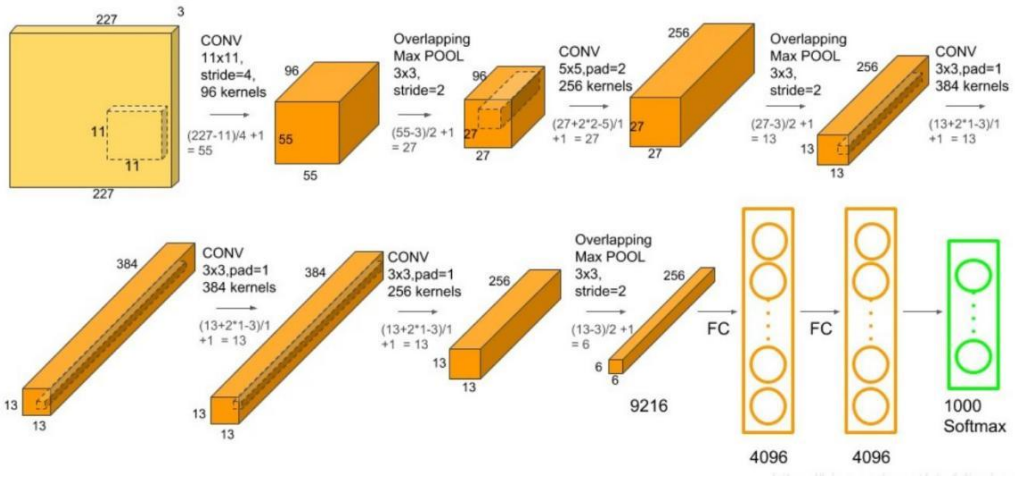
本文进一步根据 AlexNet 网络学习基本的卷积神经网络结构和代码练习，并通过学习 ImageFolder 这一读取本地数据集的方法探索数据的 label 在代码中的表示。

在 LeNet 提出后的将近 20 年里，神经网络一度被其他机器学习方法超越，如支持向量机。虽然 LeNet 可以在早期的小数据集上取得好的成绩，但是在更大的真实数据集上的表现并不尽如人意。一方面，神经网络计算复杂。虽然 20 世纪 90 年代也有过一些针对神经网络的加速硬件，但并没有像之后 GPU 那样大量普及。因此，训练一个多通道、多层和有大量参数的卷积神经网络在当年很难完成。另一方面，当年研究者还没有大量深入研究参数初始化和非凸优化算法等诸多领域，导致复杂的神经网络的训练通常较困难。

2012 年，AlexNet 横空出世。这个模型的名字来源于论文第一作者的姓名 Alex Krizhevsky [1]。AlexNet 使用了 8 层卷积神经网络，并以很大的优势赢得了 ImageNet 2012 图像识别挑战赛。它首次证明了学习到的特征可以超越手工设计的特征，从而一举打破计算机视觉研究的前状。

1.Alexnet 网络结构



整个 AlexNet 网络包括输入层×1、卷积层×5、池化层×3、全连接层×3，直接按照顺序连接，结构比较简单，下面详细看一下每层网络的特征图的尺寸通道数变化和参数计算。

1.1 图像尺寸通道变化

整个 AlexNet 网络包括输入层×1、卷积层×5、池化层×3、全连接层×3，根据前面所讲的卷积层和池化层图片尺寸变化计算公式，有

第 1 层输入层： 输入为 224×224×3 三通道的图像。

第 2 层 Conv 层： 输入为 224×224×3，经过 96 个 kernel size 为 11×11×3 的 filter, stride = 4，卷积后得到 shape 为 55×55×96 的特征图。

这里有 2 点值得注意的细节：(224-11)/4 + 1 = 54.25，两种解释：

（1）按照论文中 filter size 和 stride 的设计，输入的图片尺寸应该为 227×227×3。

（2）加上 padding=2，则(224-11+2*2)/4 + 1 = 55.25，步长会略去小数，得到 55.

