是一个相对较低的准确率。测试损失从初始的 2.2 下降到 0.87，表明模型在一定程度上学习到了有效的特征。损失下降的速度较慢，尤其是在后期 epoch 中，损失下降的幅度较小，表明模型可能已经接近收敛。

ResNet18



从训练准确率图中可以看出，随着训练的进行，模型的准确率迅速提升，并在大约2000个迭代后趋于稳定，最终接近90%。这表明模型在训练集上学习得很好。

在训练过程中，损失值从较高的初始值逐渐下降，这表明模型的预测误差在减少。在训练的后期，损失值趋于平稳，说明模型可能已经接近或达到其在训练集上的性能极限。

训练准确率接近90%，而测试准确率略低，这可能表明模型存在一定程度的过拟合。过拟合意味着模型在训练集上表现很好，但在未见过的测试集上表现稍差。

结果对比

AlexNet 是深度学习的早期经典模型，结构简单，易于理解和实现。对于较小的数据集（如 CIFAR-10），AlexNet 的参数量相对较少，训练速度较快。但是AlexNet 在 CIFAR-10 上的表现较差，可能是由于模型容量不足或过拟合。而训练准确率和测试准确率波动较大，表明模型在训练过程中可能存在梯度消失或梯度爆炸问题。测试准确率较低，表明模型在测试集上的表现不佳。

ResNet 通过引入残差连接（residual connections），解决了深度网络中的梯度消失问题，使得模型可以训练更深的网络。实验证明，ResNet 在 CIFAR-10 上的测试准确率通常可以达到 90% 以上，表明其具有更好的泛化能力。ResNet 的训练过程更加稳定，训练准确率和测试准确率曲线更加平滑。但是ResNet 的参数量通常比 AlexNet 大，训练时间更长。：ResNet 的结构比 AlexNet 复杂，理解和实现难度较大。

四、实验心得

通过本次实验，我使用了 CIFAR-10 数据集，并分别训练了两种不同结构的卷积神经网络模型：AlexNet 和 ResNet。通过对比这两种模型的训练和测试结果，我对它们的优缺点有了更深入的理解。我深刻认识到模型结构和优化策略对深度学习模型性能的重要性。AlexNet 作为早期模型，虽然结构简单，但在 CIFAR-10 上的表现较差；而 ResNet 通过引入残差连接，显著提升了模型的泛化能力和训练稳定性。在实际应用中，选择合适的模型结构和优化策略是取得良好性能的关键。

