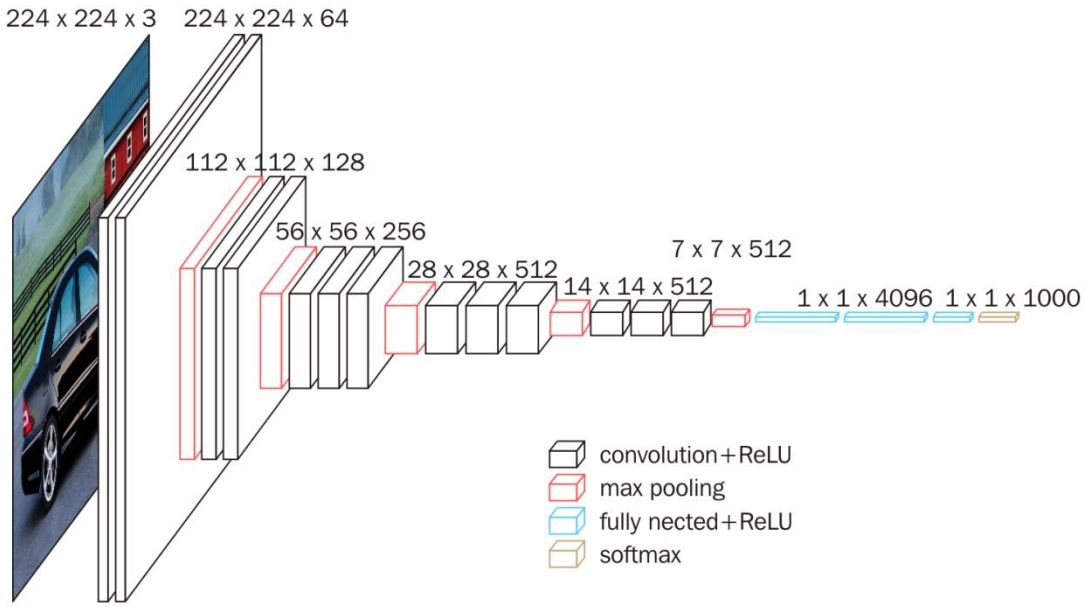
AlexNet 在 LeNet 的基础上增加了 3 个卷积层，在 ImageNet 数据集上的优异表现表明了增加卷积神经网络的深度可以提升任务效果，但并没有说明该如何设计网络结构以增加深度。

本节学习的 VGG 网络，名字来源于论文作者所在的实验室 Visual Geometry Group ，提出了可以通过重复使用简单的基础块来构建深度模型的思路。接下来详细介绍。

1.VGG 网络模型结构

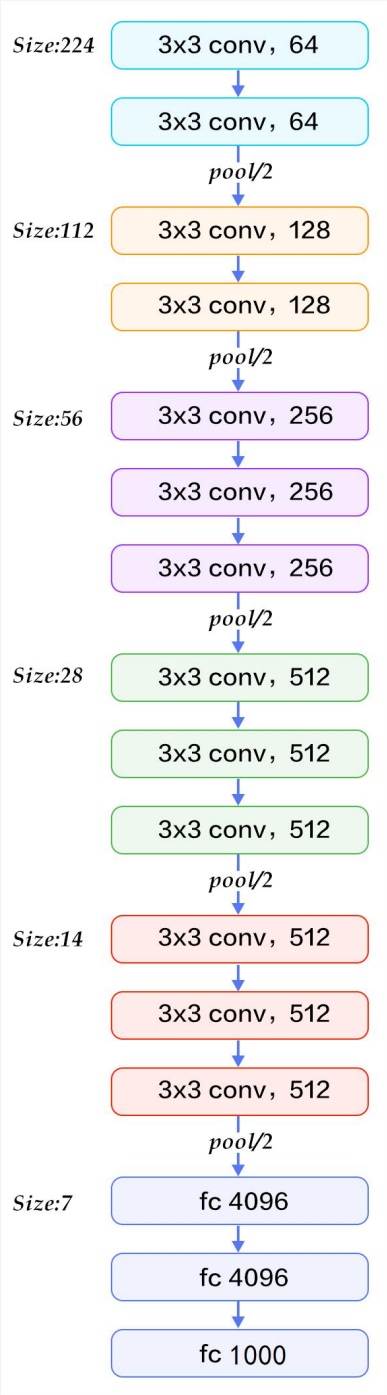


VGG 网络层的深度是可调整的，其核心在于设计了多组卷积块，每组卷积块由多个相同的小尺寸（一般为 3*3）的卷积核组成，卷积核填充为 1，卷积核保持输入特征图的高和宽不变，然后再接上一个尺寸为 2*2，步长也为 2 的池化层，用以将特征图尺寸减半。