第 3 层 Max-pooling 层： 输入为 55×55×96，经 Overlapping pooling(重叠池化)pool_size = 3,stride = 2 后得到尺寸为 27×27×96 的特征图

第 4 层 Conv 层： 输入尺寸为 27×27×96，经 256 个 5×5×96 的 filter 卷积，padding=same 得到尺寸为 27×27×256 的特征图。

第 5 层池化层： 输入为 27×27×256，，经 pool size = 3,stride = 2 的重叠池化，得到尺寸为 13×13×256 的特征图。

第 6～8 层 Conv 层： 第 6 层输入为 13×13×256，经 384 个 3×3×256 的 filter 卷积得到 13×13×384 的特征图。 第 7 层输入为 13×13×384，经 384 组 3×3×384 的 filter 卷积得到 13×13×384 的的特征图。 第 8 层输入为 13×13×384，经 256 个 3×3×384 的 filter 卷积得到 13×13×256 的的特征图。 这里可见，这三层卷积层使用的 kernel 前两个维度都是 3×3，只是通道维不同。

第 9 层 Max-pooling 层： 输入尺寸为 13×13×256，经 pool size = 3,stride = 2 的重叠池化得到尺寸为 6×6×256 的池化层。该层后面还有 flatten 操作，通过展平得到 6×6×256=9216 个特征后与全连接层相连。

第 10～12 层 Dense(全连接)层： 第 10~12 层神经元个数分别为 4096，4096,1000。其中前两层在使用 relu 后还使用了 Dropout 对神经元随机失活，最后一层全连接层用 softmax 输出 1000 个分类。（分类数量根据具体应用的数量变化，比如数据集中有 10 个类别，则最后输出 10）

1.2 参数计算

根据论文描述，AlexNet 中有 65 万个神经元，包括了 6000 万个参数。

网络层	卷积核尺寸&通道数	特征图尺寸&通道数	神经元数量（特征数）	参数数量（含bias）
2-Conv	11*11*3*96	55*55*96	55*55*96=290400	11*11*3*96+96=3, 4944
3-Pool		27*27*96	27*27*96=69984	0
4-Conv	5*5*96*256	27*27*256	27*27*256=186624	5*5*96*256+256=61, 4656
5-Pool		13*13*256	13*13*256=43264	0
6-Conv	3*3*256*384	13*13*384	13*13*384=64896	3*3*256*384+384=88, 5120
7-Conv	3*3*384*384	13*13*384	13*13*384=64896	3*3*384*384+384=132, 7488
8-Conv	3*3*384*256	13*13*256	13*13*256=43264	3*3*384*256+256=88, 4992
9-Pool		6*6*256	6*6*256=9216	0
10-Dense	/	1*1*4096	4096	9216*4096+4096=3775, 2832
11-Dense	/	1*1*4096	4096	4096*4096+4096=1678, 1312
12-Dense	/	1*1*1000	1000	1000*4096+1000=409, 7000
总计				62, 378, 344

由上表可以看出以下几点：

（1）本层的卷积核的意义是，上一层的特征图经过本层的卷积核得到本层的特征图；

（2）前面已经讲过，输出通道数就是卷积核的组数，每组卷积核数对应上层的输入通道数，所以卷积核参数应该是 kernel_size*kernel_size*in_channels*out_channels。

