第 5 章

若干经典 CNN 预训练模型及其迁移方法

顾名思义，预训练模型(Pre-training Model)是在训练数据充足的数据集上训练出来的性能优越的大模型，可以为下游任务提供支持。大规模的 CNN 预训练模型具有强大的特征抽取能力和表达能力，对解决复杂问题具有明显的优势。但是，训练大的预训练模型需要具备一定的条件，比如，需要带标注的大数据和大算力的支撑。对大规模数据的标注，其本身就是一件耗时的工程，而且大规模数据的获取也是一件不容易的事情；大算力的构建往往只有那些大的专业公司才能完成。因此，为了解决一个小问题，而去训练一个大模型是不现实的。但是，我们可以利用那些已经训练好了的且已经公开发布的模型(预训练模型)来解决我们面临的问题，这就涉及预训练模型的迁移和微调方法。通过迁移和微调，我们可以“站在巨人的肩膀上”去解决问题，从而达到事半功倍的效果。

5.1 一个使用 VGG16 的图像识别程序

在例 4.3 中，我们从“零”开始编写了一个识别猫、狗图像的程序。在本节中，以 VGG16 为基础，编写一个待学习参数非常少的图像识别程序，而且用的训练数据也很少，主要目的是让读者对已有预训练模型的使用和微调方法有一个初步的了解。

5.1.1 程序代码

在下面例子中，通过对预训练模型 VGG16 进行微调，构建一个能够识别猫狗图像的深度神经网络。该网络需要学习的参数比较少，使用的训练数据也很少，但效果更佳。

【例 5.1】 以 VGG16 作为预训练模型，通过微调，创建一个能够识别猫狗图像的深度神经网络。

本例的任务与例 4.3 的任务一样，都是识别猫和狗的图像。不同的是，本例使用了预训练模型——VGG16，这样使用的训练数据就相对少得多。在本例中，训练图像位于./data/catdog/training_set2 目录下，猫和狗的图像各 1000 张，共有 2000 张图像作为训练数据，它们都是从./data/catdog/training_set 目录中随机抽取，但测试集不变(与例 4.3 一样，位于./data/catdog/test_set 目录下，一共有 2023 张)。