论文还表明了采用堆积的小卷积核效果优于采用大的卷积核，因为可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）。例如，在 VGG 中，使用了 3 个 3x3 卷积核（27 个参数）来代替 7x7 卷积核（49 个参数），使用了 2 个 3x3 卷积核来代替 5*5 卷积核，这样做的主要目的是在保证具有相同感知野的条件下，提升了网络的深度，在一定程度上提升了神经网络的效果。

1.1 特征图尺寸和通道数计算

VGG 由 5 组卷积块（一组卷积块中会卷积不同的次数，包含不同的卷积层数和一个池化层）、3 层全连接层、softmax 输出层构成，层与层之间使用 max-pooling（最大化池）分开，所有隐层的激活单元都采用 ReLU 函数。
以常用的 VGG-16 为例：



网络结构中所有的 maxpooling（最大化池化）：滤波器尺寸为 2*2，步长为 2，特征图尺寸减半。

卷积块 1）输入图像尺寸：224*224*3；卷积核：3*3*3*64，步长为 1，padding=same 填充，卷积两次，再经 ReLU 激活函数后特征图的尺寸：224*224*64；池化后的特征图尺寸变为 112*112*64；

卷积块 2）经 3*3*64*128 的卷积核，两次卷积，ReLU 激活，尺寸变为 112*112*128；ma* pooling 池化，尺寸变为 56*56*128；

卷积块 3）经 3*3*128*256 的卷积核，三次卷积，ReLU 激活，尺寸变为 56*56*256；池化后尺寸变为 28*28*256；

卷积块 4）经 3*3*256*512 的卷积核，三次卷积，ReLU 激活，尺寸变为 28*28*512；池化后尺寸变为 14*14*512；

卷积块 5）经 3*3*512*512 的卷积核，三次卷积，ReLU，尺寸变为 14*14*512；池化后尺寸变为 7*7*512

全连接层）将二维的特征图展平 Flatten()，变成一维 512*7*7=25088。再经过两层特征数为 4096，一层 1000 的全连接层（共三层），经 ReLU 激活；最后通过 softmax 输出 1000 类结果的预测概率值

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

1.2 参数计算

以 VGG-16 为例，

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	3*3*3*64	224*224*64	224*224*64=3, 211, 264	3*3*3*64+64=1792
2-Conv	3*3*64*64	224*224*64	224*224*64=3, 211, 264	3*3*64*64+64=36928
2-Pool		112*112*64	112*112*64=802816	0
3-Conv	3*3*64*128	112*112*128	112*112*128=1605632	3*3*64*128+128=73856
3-Conv	3*3*128*128	112*112*128	112*112*128=1605632	3*3*128*128+128=147584
3-Pool		56*56*128	56*56*128=401408	0
4-Conv	3*3*128*256	56*56*256	56*56*256=802816	3*3*128*256+256=295168
4-Conv	3*3*256*256	56*56*256	56*56*256=802816	3*3*256*256+256=590080
4-Conv	3*3*256*256	56*56*256	56*56*256=802816	3*3*256*256+256=590080
4-Pool		28*28*256	28*28*256=200768	0
5-Conv	3*3*256*512	28*28*512	28*28*512=401408	3*3*256*512+512=1180160
5-Conv	3*3*512*512	28*28*512	28*28*512=401408	3*3*512*512+512=2359808
5-Conv	3*3*512*512	28*28*512	28*28*512=401408	3*3*512*512+512=2359808
5-Pool		14*14*512	14*14*512=100352	0
6-Conv	3*3*512*512	14*14*512	14*14*512=100352	3*3*512*512+512=2359808
6-Conv	3*3*512*512	14*14*512	14*14*512=100352	3*3*512*512+512=2359808
6-Conv	3*3*512*512	14*14*512	14*14*512=100352	3*3*512*512+512=2359808
6-Pool		7*7*512	7*7*512=25088	0
7-Dense	/	7*7*512=25088	4096	25088*4096+4096=102764544
8-Dense	/	4096	4096	4096*4096+4096=1678, 1312
9-Dense	/	1000	1000	4096*1000+1000=409, 7000
总计				138, 357, 544