（3）此外，卷积核也有参数偏置项，对应的是卷积核输出通道数。所以每个通道卷积核对输入通道的特征图加权求和操作之后，再加上一个常数项，偏置项在激活函数之前。

（4）池化层不涉及参数；

（5）在进入全连接层之前，需要将前面的特征图展平，即 6*6*256=9216。

（6）这里可以更直观地看到，全连接层的参数量是巨大的，占了整个网络参数的一多半。

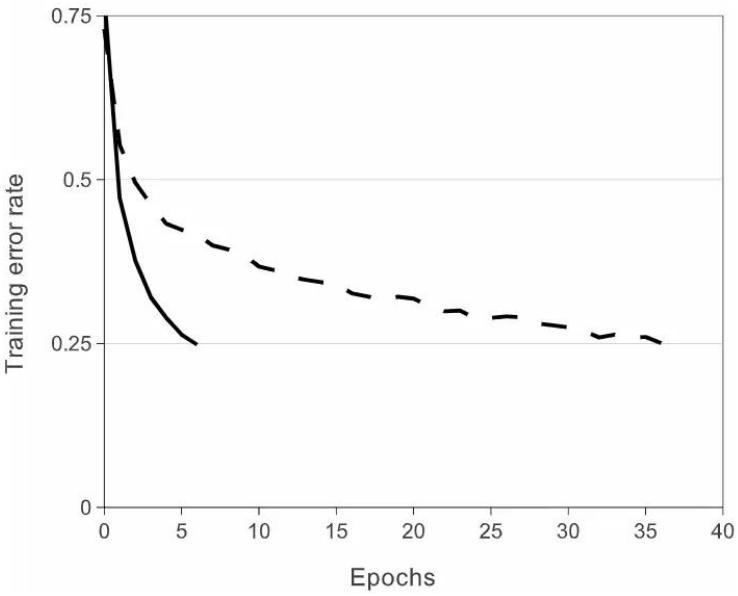
2 亮点与贡献

2.1 引入非线性激活函数 Relu

AlexNet 将 sigmoid 激活函数改成了更加简单的 ReLU 激活函数。一方面，ReLU 激活函数的计算更简单，例如它并没有 sigmoid 激活函数中的求幂运算。

另一方面，ReLU 激活函数在不同的参数初始化方法下使模型更容易训练。这是由于当 sigmoid 激活函数输出极接近 0 或 1 时，这些区域的梯度几乎为 0，从而造成反向传播无法继续更新部分模型参数；而 ReLU 激活函数在正区间的梯度恒为 1。因此，若模型参数初始化不当，sigmoid 函数可能在正区间得到几乎为 0 的梯度，从而令模型无法得到有效训练。

论文比较了在 CIFAR-10 数据集下，一个 4 层卷积的网络结构，在以 tanh 和 relu 作为激活单元时损失和训练轮数的关系，得出结论：**使用 ReLUs（实线）进行深度卷积神经网络训练的速度比使用 tanh（虚线）训练的速度**快 6 倍 。而且防止过拟合的作用比较好。



2.2 多 GPU 并行训练

论文中展示的多 GPU 并行训练是在当时 GPU 的内存较小的情况下，采用的无奈操作，得益于现在 GPU 内存的高速发展，对于一般的网络模型此类操作如今没有使用的必要了。不过目前来看随着大模型的发展，参数数量越来越多，GPU 并行训练还是很有必要的，后续将学习介绍 GPU 的并行训练。

简单举例来说，池化核大小 3 × 3， 扩充边缘 padding = 0，步长 stride = 2，因此其 FeatureMap 输出大小为(13-3+0×2+2)/2=6, 即 C5 输出为 6×6×256（此处未将输出分到两个 GPU 中，若按照论文将分成两组，每组为 6×6×128）；

2.3 Local Response Normalization LRN

Local Response Normalization(LRN)技术主要是深度学习训练时的一种提高准确度的技术方法。其中 caffe、tensorflow 等里面是很常见的方法，其跟激活函数是有区别的，LRN 一般是在激活、池化后进行的一种处理方法。LRN 归一化

技术首次在 AlexNet 模型中提出这个概念。通过实验确实证明它可以提高模型的泛化能力，但是提升的很少，以至于后面不再使用。

论文中提出，**采用 LRN 局部响应正则化可以提高模型的泛化能力**，所以在某些层应用了 ReLU 后，加上了 LRN 使得 top-1 和 top-5 的错误率分别降低了 1.4% 和 1.2%。**但是由于实际作用不大（在 VGG 网络中，作了对比，发现效果并不显著），后面 LRN 的方式并没有被人们广泛使用。**

2.4 Overlapping Pooling

论文中的 Maxpooling 采用的是重叠池化，传统池化时卷积核提取的每一小块特征不会发生重合，譬如 kernel size 记为 k×k,步长 stride 记为 s,当 k = s 时，就不会发生重合，当 k> s 时，就会发生重合，即重叠池化。**AlexNet 训练中的 stride 为 2，kernel size 为 k = 3，使用的就是重叠池化。**

以往池化的大小 PoolingSize 与步长 stride 一般是相等的，例如：图像大小为 256*256，PoolingSize=2×2，stride=2，这样可以使图像或是 FeatureMap 大小缩小一倍变为 128，此时池化过程没有发生层叠。但是 AlexNet 采用了层叠池化操作，即 PoolingSize > stride。这种操作非常像卷积操作，可以使相邻像素间产生信息交互和保留必要的联系。论文中也证明，此操作可以有效防止过拟合的发生。