五、附录（关键代码）

AlexNet实现：

|  |
| --- |
| import numpy as np import pandas as pd import os  for dirname, \_, filenames in os.walk('/kaggle/input'):     for filename in filenames:         print(os.path.join(dirname, filename))  !pip install torchsummary  import torch import torchvision import torchvision.transforms as transforms from torchsummary import summary import torch.nn as nn import torch.nn.functional as F import torch.optim as optim from torch.optim.lr\_scheduler import StepLR, ReduceLROnPlateau import matplotlib.pyplot as plt %matplotlib inline  device = "cuda" if torch.cuda.is\_available() else "cpu" print(device)  class GetTransforms:     def \_\_init\_\_(self):         pass      def trainparams(self):         train\_transformations = [             transforms.RandomHorizontalFlip(),             transforms.RandomRotation((-7,7)),             transforms.RandomAffine(0, shear=10, scale=(0.8,1.2)),             transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),             transforms.ToTensor(),             transforms.Normalize((0.491, 0.482, 0.446), (0.247, 0.243, 0.261))         ]         return train\_transformations      def testparams(self):         test\_transforms = [             transforms.ToTensor(),             transforms.Normalize((0.491, 0.482, 0.446), (0.247, 0.243, 0.261))         ]         return test\_transforms  transformations = GetTransforms() train\_transforms = transforms.Compose(transformations.trainparams()) test\_transforms = transforms.Compose(transformations.testparams())  class GetCIFAR10\_TrainData:     def \_\_init\_\_(self, dir\_name:str):         self.dirname = dir\_name      def download\_train\_data(self):         return torchvision.datasets.CIFAR10('./data', train=True, download=True, transform=train\_transforms)      def download\_test\_data(self):         return torchvision.datasets.CIFAR10('./data', train=False, download=True, transform=test\_transforms)  data = GetCIFAR10\_TrainData(os.chdir("..")) trainset = data.download\_train\_data() testset = data.download\_test\_data() trainloader = torch.utils.data.DataLoader(trainset, batch\_size=512, shuffle=True, num\_workers=4) testloader = torch.utils.data.DataLoader(testset, batch\_size=512, shuffle=False, num\_workers=4)  class AlexNet(nn.Module):     def \_\_init\_\_(self, num\_classes=10):         super(AlexNet, self).\_\_init\_\_()         self.features = nn.Sequential(             nn.Conv2d(3, 64, kernel\_size=3, stride=2, padding=1),             nn.ReLU(inplace=True),             nn.MaxPool2d(kernel\_size=2),             nn.Conv2d(64, 192, kernel\_size=3, padding=1),             nn.ReLU(inplace=True),             nn.MaxPool2d(kernel\_size=2),             nn.Conv2d(192, 384, kernel\_size=3, padding=1),             nn.ReLU(inplace=True),             nn.Conv2d(384, 256, kernel\_size=3, padding=1),             nn.ReLU(inplace=True),             nn.Conv2d(256, 256, kernel\_size=3, padding=1),             nn.ReLU(inplace=True),             nn.MaxPool2d(kernel\_size=2),         )         self.classifier = nn.Sequential(             nn.Dropout(),             nn.Linear(256 \* 2 \* 2, 4096),             nn.ReLU(inplace=True),             nn.Dropout(),             nn.Linear(4096, 4096),             nn.ReLU(inplace=True),             nn.Linear(4096, num\_classes),         )      def forward(self, x):         x = self.features(x)         x = x.view(x.size(0), 256 \* 2 \* 2)         x = self.classifier(x)         return F.log\_softmax(x, dim=-1)  model = AlexNet().to(device) summary(model, input\_size=(3,32,32))  from tqdm import tqdm  def model\_training(model, device, train\_dataloader, optimizer, train\_acc, train\_losses):     model.train()     pbar = tqdm(train\_dataloader)     correct = 0     processed = 0     running\_loss = 0.0      for batch\_idx, (data, target) in enumerate(pbar):         data, target = data.to(device), target.to(device)         optimizer.zero\_grad()         y\_pred = model(data)         loss = F.nll\_loss(y\_pred, target)         train\_losses.append(loss.item())  *# Convert loss to a scalar*         loss.backward()         optimizer.step()          pred = y\_pred.argmax(dim=1, keepdim=True)         correct += pred.eq(target.view\_as(pred)).sum().item()         processed += len(data)         running\_loss += loss.item()         pbar.set\_description(desc=f'Loss={loss.item()} Batch\_id={batch\_idx} Accuracy={100\*correct/processed:0.2f}')         train\_acc.append(100\*correct/processed)  def model\_testing(model, device, test\_dataloader, test\_acc, test\_losses, misclassified = []):     model.eval()     test\_loss = 0     correct = 0     classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')      with torch.no\_grad():         for index, (data, target) in enumerate(test\_dataloader):             data, target = data.to(device), target.to(device)             output = model(data)             pred = output.argmax(dim=1, keepdim=True)              for d,i,j in zip(data, pred, target):                 if i != j:                     misclassified.append([d.cpu(),i[0].cpu(),j.cpu()])              test\_loss += F.nll\_loss(output, target, reduction='sum').item()             correct += pred.eq(target.view\_as(pred)).sum().item()     test\_loss /= len(test\_dataloader.dataset)     test\_losses.append(test\_loss)      print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(         test\_loss, correct, len(test\_dataloader.dataset),         100. \* correct / len(test\_dataloader.dataset)))     test\_acc.append(100. \* correct / len(test\_dataloader.dataset))     return misclassified  criterion = nn.CrossEntropyLoss() optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.05, patience=2, threshold=0.0001, threshold\_mode='rel', cooldown=0, min\_lr=0, eps=1e-08, verbose=True)  train\_acc = [] train\_losses = [] test\_acc = [] test\_losses = []  EPOCHS = 40  for i in range(EPOCHS):     print(f'EPOCHS : {i}')     model\_training(model, device, trainloader, optimizer, train\_acc, train\_losses)     scheduler.step(train\_losses[-1])     misclassified = model\_testing(model, device, testloader, test\_acc, test\_losses)  *# Convert tensors to NumPy arrays for plotting* train\_losses = np.array(train\_losses) train\_acc = np.array(train\_acc) test\_losses = np.array(test\_losses) test\_acc = np.array(test\_acc)  fig, axs = plt.subplots(2,2, figsize=(25,20)) axs[0,0].set\_title('Train Losses') axs[0,1].set\_title('Training Accuracy') axs[1,0].set\_title('Test Losses') axs[1,1].set\_title('Test Accuracy')  axs[0,0].plot(train\_losses) axs[0,1].plot(train\_acc) axs[1,0].plot(test\_losses) axs[1,1].plot(test\_acc) |