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 7*7*512=25088。

（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。

神经元数量是图像的尺度信息，是计算时占用内存的空间大小（memory）；参数数量对于卷积层是卷积核的尺寸，通道数与数量计算得到的，对于全连接层则是前后两层神经元的数量的乘积。

VGG16 具有如此之大的参数数目，可以预期它具有很高的拟合能力；但同时缺点也很明显：

- 即训练时间过长，调参难度大。
- 需要的存储容量大，不利于部署。例如存储 VGG16 权重值文件的大小为 500 多 MB，不利于安装到嵌入式系统中。

2 亮点与贡献

2.1、小卷积核和多卷积子层

VGG 使用多个较小卷积核（3x3）的卷积层代替一个卷积核较大的卷积层，一方面可以减少参数，另一方面相当于进行了更多的非线性映射，可以增加网络的拟合/表达能力。

VGG 的作者认为两个 3x3 的卷积堆叠获得的感受野大小，相当一个 5x5 的卷积；而 3 个 3x3 卷积的堆叠（卷积核级联）获取到的感受野相当于一个 7x7 的卷积。这样可以增加非线性映射，也能很好地减少参数（例如 7x7 的参数为

49 个，而 3 个 3x3 的参数为 27）。

2.2、小池化核

相比 AlexNet 的 3x3 的池化核，VGG 全部采用 2x2 的池化核。

2.3 通道数多

VGG 网络第一层的通道数为 64，后面每层都进行了翻倍，最多到 512 个通道，通道数的增加，使得更多的信息可以被提取出来。每一层通道相当于提取一种特征信息。

2.4、层数更深、特征图更宽

由于卷积核专注于扩大通道数、池化专注于缩小宽和高，使得模型架构上更深更宽的同时，控制了计算量的增加规模。

2.5 优缺点

VGG 优点：

- VGGNet 的结构非常简洁，整个网络都使用了同样大小的卷积核尺寸（3x3）和最大池化尺寸（2x2）。
- 几个小滤波器（3x3）卷积层的组合比一个大滤波器（5x5 或 7x7）卷积层好；
- 验证了通过不断加深网络结构可以提升性能。

VGG 缺点：

从参数计算那节可以看出，VGG 耗费更多计算资源，并且使用了更多的参数（这里不是 3x3 卷积的锅），导致更多的内存占用（140M）。其中绝大多数的参数都是来自于第一个全连接层。且 VGG 有 3 个全连接层。