2.5. 降低过拟合

2.5.1 数据扩增

为了降低过拟合，提高模型的鲁棒性，这里**采用了两种 Data Augmentation 数据扩增方式：**

- 1.生成图像平移和水平反射。通过从 256×256 幅图像中提取随机 224×224 块图像(及其水平反射)，并在这些提取的图像上训练网络。这将训练集的大小增加了 2048 倍
- 2.改变了训练图像中 RGB 通道的强度。在整个 imagenet 训练集中对 RGB 像素值集执行 PCA 操作

2.5.2 Dropout

训练采用了 0.5 丢弃率的传统 Dropout，对于使用了 Dropout 的 layer 中的每个神经元，训练时都有 50%的概率被丢弃。所以每次输入时，神经网络都会对不同的结构进行采样，但是所有这些结构都共享权重。

这种技术减少了神经元之间复杂的相互适应，因为神经元不能依赖于其他神经元的存在。因此，它被迫获得更健壮的特征。测试时使用所有的神经元，但将

它们的输出乘以 0.5。 **论文中还提到了：Dropout 使收敛所需的迭代次数增加了一倍**

2.6.网络层数的增加

与原始的 LeNet 相比，AlexNet 网络结构更深，LeNet 为 5 层，AlexNet 为 8 层。在随后的神经网络发展过程中，AlexNet 逐渐让研究人员认识到网络深度对性能的巨大影响。当然，这种思考的重要节点出现在 VGG 网络，但是很显然从 AlexNet 为起点就已经开始了这项工作。

3.超参数与代码

训练使用的是小批量随机梯度下降，batch size = 128,动量 momentum 为 0.9，weight decay 权重衰减为 0.0005；每一层的权重初始化为均值 1 标准差 0.01 的正态分布，在第 2,4,5 卷积层和全连接层中的 bias 初始化为常数 1，其余层则为 0. 所有层采用了相同的初始化为 0.01 的学习率，不过可以手动调整。整个训练过程在两台 NVIDIA GTX 580 3GB gpu 上用了 5 到 6 天的时间，120 万张图像的训练集，大约 90 轮迭代。 论文中提到：weight decay 对模型的学习很重要。换句话说，这里的重量衰减不仅仅是一个正则化器:它减少了模型的训练误差。