本程序首先导入 VGG16，然后冻结参数并修改模型的部分结构，以适合本例的任务，最后进行训练和测试。程序的全部代码如下：

```
from torchvision import datasets, transforms, models
import torch
import torch.nn as nn
```

```
from torch.utils.data import DataLoader,Dataset
from PIL import Image
import os
import time
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#-------------------------------------
transform = transforms.Compose([
    transforms.Resize((224,224)),        #调整图像大小为(224,224)
    transforms.ToTensor(),               #转换为张量
])
class cat_dog_dataset(Dataset):
    def __init__(self, dir):
        self.dir = dir
        self.files = os.listdir(dir)
    def __len__(self):                   #需要重写该方法,返回数据集大小
        return len(self.files)
    def __getitem__(self, idx):
        file = self.files[idx]
        fn = os.path.join(self.dir, file)
        img = Image.open(fn).convert('RGB')
        img = transform(img)             #调整图像形状为(3,224,224),并转换为张量
        img = img.reshape(-1,224,224)
        y = 0 if 'cat' in file else 1    #构造图像的类别
        return img,y
#=============================================
batch_size = 20
train_dir = './data/catdog/training_set2'          #训练集所在的目录
test_dir ='./data/catdog/test_set'                 #测试集所在的目录
train_dataset = cat_dog_dataset(train_dir)         #创建数据集
train_loader = DataLoader(dataset=train_dataset,   #打包
                          batch_size=batch_size,
                          shuffle=True)
test_dataset = cat_dog_dataset(test_dir)
test_loader = DataLoader(dataset=test_dataset,
                          batch_size=batch_size,
                          shuffle=True)
print('训练集大小: ',len(train_loader.dataset))
print('测试集大小: ',len(test_loader.dataset))
#================================
cat_dog_vgg16 = models.vgg16(pretrained=True).to(device)
for i,param in enumerate(cat_dog_vgg16.parameters()):
    param.requires_grad = False      #冻结 cat_dog_vgg16 中已有的所有参数
cat_dog_vgg16.classifier[3] = nn.Linear(4096,1024)   #其参数默认是可学习的
cat_dog_vgg16.classifier[6] = nn.Linear(1024,2)      #其参数默认是可学习的
cat_dog_vgg16.train()
cat_dog_vgg16 = cat_dog_vgg16.to(device)
optimizer = torch.optim.SGD(cat_dog_vgg16.parameters(), lr=0.01, momentum=0.9)
start=time.time()                                    #开始计时
cat_dog_vgg16.train()
for epoch in range(10):                              #执行 10 代
```

```
    ep_loss=0
    for i,(x,y) in enumerate(train_loader):
        x, y = x.to(device),y.to(device)
        pre_y = cat_dog_vgg16(x)
        loss = nn.CrossEntropyLoss()(pre_y, y.long())  #使用交叉熵损失函数
        ep_loss += loss * x.size(0)    #loss 是损失函数的平均值,故要乘以样本数量
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print('第 %d 轮循环中,损失函数的平均值为: %.4f'\
          %(epoch+1,(ep_loss/len(train_loader.dataset))))
end = time.time()                                          #计时结束
print('训练时间为:  %.1f 秒 '%(end-start))
#=============================================
correct = 0
cat_dog_vgg16.eval()
with torch.no_grad():
    for i, (x, y) in enumerate(train_loader):              #计算在训练集上的准确率
        x, y = x.to(device), y.to(device)
        pre_y = cat_dog_vgg16(x)
        pre_y = torch.argmax(pre_y, dim=1)
        t = (pre_y == y).long().sum()
        correct += t
t = 1. * correct/len(train_loader.dataset)
print('1. 网络模型在训练集上的准确率: {:.2f}%'\
      .format(100 * t.item()))
correct = 0
with torch.no_grad():
    for i, (x, y) in enumerate(test_loader):               #计算在测试集上的准确率
        x, y = x.to(device), y.to(device)
        pre_y = cat_dog_vgg16(x)
        pre_y = torch.argmax(pre_y, dim=1)
        t = (pre_y == y).long().sum()
        correct += t
t = 1. * correct/len(test_loader.dataset)
print('2. 网络模型在测试集上的准确率: {:.2f}%'\
      .format(100 * t.item()))
```

执行上述代码,输出结果(部分)如下:

```
… …
第 9 轮循环中,损失函数的平均值为: 0.0460
第 10 轮循环中,损失函数的平均值为: 0.0553
训练时间为:  86.4 秒
1. 网络模型在训练集上的准确率: 99.70%
2. 网络模型在测试集上的准确率: 96.69%
```

可见,与例 4.3 相比,该程序的训练数据少了,运行的代数也少了,但准确率却大幅上升了。显然,这得益于预训练模型 VGG16 的功劳,是站在 VGG16 这个“巨人肩膀”上的结果。

5.1.2 代码解释

本例主要是导入了一个预训练模型——VGG16，创建实例 cat_dog_vgg16，以代替在例 4.3 中创建的实例 model_CatDog，其他部分代码基本相同。相关代码说明如下：

(1) 通过下面语句从模型库 models 中导入已经训练好的模型 VGG16。

```
cat_dog_vgg16 = models.vgg16(pretrained=True)
```

其中，pretrained＝True 表示要下载已训练好的所有参数。如果 pretrained＝False，则表示不下载这些参数，而使用随机方法初始化所有参数。这相当于只使用模型 VGG16 的结构，而不要其训练过的参数。显然，一般情况下 pretrained＝True。

如果想导入 VGGNet 的另一个家族成员——VGG19，则用下列语句即可。

```
cat_dog_vgg19 = models.vgg19(pretrained=True)
```

注意，此处的 cat_dog_vgg16 就是相当于例 4.3 中的 model_CatDog，都是已经创建好的实例。因此，在本例中可以不再创建一个类。

(2) 使用下列语句冻结刚创建的模型 cat_dog_vgg16 的参数。

```
for i,param in enumerate(cat_dog_vgg16.parameters()):
    param.requires_grad = False       #冻结 cat_dog_vgg16 的所有参数
```

如果一个参数的 requires_grad 属性值设置为 False，则该参数在训练过程中是不能被更新的，因而称为“冻结”。由于 VGG16 中的参数都是训练过的，且已被实践证明是可行的，因而就不需要再训练了，而且 VGG16 中的参数量巨大，一般也没有条件来训练它们。