3. 代码

model.py

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

```
import random

# 定义 VGG 网络模型
# VGG（子类）继承 nn.Module（父类）
class VGG(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 类的 init 函数里传需要预定义的实参数，类里的其它函数也可以使用，这里可以定义一些类的其它函数或变量
    # features：make_features(cfg: list)生成提取特征的网络结构，这里是将重复的网络层抽象出统一的结构，可以直接改变参数生成不同的网络层
    # num_classes：需要分类的类别个数
    # init_weights：是否对网络进行权重初始化
    def __init__(self,features,num_classes=1000,init_weights=False):
        # super：引入父类的初始化方法给子类进行初始化
        super(VGG,self).__init__()
        # 生成提取特征的网络结构,这里直接调用传来的参数 features
        self.features=features
        # 生成分类的网络结构
        # Sequential：自定义顺序连接成模型，生成网络结构
        #网络层的一种组织形式，连续形，和前面直接按照顺序排列网络层相同，这里利用 Sequential 将网络层组织起来
        self.classifier=nn.Sequential(
            # 这里等价于前面 Lenet 和 Alexnet 介绍的 init 里初始化网络层，只不过这里没有将网络层赋给某个变量，而是统一保存后将其赋给 classifier
            # Dropout：随机地将输入中 50%的神经元激活设为 0，即去掉了一些神经节点，防止过拟合
            nn.Dropout(p=0.5),
            nn.Linear(in_features=7*7*512,out_features=4096),
            # ReLU(inplace=True)：将 tensor 直接修改，不找变量做中间的传递，节省运算内存，不用多存储额外的变量
            nn.ReLU(inplace=True),

            nn.Dropout(p=0.5),
            nn.Linear(in_features=4096,out_features=4096),
            nn.ReLU(inplace=True),

            nn.Dropout(p=0.5),
            nn.Linear(in_features=4096, out_features=num_classes),
        )
        # 如果为真，则对网络参数进行初始化
        if init_weights:
            self._initialize_weights()
```

```
    # 在 init 函数里还根据传入的参数判断是否网络参数初始化，这个是 LeNet 和 AlexNet 代码里未体现的，但是好的参数初始化可以提高网络效果，加快网络收敛
    # 网络参数初始化的方法前面已经介绍过，这里使用其中一种 xaiver

    def _initialize_weights(self):
        # 初始化需要对每一个网络层的参数进行操作，所以利用继承 nn.Module 类中的一个方法:self.modules()遍历返回所有 module
        for m in self.modules():
            # isinstance(object, type)：如果指定对象是指定类型，则 isinstance()函数返回 True
            # 如果是卷积层
            if isinstance(m, nn.Conv2d):
                # 首先是对权重参数进行初始化，uniform_(tensor, a=0, b=1)：服从~U(a,b)均匀分布，进行初始化
                nn.init.xavier_uniform_(m.weight)
                # 然后如果存在偏置参数，一般将其初始化为 0
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            # 如果是全连接层
            elif isinstance(m, nn.Linear):
                # 权重参数正态分布初始化
                nn.init.xavier_uniform_(m.weight)
                # 偏置参数初始化为 0
                nn.init.constant_(m.bias, 0)

    # 处理好网络层的定义和网络参数的初始化，便可以和前面那样利用 forward()函数定义前向传播过程， 描述各层之间的连接关系
    def forward(self, x):
        # 将数据输入至提取特征的网络结构，N x 3 x 224 x 224，最笨的方法是按照之前学习的，将所有网络层一一列出来，但这样一方面太过冗杂，另一方面灵活性太差。
        x = self.features(x)
        # N x 512 x 7 x 7
        # 图像经过提取特征网络结构之后，得到一个 7*7*512 的特征矩阵，进行展平
        # Flatten()：将张量（多维数组）平坦化处理，神经网络中第 0 维表示的是 batch_size，所以 Flatten()默认从第二维开始平坦化
        x = torch.flatten(x, start_dim=1)
        # 将数据输入分类网络结构，N x 512*7*7
        x = self.classifier(x)
        return x

#init 函数里直接利用传入的参数定义了特提取网络，这里要定义如何创建,
```

```
# 之所以单独用一个函数定义，是因为 vgg 有多种配置，需要根据配置创建不同的网络结构，而配置则是用列表逐一描述了网络层的类型和通道数
# 定义 cfgs 字典文件，每一个 key-value 对代表一个模型的配置文件，在模型实例化时，我们根据选用的模型名称 key，将对应的值-配置列表作为参数传到函数里
# 如：VGG11 代表 A 配置，也就是一个 11 层的网络， 数字代表卷积层中卷积核的个数，'M'代表池化层
# 通过函数 make_features(cfg: list)生成提取特征网络结构
cfgs = {
    'vgg11': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'vgg13': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'vgg16': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'],
    'vgg19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'],
}
def make_features(cfg:list):
    #首先定义一个保存网络结构的空列表
    layers=[]
    #根据最初的输入图像通道数定义，一般是 RGB 3 通道
    in_channels=3
    #然后遍历配置表，根据遇到的情况（池化层还是卷积层）增加不同的网络层结构
    for v in cfg:
        # 如果列表的值是 M 字符，说明该层是最大池化层
        if v=="M":
            # 创建一个最大池化层，在 VGG 中所有的最大池化层的 kernel_size=2，stride=2
            layers+=[nn.MaxPool2d(kernel_size=2,stride=2)]
        # 否则是卷积层
        else:
            # 在 Vgg 中，所有的卷积层的 kernel_size=3,padding=1，stride=1
            conv2d=nn.Conv2d(in_channels, v, kernel_size=3, stride=1,padding=1)
            # 将卷积层和 ReLU 放入列表
            layers+=[conv2d,nn.ReLU(True)]
            #网络列表每加一层，本层输入通道数都要改成上层的输出通道数
            in_channels=v
    # 将列表通过非关键字参数的形式返回，*layers 可以接收任意数量的参数
    return nn.Sequential(*layers)


#vgg 实例化和 LeNet,AlexNet 有点不同，因为要先手动选择网络名称，以 VGG16 为例，定义如下
# **kwargs 表示可变长度的字典变量，在调用 VGG 函数时传入的字典变量
```

```
def vgg(model_name="vgg16",**kwargs):
    # 如果 model_name 不在 cfgs，序会抛出 AssertionError 错误，报错为参数内容“ ”
    assert model_name in cfgs, "Warning: model number {} not in cfgs dict!".format(model_name)
    cfg=cfgs[model_name]
    # 这个字典变量包含了分类的个数以及是否初始化权重的布尔变量
    model=VGG(make_features(cfg),**kwargs)
    return model

# 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__":
    x=torch.randn([1,3,224,224])
    model=vgg("vgg16")
    model_name = "vgg16"
    #model = vgg(model_name=model_name, num_classes=7, init_weights=True)
    y=model(x)
    print(y.size())
```