model.py

#此部分代码结合前面的图像维度变化来看，相比于之前介绍的 LeNet，特殊之处在于全连接层前加了 dropout，激活函数使用的 ReLu。

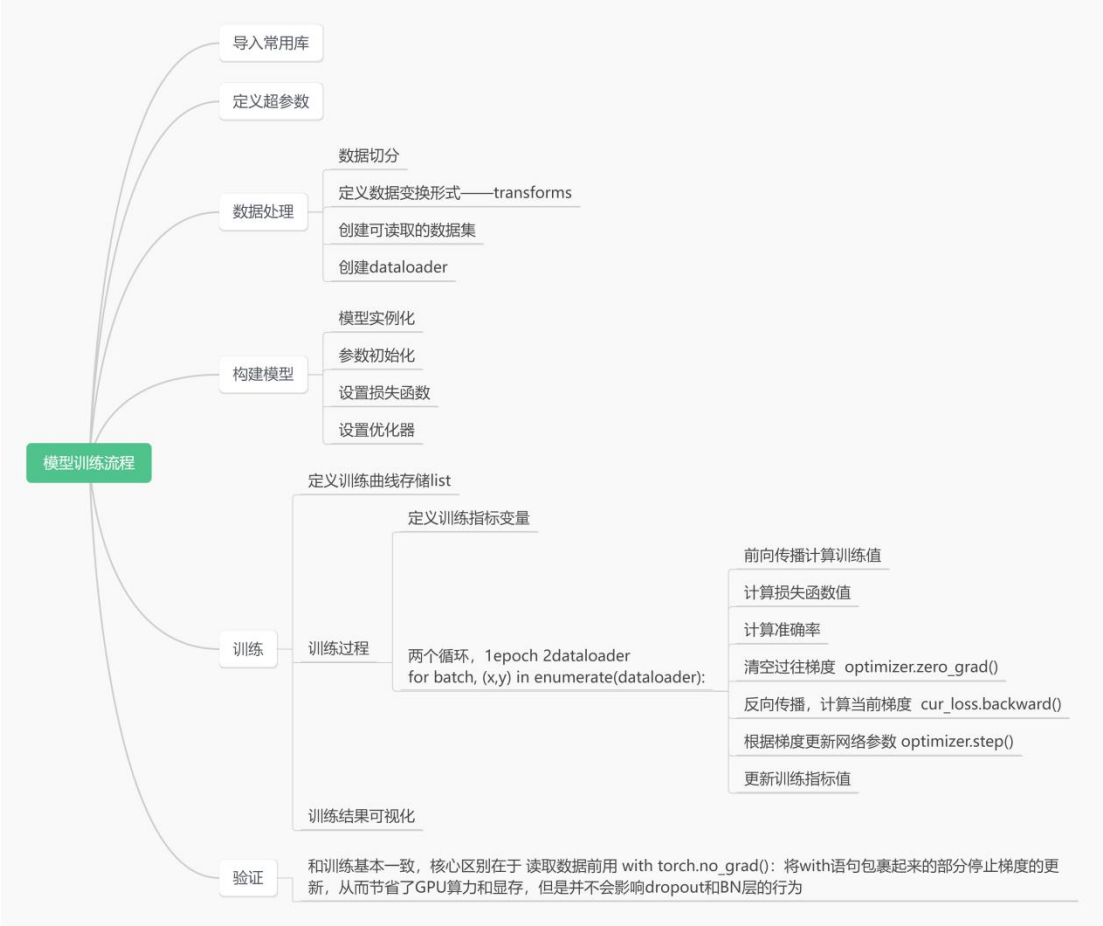
```
import torch
from torch import nn
import torch.nn.functional as F
import random

# 定义 AlexNet 网络模型
# AlexNet（子类）继承 nn.Module（父类）
class AlexNet(nn.Module):
    # 子类继承中重新定义 Module 类的__init__()和 forward()函数，这也是网络模型必须包含的两个函数
    # init()函数：进行初始化，申明模型中各层的定义
    def __init__(self):
        # super：引入父类的初始化方法给子类进行初始化
        super(AlexNet,self).__init__()
        # 卷积层，输入大小为 224*224，输出大小为 55*55，输入通道为 3，
```