用下列代码，可以查看模型中各层参数张量是否可以被训练。

```
for layer in cat_dog_vgg16.named_modules():
    t = list(layer[1].parameters())
    if len(t) == 0:                  #如果当前层没有训练参数，则 len(t) = 0
        continue
    L = []
    for param in layer[1].parameters():
        L.append(param.requires_grad)
    print(layer[0], ' ------------> ',L)
                                     #True 表示相应参数张量可训练，False 表示不可以
```

(3) 对模型 cat_dog_vgg16 进行微调，改为适合本例识别任务的网络结构。先用下列语句打印出 cat_dog_vgg16 的层次结构：

```
print(cat_dog_vgg16)
```

结果如图 5-1 所示。从图 5-1 中可以看出，该网络有 1000 个输出，而本程序只需要两个输出，因而至少需要更改最后一层网络的输出结构。作为例子，本例修改最后面的两个全连接层，即修改下面这两层：

(3)：Linear(in_features＝4096，out_features＝4096，bias＝True)

(6)：Linear(in_features＝4096，out_features＝1000，bias＝True)

修改后的 VGG16 结构如下：

```
VGG(
(features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace=True)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace=True)
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace=True)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace=True)
        (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace=True)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace=True)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace=True)
        (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace=True)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace=True)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace=True)
        (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace=True)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace=True)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace=True)
        (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
(classifier): Sequential(
        (0): Linear(in_features=25088, out_features=4096, bias=True)
        (1): ReLU(inplace=True)
        (2): Dropout(p=0.5, inplace=False)
        (3): Linear(in_features=4096, out_features=4096, bias=True)
        (4): ReLU(inplace=True)
        (5): Dropout(p=0.5, inplace=False)
        (6): Linear(in_features=4096, out_features=1000, bias=True)
        )
)
```

图 5-1　VGG16 结构的层次图

使用的修改代码如下：

```
cat_dog_vgg16.classifier[3] = nn.Linear(4096,1024)      #其参数默认是可学习的
cat_dog_vgg16.classifier[6] = nn.Linear(1024,2)         #其参数默认是可学习的
```

注意，只有这两层发生改变，且其参数也是默认可学习的（即这两层参数的 requires_grad 属性值默认为 True），而其他网络层都保持不变，它们的参数已被冻结。

（4）在加载数据时，以./data/catdog/training_set2 目录下的图像文件作为训练数据，训练的代数改为 10 代。

除了上述改变外，数据加载方法、模型训练方法和测试方法等其他部分与例 4.3 的相同。

5.2 经典卷积神经网络的结构

上一节已经见证了预训练模型 VGG16 的魅力。本节将介绍包括 VGG16 在内的若干经典预训练模型的结构，一方面可以为今后模型结构设计提供参考，另一方面也可以为更好地通过微调方法利用这些模型作准备。

5.2.1 卷积神经网络的发展过程

神经网络的出现可以追溯到 1943 年。当年，心理学家 Warren McCulloch 和数理逻辑学家 Walter Pitts 首先提出了人工神经网络的概念，并给出了人工神经元的数学模型，从此开启了人工神经网络研究的时代。1957 年，美国神经学家 Frank Rosenblatt 成功地在 IBM 704 机上完成了感知器的仿真，并于 1960 年实现了手写英文字母的识别。1974 年，Paul Werbos 在其博士论文中首次提出后向传播（Back propagation，BP）思想来修正网络参数的方法，这是 BP 算法的雏形。但在当时由于人工智能正处于发展的低谷，这项工作并没有引起足够的重视。1986 年，在 Meclelland 和 Rumelhart 等的努力下，BP 算法被进一步发展，并逐步引起广泛关注，被大量应用于神经网络训练任务当中。BP 算法的主要贡献在于，提出一种基于梯度信息的参数修正算法，为神经网络的训练提供了一种非常成功的参数训练方法。目前，正在盛行的深度学习中各种网络模型也均采用 1986 年提出的 BP 算法。