ResNet实现：

|  |
| --- |
| import numpy as np  *# 线性代数* import pandas as pd  *# 数据处理，CSV文件输入/输出（例如pd.read\_csv）*  import os for dirname, \_, filenames in os.walk('/kaggle/input'):     for filename in filenames:         print(os.path.join(dirname, filename))  *# 你可以在当前目录（/kaggle/working/）中写入最多5GB的数据，这些数据在创建版本时会被保存为输出* *# 你还可以写入临时文件到/kaggle/temp/，但这些文件在当前会话之外不会被保存* !pip install torchsummary *# 导入库* import torch import torchvision import torchvision.transforms as transforms from torchsummary import summary import torch.nn as nn import torch.nn.functional as F import torch.optim as optim from torch.optim.lr\_scheduler import StepLR, ReduceLROnPlateau  import matplotlib.pyplot as plt %matplotlib inline  device = "cuda" if torch.cuda.is\_available() else "cpu" print(device) from torchvision import transforms import numpy as np import torch  *# 返回一个转换列表* class GetTransforms():     '''根据请求返回训练/测试数据的转换列表        Transforms('train') = 应用于训练数据的转换列表        Transforms('test') = 应用于测试数据的转换列表'''      def \_\_init\_\_(self):         pass      def trainparams(self):         train\_transformations = [  *# 调整图像大小，使其适合我们的模型。*             transforms.RandomHorizontalFlip(),  *# 相对于水平轴翻转图像*             transforms.RandomRotation((-7,7)),     *# 旋转图像到指定角度*             transforms.RandomAffine(0, shear=10, scale=(0.8,1.2)),  *# 执行缩放、改变倾斜角度等操作*             transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),  *# 设置颜色参数*             transforms.ToTensor(),  *# 将图像转换为张量，以便与torch一起工作*             transforms.Normalize((0.491, 0.482, 0.446), (0.247, 0.243, 0.261))  *# 归一化所有图像*             ]         return train\_transformations      def testparams(self):         test\_transforms = [             transforms.ToTensor(),             transforms.Normalize((0.491, 0.482, 0.446), (0.247, 0.243, 0.261))         ]         return test\_transforms from torchvision import datasets from torchvision import transforms   transformations = GetTransforms() train\_transforms = transforms.Compose(transformations.trainparams()) test\_transforms = transforms.Compose(transformations.testparams())   class GetCIFAR10\_TrainData():     def \_\_init\_\_(self, dir\_name:str):         self.dirname = dir\_name      def download\_train\_data(self):         return datasets.CIFAR10('./data', train=True, download=True, transform=train\_transforms)      def download\_test\_data(self):         return datasets.CIFAR10('./data', train=False, download=True, transform=test\_transforms)  data = GetCIFAR10\_TrainData(os.chdir("..")) trainset = data.download\_train\_data() testset = data.download\_test\_data() trainloader = torch.utils.data.DataLoader(trainset, batch\_size=512,                                           shuffle=True, num\_workers=4) testloader = torch.utils.data.DataLoader(testset, batch\_size=512,                                          shuffle=False, num\_workers=4) class BasicBlock(nn.Module):     expansion = 1      def \_\_init\_\_(self, in\_planes, planes, stride=1):         super(BasicBlock, self).\_\_init\_\_()          DROPOUT = 0.1          self.conv1 = nn.Conv2d(             in\_planes, planes, kernel\_size=3, stride=stride, padding=1, bias=False)         self.bn1 = nn.BatchNorm2d(planes)         self.dropout = nn.Dropout(DROPOUT)         self.conv2 = nn.Conv2d(planes, planes, kernel\_size=3,                                stride=1, padding=1, bias=False)         self.bn2 = nn.BatchNorm2d(planes)         self.dropout = nn.Dropout(DROPOUT)          self.shortcut = nn.Sequential()         if stride != 1 or in\_planes != self.expansion\*planes:             self.shortcut = nn.Sequential(                 nn.Conv2d(in\_planes, self.expansion\*planes,                           kernel\_size=1, stride=stride, bias=False),                 nn.BatchNorm2d(self.expansion\*planes),                 nn.Dropout(DROPOUT)             )      def forward(self, x):         out = F.relu(self.dropout(self.bn1(self.conv1(x))))         out = self.dropout(self.bn2(self.conv2(out)))         out += self.shortcut(x)         out = F.relu(out)         return out   class ResNet(nn.Module):     def \_\_init\_\_(self, block, num\_blocks, num\_classes=10):         super(ResNet, self).