输出通道为 96，卷积核尺寸为 11，扩充边缘为 2

```
self.c1=nn.Conv2d(in_channels=3,out_channels=96,kernel_size=11,stride=4,padding=2)
        #引入了 ReLu 激活函数
        self.Relu=nn.ReLU()
        #最大池化层，输入为 55×55×96，经 Overlapping pooling(重叠池化)pool_size = 3,stride = 2 后得到尺寸为 27×27×96  的特征图



        self.s2=nn.MaxPool2d(kernel_size=3,stride=2)
        #Conv 层： 输入尺寸为 27×27×96，经 256 个 5×5×96 的 filter 卷积，padding=same 得到尺寸 27×27×256。

self.c3=nn.Conv2d(in_channels=96,out_channels=256,kernel_size=5,stride=1,padding=2)
       #池化层： 输入为 27×27×256，，经 pool size = 3,stride = 2 的重叠池化，得到尺寸为 13×13×256 的池化层。
        self.s4= nn.MaxPool2d(kernel_size=3, stride=2)
        #输入为 13×13×256，经 384 个 3×3×256 的 filter 卷积得到 13×13×384 的卷积层。

self.c5=nn.Conv2d(in_channels=256,out_channels=384,kernel_size=3,stride=1,padding=1)
        #输入为 13×13×384，经 384 组 3×3×384 的 filter 卷积得到 13×13×384
        self.c6 = nn.Conv2d(in_channels=384, out_channels=384, kernel_size=3, stride=1,padding=1)
        #输入为 13×13×384，经 256 个 3×3×384 的 filter 卷积得到 13×13×256
        self.c7 = nn.Conv2d(in_channels=384, out_channels=256, kernel_size=3,stride=1, padding=1)
        #输入尺寸为 13×13×256，经 pool size = 3,stride = 2 的重叠池化得到尺寸为 6×6×256
        self.s8 = nn.MaxPool2d(kernel_size=3, stride=2)
        #展平 flatten，通过展平得到 6×6×256=9216 个特征后与之后的全连接层相连
        self.flatten=nn.Flatten()
        #第 10~12 层神经元个数分别为 4096，4096,1000。其中前两层在使用 relu 后还使用了 Dropout 对神经元随机失活，
        # 最后一层全连接层用 softmax 输出 1000 个分类（分类数量根据具体应用的数量变化，比如数据集中有 10 个类别，则最后输出 10）
        self.f9=nn.Linear(in_features=6*6*256,out_features=4096)
        self.f10=nn.Linear(in_features=4096,out_features=4096)
        self.output=nn.Linear(in_features=4096,out_features=7)
```

```
    # forward()：定义前向传播过程,描述了各层之间的连接关系
    def forward(self,x):
        x=self.Relu(self.c1(x))
        x=self.s2(x)
        x=self.Relu(self.c3(x))
        x=self.s4(x)
        x=self.Relu(self.c5(x))
        x=self.Relu(self.c6(x))
        x=self.Relu(self.c7(x))
        x = self.s8(x)
        x=self.flatten(x)
        x = F.dropout(x, p=0.5)
        x=self.f9(x)
        x=F.dropout(x,p=0.5)
        x = self.f10(x)
        x = F.dropout(x, p=0.5)
        output=self.output(x)
        return output

        # 测试代码
        # 每个 python 模块（python 文件）都包含内置的变量 __name__，当该模块被直接执行的时候，__name__ 等于文件名（包含后缀 .py ）
        # 如果该模块 import 到其他模块中，则该模块的 __name__ 等于模块名称（不包含后缀.py）
        #“__main__”始终指当前执行模块的名称（包含后缀.py）
        # if 确保只有单独运行该模块时，此表达式才成立，才可以进入此判断语法，执行其中的测试代码，反之不行
if __name__=="__main__":
    #rand:返回一个张量，包含了从区间[0, 1)的均匀分布中抽取的一组随机数，此处为四维张量
    x=torch.randn([1,3,227,227])
    # # 模型实例化
    model=AlexNet()
    y=model(x)
    print(y.size())
```

train.py



#本次训练代码和前面 LeNet 的主要区别在于数据集的读取，LeNet 练习时直接从下载 mnist，本次练习从本地文件夹中使用 ImageFolder 读取。
#ImageFolder(root, transform``=``None``, target_transform``=``None``, loader``=``default_loader)
#root 指定路径加载图片； transform：对 PIL Image 进行的转换操作，transform 的输入是使用 loader 读取图片的返回对象
#target_transform：对 label 的转换 loader：给定路径后如何读取图片，默认读取为 RGB 格式的 PIL Image 对象
#label 是按照文件夹名顺序排序后存成字典，即{类名:类序号(从 0 开始)}，一般来说最好直接将文件夹命名为从 0 开始的数字，举例来说，两个类别，
#狗和猫，把狗的图片放到文件夹名为 0 下；猫的图片放到文件夹名为 1 的下面。
这样会和 ImageFolder 实际的 label 一致， 如果不是这种命名规范，建议看看 self.class_to_idx 属性以了解 label 和文件夹名的映射关系