最早的卷积神经网络是由 LannYeCun 等于 1998 年提出来的，这就是 LeNet。LeNet 主要用于识别手写数字图像，由两个卷积层和两个池化层组成，结构比较简单，但它是最早达到实用水平的神经网络。如今，真正掀起深度学习风暴的是 LeNet 的加宽版——AlexNet。AlexNet 是于 2012 年由 Hinton 的学生 Krizhevsky Alex 提出来的，并在当年的 ImageNet 视觉挑战赛（ImageNet Large Scale Visual Recognition Challenge，ILSVRC）上以巨大的优势获得冠军。相比于以往战绩，AlexNet 大幅降低了图像识别错误率，它的出现标志着深度学习时代的来临。

2014 年，GoogLeNet 和 VGG 同时诞生。GoogLeNet 是当年的 ILSVRC 冠军，通过设计和开发 Inception 模块，使得模型的参数大幅减少。VGG 则继续加深网络，通过扩展网络的深度来获取性能的提升。

2015 年，残差神经网络 ResNet 诞生，并在当年获得 ILSVRC 冠军。ResNet 旨在解决网络因深度增加而出现性能退化的问题，它提供了一种构造大深度卷积网络的技术和方法。

2019 年，谷歌公司开发了一种以效率著称的深度神经网络——EfficientNet。EfficientNet 仍然是至今为止最好的图像识别网络之一。

5.2.2　AlexNet 网络

在结构上，AlexNet 要比 LeNe 复杂得多，它由 5 个卷积层、3 个最大池化层、2 个归一化层和 3 个全连接层组成。

在第一层（卷积层 1）中，输入图像的尺寸为 227×227×3，采用 11×11 卷积核，设置的输出通道数为 96、步长为 4，因而在该层输出时，特征图的大小为(227－11)/4＋1＝55，输出特征图的形状为(55×55×96)。

在第二层（池化层 1）中，输入的特征图就是上一层的输出，其尺寸为 227×227×3，该层采用 3×3 池化核，步长为 2，因而输出特征图的尺寸为(55－3)/2＋1＝27，从而该层输出特征图的形状为 27×27×96(池化层不改变通道数)。

其他层输出的特征图的形状变化可以依此类推，具体操作和输出特征图的形状变化如表 5-1 所示。

表 5-1　AlexNet 网络的层次结构

网络层	输入形状	操作(等效操作)	输出形状	特征图大小计算	当前层中的参数量
卷积层 1	227×227×3	11×11 卷积核，输出通道数为 96，步长为 4	55×55×96	(227－11)/4＋1＝55	96×3×11×11＋96＝34944
池化层 1	55×55×96	3×3 池化核，步长为 2	27×27×96	(55－3)/2＋1＝27	0
归一化层					0
卷积层 2	27×27×96	5×5 卷积核，输出通道数为 256，步长为 1，填充为 2	27×27×256	(27－5＋2×2)/1＋1＝27	256×96×3×3＋256＝221440
池化层 2	27×27×256	3×3 池化核，步长为 2	13×13×256	(27－3)/2＋1＝27	0
归一化层					0
卷积层 3	13×13×256	3×3 卷积核，输出通道数为 384，步长为 1，填充为 1	13×13×384	(13－3＋2×1)/1＋1＝13	384×256×3×3＋384＝885120
卷积层 4	13×13×384	3×3 卷积核，输出通道数为 384，步长为 1，填充为 1	13×13×384	(13－3＋2×1)/1＋1＝13	384×384×3×3＋384＝1327488
卷积层 5	13×13×384	3×3 卷积核，输出通道数为 256，步长为 1，填充为 1	13×13×256	(13－3＋2×1)/1＋1＝13	256×384×3×3＋256＝884992
池化层 3	13×13×256	3×3 池化核，步长为 2	6×6×256	(13－3)/2＋1＝6	0

续表

网络层	输入形状	操作(等效操作)	输出形状	特征图大小计算	当前层中的参数量
扁平化	6×6×256	将特征图向量化	9216		0
全连接层 1	9216	全连接	4096		9216×4096+4096=37752832
全连接层 2	4096	全连接	4096		4096×4096+4096=16781312
全连接层 3	4096	全连接	1000		4096×1000+1000=4097000

按照第 2 章介绍的方法,我们可以计算 AlexNet 的参数总量为 61 975 936,即 AlexNet 有六千多万个参数需要优化。

5.2.3 VGGNet 网络

VGGNet 是牛津大学 Simonyan 等提出的一种深度神经网络结构,其中比较常用的结构是 VGG16,其次是 VGG19。作为一个例子,下面主要介绍 VGG16 网络的层次结构和特点。