train.py

```
#从自己创建的 models 库里导入 VGG 模块
#import VGG 仅仅是把 VGG.py 导入进来,当我们创建 VGG 的实例的时候需要通过指定 VGG.py 中的具体类.
#例如:我的 VGG.py 中的类名是 VGG,则后面的模型实例化 VGG 需要通过 **VGG.VGG()**来操作
#还可以通过 from 还可以通过 from VGG import * 直接把 VGG.py 中除了以 _ 开头的内容都导入
from models.cv.VGG import *
# 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([
# RandomResizedCrop(224)：将给定图像随机裁剪为不同的大小和宽高比，然后缩放所裁剪得到的图像为给定大小
    transforms.RandomResizedCrop(224),
# RandomVerticalFlip()：以 0.5 的概率竖直翻转给定的 PIL 图像
    transforms.RandomHorizontalFlip(),
    # ToTensor()：数据转化为 Tensor 格式
    transforms.ToTensor(),
# Normalize()：将图像三个通道的像素值归一化到[-1,1]之间，使模型更容易收敛
    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=32,shuffle=True)

test_dataloader=DataLoader(test_dataset,batch_size=32,shuffle=True)

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

model_name="vgg16"
#model=vgg(model_name=model_name,num_classes=7,init_weights=True).to(device)
model=vgg(model_name=model_name,num_classes=7,init_weights=True).to(device)
# 定义损失函数（交叉熵损失）
loss_fn=nn.CrossEntropyLoss()
# 定义 adam 优化器
# 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，默认：1e-3
# momentum(float, 可选)：动量因子（默认：0），矫正优化率
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# 迭代次数（训练次数）
epochs = 30
# 用于判断最佳模型
best_acc = 0.0
# 最佳模型保存地址
save_path = './{}Net.pth'.format(model_name)
train_steps = len(train_dataloader)

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)
            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，防止上个 batch 的数据的梯度值累积
            optimizer.zero_grad()
            cur_loss.backward()
            optimizer.step()
            loss+=cur_loss.item()
            acc+=cur_acc.item()
            n=n+1
    train_loss=loss/n
    train_acc=acc/n
    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(test_dataloader,model,loss_fn):
    loss,acc,n=0.0,0.0,0
    # 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(test_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]
            optimizer.zero_grad()
            cur_loss.backward()
            optimizer.step()
            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('test_acc' + str(test_acc))
        return test_loss, test_acc

def matplot_loss(train_loss, test_loss):
    fig, ax = plt.subplots(1, 1)
    # 参数 label = "传入字符串类型的值，也就是图例的名称
    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()

epochs=20
min_acc=0.0

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

for t in range(epochs):
    #不同的优化函数不同的使用方法
   # lr_scheduler.step()
```

```
        print(f"{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(folder)
            min_acc=test_acc
            print(f"保存{t+1}轮")
            #torch.save(model.state_dict(),path)只保存模型参数，推荐使用的方式
            torch.save(model.state_dict(),"save_model/vgg-best-model.pth")
        if t==epochs-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

[1]https://zh.d2l.ai/index.html
[2]Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.
[3]https://www.cnblogs.com/fusheng-rextimmy/p/15452248.html