#从自己创建的 models 库里导入 AlexNet 模块
#import AlexNet 仅仅是把 AlexNet.py 导入进来,当我们创建 AlexNet 的实例的时候需要通过指定 AlexNet.py 中的具体类.
#例如:我的 AlexNet.py 中的类名是 AlexNet,则后面的模型实例化 AlexNet 需要通过**AlexNet.AlexNet()**来操作

```
#还可以通过 from 还可以通过 from AlexNet import * 直接把 AlexNet.py 中除了以 _ 开头的内容都导入
from models.cv.AlexNet import *
# lr_scheduler：提供一些根据 epoch 训练次数来调整学习率的方法
from torch.optim import lr_scheduler
# torchvision：PyTorch 的一个图形库，服务于 PyTorch 深度学习框架的，主要用来构建计算机视觉模型
# transforms：主要是用于常见的一些图形变换
# datasets：包含加载数据的函数及常用的数据集接口
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
# os：operating system（操作系统），os 模块封装了常见的文件和目录操作
import os
#导入画图的库，后面将主要学习使用 axes 方法来画图
import matplotlib.pyplot as plt

# 设置数据转化方式，如数据转化为 Tensor 格式，数据切割等
# Compose()：将多个 transforms 的操作整合在一起
# ToTensor(): 将 numpy 的 ndarray 或 PIL.Image 读的图片转换成形状为(C,H, W)的 Tensor 格式，且归一化到[0,1.0]之间
#compose 的参数为列表[]
train_transform=transforms.Compose([
    transforms.Resize([224,224]),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    #normalize 的意义
    transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])
])

test_transform=transforms.Compose([
    transforms.Resize([224,224]),
    transforms.ToTensor(),
    transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])
])
#ImageFolder(root, transform``=``None``, target_transform``=``None``, loader``=``default_loader)
#root 指定路径加载图片； transform：对 PIL Image 进行的转换操作，transform 的输入是使用 loader 读取图片的返回对象
#target_transform：对 label 的转换 loader：给定路径后如何读取图片，默认读取为 RGB 格式的 PIL Image 对象
#label 是按照文件夹名顺序排序后存成字典，即{类名:类序号(从 0 开始)}，一般来说最好直接将文件夹命名为从 0 开始的数字，举例来说，两个类别，
#狗和猫，把狗的图片放到文件夹名为 0 下；猫的图片放到文件夹名为 1 的下面。
```

```
# 这样会和 ImageFolder 实际的 label 一致， 如果不是这种命名规范，建议看看 self.class_to_idx 属性以了解 label 和文件夹名的映射关系
#python 中\是转义字符，Windows 路径如果只有一个\，会把它识别为转义字符。#可以用 r"把它转为原始字符，也可以用\\,也可以用 Linux 的路径字符/。
train_dataset=ImageFolder(r"E:\计算机\data\fer2013_数据增强版本\train",train_transform)
test_dataset=ImageFolder(r"E:\计算机\data\fer2013_数据增强版本\test",test_transform)

# DataLoader：将读取的数据按照 batch size 大小封装并行训练
# dataset (Dataset)：加载的数据集
# batch_size (int, optional)：每个 batch 加载多少个样本(默认: 1)
# shuffle (bool, optional)：设置为 True 时会在每个 epoch 重新打乱数据(默认: False)
train_dataloader=DataLoader(train_dataset,batch_size=16,shuffle=True)
test_dataloader=DataLoader(test_dataset,batch_size=16,shuffle=True)

device='cuda' if torch.cuda.is_available() else 'cpu'

model=AlexNet().to(device)
# 定义损失函数（交叉熵损失）
loss_fn=nn.CrossEntropyLoss()
# 定义优化器(随机梯度下降法)
# params(iterable)：要训练的参数，一般传入的是 model.parameters()
# lr(float)：learning_rate 学习率，也就是步长
# momentum(float, 可选)：动量因子（默认：0），矫正优化率
optimizer=torch.optim.SGD(model.parameters(),lr=0.01,momentum=0.9)

# 学习率，每隔 10 轮变为原来的 0.1
# StepLR：用于调整学习率，一般情况下会设置随着 epoch 的增大而逐渐减小学习率从而达到更好的训练效果
# optimizer （Optimizer）：需要更改学习率的优化器
# step_size（int）：每训练 step_size 个 epoch，更新一次参数
# gamma（float）：更新 lr 的乘法因子
lr_scheduler=lr_scheduler.StepLR(optimizer,step_size=10,gamma=0.5)

def train(train_dataloader,model,loss_fn,optimizer):
    loss,acc,n=0.0,0.0,0
    # dataloader: 传入数据（数据包括：训练数据和标签）
    # enumerate()：用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，一般用在 for 循环当中
    # enumerate 返回值有两个：一个是序号，一个是数据（包含训练数据和标签）
    # x：训练数据（inputs）(tensor 类型的），y：标签（labels）(tensor 类型的）
    #和 dataloader 结合使用时返回数据下标是 batch（在创建 dataloader 时会把
```

```
batch size 作为参数传入），
    # 从 0 开始，最大数为样本总数除以 batch size 大小，数据是一 batch 的数据和标签
    for batch,(x,y) in enumerate(train_dataloader):