VGG16 有 13 个卷积层和 3 个全连接层,这些都是带有待优化参数的网络层,共 16 个网络,因而称为 VGG16。VGG16 网络的层次结构如表 5-2 所示。

表 5-2 VGG16 网络的层次结构

网络层	输入形状	操作(等效操作)	输出形状	特征图大小计算	当前层中的参数量
卷积层 1	(3,224,224)	3×3 卷积核,输出通道数为 64	(64,224,224)	224−3+2×1+1=224	64×3×3×3+64=1792
卷积层 2	(64,224,224)	3×3 卷积核,输出通道数为 64	(64,224,224)	同上	64×64×3×3+64=36928
池化层 1	(64,224,224)	2×2 池化核,步长为 2	(64,112,112)	224/2=112	0
卷积层 3	(64,112,112)	3×3 卷积核,输出通道数为 128	(128,112,112)	112−3+2×1+1=112	128×64×3×3+128=73856
卷积层 4	(128,112,112)	3×3 卷积核,输出通道数为 128	(128,112,112)	同上	128×128×3×3+128=147584
池化层 2	(128,112,112)	2×2 池化核,步长为 2	(128,56,56)	112/2=56	0
卷积层 5	(128,56,56)	3×3 卷积核,输出通道数为 256	(256,56,56)	56−3+2×1+1=56	256×128×3×3+256=295168
卷积层 6	(256,56,56)	3×3 卷积核,输出通道数为 256	(256,56,56)	同上	256×256×3×3+256=590080
卷积层 7	(256,56,56)	3×3 卷积核,输出通道数为 256	(256,56,56)	同上	256×256×3×3+256=590080
池化层 3	(256,56,56)	2×2 池化核,步长为 2	(256,28,28)	56/2=28	0

续表

网络层	输入形状	操作(等效操作)	输出形状	特征图大小计算	当前层中的参数量
卷积层 8	(256,28,28)	3×3 卷积核,输出通道数为 512	(512,28,28)	28−3+2×1+1=28	512×256×3×3+512=1180160
卷积层 9	(512,28,28)	3×3 卷积核,输出通道数为 512	(512,28,28)	同上	512×512×3×3+512=2359808
卷积层 10	(512,28,28)	3×3 卷积核,输出通道数为 512	(512,28,28)	同上	512×512×3×3+512=2359808
池化层 4	(512,28,28)	2×2 池化核,步长为 2	(512,14,14)	28/2=14	0
卷积层 11	(512,14,14)	3×3 卷积核,输出通道数为 512	(512,14,14)	14−3+2×1+1=14	512×512×3×3+512=2359808
卷积层 12	(512,14,14)	3×3 卷积核,输出通道数为 512	(512,14,14)	同上	512×512×3×3+512=2359808
卷积层 13	(512,14,14)	3×3 卷积核,输出通道数为 512	(512,14,14)	同上	512×512×3×3+512=2359808
池化层 5	(512,14,14)	2×2 池化核,步长为 2	(512,7,7)	14/2=7	0
全连接层 1	7×7×512=25088	全连接	4096		25088×4096+4096=102764544
全连接层 2	4096	全连接	4096		4096×4096+4096=16781312
全连接层 3	4096	全连接	1000		4096×1000+1000=4097000
Softmax 层	1000	计算概率分布	1000		0

从表 5-2 中可以看出,VGG16 全部采用 3×3 卷积核(步长均为 1)和 2×2 池化核(步长均为 2),在卷积时均填充数为 1(即填充 1 个 0 圈)。AlexNet 采用大的卷积核,以扩大其感受野,因此层次不需要很高。与 AlexNet 相比,VGG16 采用小卷积核和小池化核,各层的参数不多,但堆叠了 13 层 3×3 卷积核。底层卷积核的感受野确实不大,但高层的感受野同样很大,而且层与层之间的非线性映射可以提高对底层特征学习的抽象能力。总体而言,AlexNet 显得“矮胖”,宽度大;VGG16 则比较“高瘦”,深度大,VGG16 参数总量为 138 357 544,是 AlexNet 两倍多,其性能当然也比 AlexNet 好得多。