\_\_init\_\_()         self.in\_planes = 64          self.conv1 = nn.Conv2d(3, 64, kernel\_size=3,                                stride=1, padding=1, bias=False)         self.bn1 = nn.BatchNorm2d(64)         self.layer1 = self.\_make\_layer(block, 64, num\_blocks[0], stride=1)         self.layer2 = self.\_make\_layer(block, 128, num\_blocks[1], stride=2)         self.layer3 = self.\_make\_layer(block, 256, num\_blocks[2], stride=2)         self.layer4 = self.\_make\_layer(block, 512, num\_blocks[3], stride=2)         self.linear = nn.Linear(512\*block.expansion, num\_classes)      def \_make\_layer(self, block, planes, num\_blocks, stride):         strides = [stride] + [1]\*(num\_blocks-1)         layers = []         for stride in strides:             layers.append(block(self.in\_planes, planes, stride))             self.in\_planes = planes \* block.expansion         return nn.Sequential(\*layers)      def forward(self, x):         out = F.relu(self.bn1(self.conv1(x)))         out = self.layer1(out)         out = self.layer2(out)         out = self.layer3(out)         out = self.layer4(out)         out = F.avg\_pool2d(out, 4)         out = out.view(out.size(0), -1)         out = self.linear(out)         return F.log\_softmax(out, dim=-1)   def ResNet18():     return ResNet(BasicBlock, [2, 2, 2, 2]) *# 导入模型并打印摘要* model = ResNet18().to(device) summary(model, input\_size=(3,32,32)) from tqdm import tqdm from torch import nn import torch.nn from torch.functional import F import os   def model\_training(model, device, train\_dataloader, optimizer, train\_acc, train\_losses):                  model.train()     pbar = tqdm(train\_dataloader)     correct = 0     processed = 0     running\_loss = 0.0      for batch\_idx, (data, target) in enumerate(pbar):         data, target = data.to(device), target.to(device)         optimizer.zero\_grad()         y\_pred = model(data)         loss = F.nll\_loss(y\_pred, target)                  train\_losses.append(loss)         loss.backward()         optimizer.step()          pred = y\_pred.argmax(dim=1, keepdim=True)         correct += pred.eq(target.view\_as(pred)).sum().item()         processed += len(data)         *# 打印统计信息*         running\_loss += loss.item()         pbar.set\_description(desc=f'Loss={loss.item()} Batch\_id={batch\_idx} Accuracy={100\*correct/processed:0.2f}')         train\_acc.append(100\*correct/processed) import torch import os from torch.functional import F  cwd = os.getcwd()  def model\_testing(model, device, test\_dataloader, test\_acc, test\_losses, misclassified = []):          model.eval()     test\_loss = 0     correct = 0     class\_correct = list(0. for i in range(10))     class\_total = list(0. for i in range(10))     *# label = 0*     classes = ('plane', 'car', 'bird', 'cat',            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')          with torch.no\_grad():          for index, (data, target) in enumerate(test\_dataloader):             data, target = data.to(device), target.to(device)             output = model(data)             pred = output.argmax(dim=1, keepdim=True)                          for d,i,j in zip(data, pred, target):                 if i != j:                     misclassified.append([d.cpu(),i[0].cpu(),j.cpu()])              test\_loss += F.nll\_loss(output, target, reduction='sum').item()             correct += pred.eq(target.view\_as(pred)).sum().item()     test\_loss /= len(test\_dataloader.dataset)     test\_losses |