        x,y=x.to(device),y.to(device)
        #print("x-shape",x.size())
        output=model(x)
        cur_loss=loss_fn(output,y)
        # torch.max(input, dim)函数
        # input 是具体的 tensor，dim 是 max 函数索引的维度，0 是每列的最大值，1 是每行的最大值输出
        # 函数会返回两个 tensor，第一个 tensor 是每行的最大值；第二个 tensor 是每行最大值的索引
        _,pred=torch.max(output,axis=1)
        # 计算每批次的准确率
        # output.shape[0]一维长度为该批次的数量
        # torch.sum()对输入的 tensor 数据的某一维度求和
        cur_acc=torch.sum(pred==y)/output.shape[0]

        #清除过往梯度值
        optimizer.zero_grad()
        #后向传播
        cur_loss.backward()
        #优化迭代
        optimizer.step()

        loss+=cur_loss.item()
        acc+=cur_acc.item()
        n=n+1
    train_loss=loss/n
    train_acc=acc/n
    # 计算训练的损失函数变化
    print('train_loss==' + str(train_loss))
    # 计算训练的准确率
    print('train_acc' + str(train_acc))
    return train_loss,train_acc

#测试函数里参数无优化器，不需要再训练，只需要测试和验证即可
def test(dataloader,model,loss_fn):
    loss,acc,n=0.0,0.0,0

    ## model.eval()：设置为验证模式，如果模型中有 Batch Normalization 或 Dropout，则不启用，以防改变权值
    model.eval()
```

```
    #with torch.no_grad()：将 with 语句包裹起来的部分停止梯度的更新，从而节省了 GPU 算力和显存，但是并不会影响 dropout 和 BN 层的行为
    with torch.no_grad():
        for batch,(x,y) in enumerate(dataloader):
            x,y=x.to(device),y.to(device)
            output=model(x)
            cur_loss=loss_fn(output,y)
            _,pred=torch.max(output,axis=1)
            cur_acc=torch.sum(pred==y)/output.shape[0]
            loss += cur_loss.item()
            acc += cur_acc.item()
            n = n + 1
        test_loss=loss/n
        test_acc=acc/n
        print('test_loss==' + str(test_loss))
        # 计算训练的准确率
        print('train_acc' + str(test_acc))
        return test_loss, test_acc

    # 定义画图函数
    # 错误率
def matplot_loss(train_loss, test_loss):
        # 参数 label = "传入字符串类型的值，也就是图例的名称
    fig,ax=plt.subplots(1,1)
    ax.plot(train_loss, label='train_loss')
    ax.plot(test_loss, label='test_loss')
        # loc 代表了图例在整个坐标轴平面中的位置（一般选取'best'这个参数值）
    ax.legend(loc='best')
    ax.set_xlabel('loss')
    ax.set_ylabel('epoch')
    ax.set_title("训练集和验证集的 loss 值对比图")
    plt.show()

    # 准确率
def matplot_acc(train_acc, test_acc):
    fig, ax = plt.subplots(1,1)
    ax.plot(train_acc, label='train_acc')
    ax.plot(test_acc, label='test_acc')
    ax.legend(loc='best')
    ax.set_xlabel('acc')
    ax.set_ylabel('epoch')
    ax.set_title("训练集和验证集的 acc 值对比图")
    plt.show()
```

```
loss_train=[]
acc_train=[]
loss_test=[]
acc_test=[]

epoch=1
min_acc=0
for t in range(epoch):
    lr_scheduler.step()
    print(f"epcoh{t+1}\n-------")
    train_loss,train_acc=train(train_dataloader,model,loss_fn,optimizer)
    test_loss,test_acc=test(test_dataloader,model,loss_fn)

    loss_train.append(train_loss)
    acc_train.append(train_acc)

    loss_test.append(test_loss)
    acc_test.append(test_acc)

    if test_acc>min_acc:
        folder='save_model'
        if not os.path.exists(folder):
            os.mkdir("../save_model")
        min_acc=test_acc
        print(f"save model {t+1}轮")
        torch.save(model.state_dict(), '../save_model/alexnet-best-model.pth')
    if t==epoch-1:
        torch.save(model.state_dict(), '../save_model/alexnet-best-model.pth')
matplot_loss(loss_train,loss_test)
matplot_acc(acc_train,acc_test)
```

参考资料

https://zh.d2l.ai/index.html

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).