注意,在图 5-1 所示的 VGG16 的结构中,第 37 行所示的网络层是第一个全连接层。该层要求输入张量的最后一维的大小必须为 25 088。然而,VGG16 可以接收不同尺寸图像的输入,从而卷积网络部分会产生不同尺寸的特征图(第 33 行所表示的网络层的输出)。那么,VGG16 是如何把不同尺寸的特征图都转换为最后一维的大小为 25 088 的张量呢?这主要依赖于第 33 行所示的自适应平均池化层。该层对应的代码如下:

```
nn.AdaptiveAvgPool2d(output_size=(7, 7))
```

其作用是,对输入该层的特征图,不管图像尺寸为多少,其输出特征图的尺寸永远为

7×7(批量大小和通道数不变,通道数为 512)。这样,经过扁平化后得到输入全连接网络层的维度大小为 7×7×512=25 088。也就是说,自适应平均池化层保证了 VGG16 可以接收不同尺寸图像的输入,而不需改变网络的结构。读者也可以在自己构建的模型中使用自适应平均池化层,以使得网络可以接收不同尺寸图像的输入。

5.2.4 GoogLeNet 网络与 1×1 卷积核

一般来说,如果一个网络越宽(卷积核数量增加)、越深(深度增加),那么它的参数就越多,就能解决越复杂的问题。但带来的问题也是明显的:一是在训练数据有限的情况下,容易造成过拟合,无法真正解决问题;二是极大地增加计算量,需要更强的算力支撑。自然地,在保持同样宽度和高度的情况下,如何尽可能地减少网络参数的个数呢?而这就是 GoogLeNet 要解决的主要问题。为此,GoogLeNet 使用了许多关键技术,其中很重要的技术就是设计了 1×1 卷积核。下面先看看 1×1 卷积核的作用。

从 nn.Conv2d()函数看,1×1 卷积核对应的函数如下:

```
nn.Conv2d(in_channels, out_channels, (1, 1))
```

其中,默认步长为 1,无填充。

假设输入特征图的高和宽分别为 H 和 W,则在此卷积核作用下输出特征图的高和宽分别为 $H-1+1=H$ 和 $W-1+1=W$。也就是说,在 1×1 卷积核作用下,卷积后特征图的高和宽均保持不变。但根据卷积的定义,特征图中各个元素是各通道上对应元素的加权和,因此输出特征图对输入特征图进行了一种线性变换。重要的是,虽然 in_channels 是由输入特征图确定的,但 out_channels 可以根据需要自由设置。如果设置结果是 out_channels<in_channels,那么这种设置是对输入特征图的压缩,可以理解为降维;如果 out_channels>in_channels,那么这种设置是对输入特征图的扩张,可以理解为升维。也就是说,1×1 卷积可以起到升维和降维的作用,同时也对输入特征图进行了一种线性加权变换。简单地理解,1×1 卷积是通过线性变换改变特征图的通道数,从而起到升维和降维的作用。

GoogLeNet 这个名字可以理解为 Google+LeNet,意指是谷歌公司在 LeNet 的基础上发展出来的。GoogLeNet 有两个特点,一个是 GoogLeNet 由 9 个称为 Inception 的模块构成,另一个是有 3 个 softmax 输出层。下面先介绍第一个特点。

Inception 模块经过了几个版本演进,分别是原始版本、v1、v2 和 v3。Inception 原始版本和 Inception v1 的结构分别如图 5-2(a)和图 5-2(b)所示。原始版本是由并列的 3 个卷积层和 1 个池化层构成,分别是:1×1 卷积层、3×3 卷积层、5×5 卷积层和 3×3 最大池化层。每个卷积层设置有多个卷积核,卷积核个数即为该卷积层的输出通道数,池化层的通道数保持不变。在 Inception 模块中,所有这些通道被叠加在一起作为该 Inception 模块的输出通道。之所以用不同大小的卷积核,是因为这些不同尺寸的卷积核可以提取不同粒度的特征,实现多尺度特征提取,目的是充分利用不同粒度的特征,这是 GoogLeNet 的创新之一。

我们注意到,GoogLeNet 并没有使用原始版本的 Inception 模块,而是使用了如图 5-2(b)所示的带降维的 Inception v1。实际上,后者是前者的改进版本,改进的结果体现在减少了神经网络的参数量,从而提高运行效率。与原始版本相比,带降维的 Inception v1 主要导入