



第5章 若干经典 CNN 预训练模型及其迁移方法

授课人：李仪

liyi1002@csu.edu.cn

<http://faculty.csu.edu.cn/liyi>

中南大学 自动化学院



主要内容

01

一个使用 VGG16 的图像识别程序

02

经典卷积神经网络的结构

03

预训练模型的迁移方法



01

一个使用 VGG16 的图像识别程序



5.1.1 程序代码

【例5.1】 创建一个能够识别猫狗图像的深度神经网络。

- 任务跟例 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。
- 训练图像位于./data/catdog/training_set2 目录下，猫和狗的图像各 1000 张，一共有 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（跟例 4.3 一样，位于./data/catdog/test_set 目录下，一共有 2023 张）。

5.1.1 程序代码

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。程序的核心代码如下：

```
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))
```

该程序的核心代码（全部代码见教材P120）

5.1.1 程序代码

执行上述代码后，结果如下（部分）：

... ..

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%

与例 4.3 相比，该程序的训练数据少了，运行的代数少了，但准确率却大幅度上升。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，一般情况下使用 `pretrained=True`。

5.1.2 代码解释

如果向导入 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了。

5.1.2 代码解释

(2) 使用下列语句冻结刚创建的模型 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，已被实践证明是可行的，因而在后面的训练过程中就不需要再训练了，而且 VGG16 中的参数量巨大，一般也没有条件来训练它们。

5.1.2 代码解释

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
```

5.1.2 代码解释

(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)
```

结果如图所示（下页）。从图中可以看出，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)
```

5.1.2 代码解释

修改后VGG结构如右图所示:

```
1. VGG(  
2.   (features): Sequential(  
3.     (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
4.     (1): ReLU(inplace=True)  
5.     (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
6.     (3): ReLU(inplace=True)  
7.     (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
8.     (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
9.     (6): ReLU(inplace=True)  
10.    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
11.    (8): ReLU(inplace=True)  
12.    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
13.    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
14.    (11): ReLU(inplace=True)  
15.    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
16.    (13): ReLU(inplace=True)  
17.    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
18.    (15): ReLU(inplace=True)  
19.    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
20.    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
21.    (18): ReLU(inplace=True)  
22.    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
23.    (20): ReLU(inplace=True)  
24.    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
25.    (22): ReLU(inplace=True)  
26.    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
27.    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
28.    (25): ReLU(inplace=True)  
29.    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
30.    (27): ReLU(inplace=True)  
31.    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
32.    (29): ReLU(inplace=True)  
33.    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
34.  )  
35.  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))  
36.  (classifier): Sequential(  
37.    (0): Linear(in_features=25088, out_features=4096, bias=True)  
38.    (1): ReLU(inplace=True)  
39.    (2): Dropout(p=0.5, inplace=False)  
40.    (3): Linear(in_features=4096, out_features=4096, bias=True)  
41.    (4): ReLU(inplace=True)  
42.    (5): Dropout(p=0.5, inplace=False)  
43.    (6): Linear(in_features=4096, out_features=1000, bias=True)  
44.  )  
45. )
```

VGG16 结构的层次图

5.1.2 代码解释

(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跟例 4.3 的相同。



02

经典卷积神经网络的结构





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，从此掀开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 IBM704 机上完成了感知机的仿真，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 Mcclelland 和 Rumelhart 等人的努力下，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。

5.2.1 卷积神经网络的发展过程

- 最早的卷积神经网络是由 LannYeCun 等人于 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。
- 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，并在当年 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 大幅度地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。

5.2.1 卷积神经网络的发展过程

- 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度减少。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，Google 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 $227 \times 227 \times 3$ ，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 $(227-11)/4+1=55$ ，因此输出特征图的形状为 $(55 \times 55 \times 96)$ 。
- 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 $227 \times 227 \times 3$ ，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 $27 \times 27 \times 96$ （池化层不改变通道数）。

5.2.2 AlexNet 网络

- 其他层输出的特征图的形状变化可以依次类推，具体操作和输出特征图的形状变化见右表：

网络层	输入形状	操作(等效操作)	输出形状	特征图大小 计算依据	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	$(227-11)/4+1=55$	$96 \times 3 \times 11 \times 11 + 96 = 34944$
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	$(27-5+2 \times 2)/1+1=27$	$256 \times 96 \times 3 \times 3 + 256 = 221440$
池化层 2	27×27×256	3×3 池化核, 步长为 2	13×13×256	$(27-3)/2+1=13$	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	$(13-3+2 \times 1)/1+1=11$	$384 \times 256 \times 3 \times 3 + 384 = 885120$
卷积层 4	13×13×384	出通道数为 384, 步长为 1, 填充为 1	13×13×384	$(13-3+2 \times 1)/1+1=11$	$384 \times 384 \times 3 \times 3 + 384 = 1327488$
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	$(13-3+2 \times 1)/1+1=11$	$256 \times 384 \times 3 \times 3 + 256 = 884992$
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
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		$9216 \times 4096 + 4096 = 37752832$
全连接层 2	4096	全连接	4096		$4096 \times 4096 + 4096 = 16781312$
全连接层 3	4096	全连接	1000		$4096 \times 1000 + 1000 = 4097000$

5.2 经典卷积神经网络的结构

5.2.3 VGGNET 网络

VGGNet 是牛津大学 Simonyan 等人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。

VGG16 有十余个网络层，其中有 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，一共 16 个网络，因而称为 VGG16。VGG16 网络的层次结构见教材P126：[表 5-2 VGG16 网络的层次结构](#)。

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。



5.2 经典卷积神经网络的结构

5.2.3 VGGNET 网络

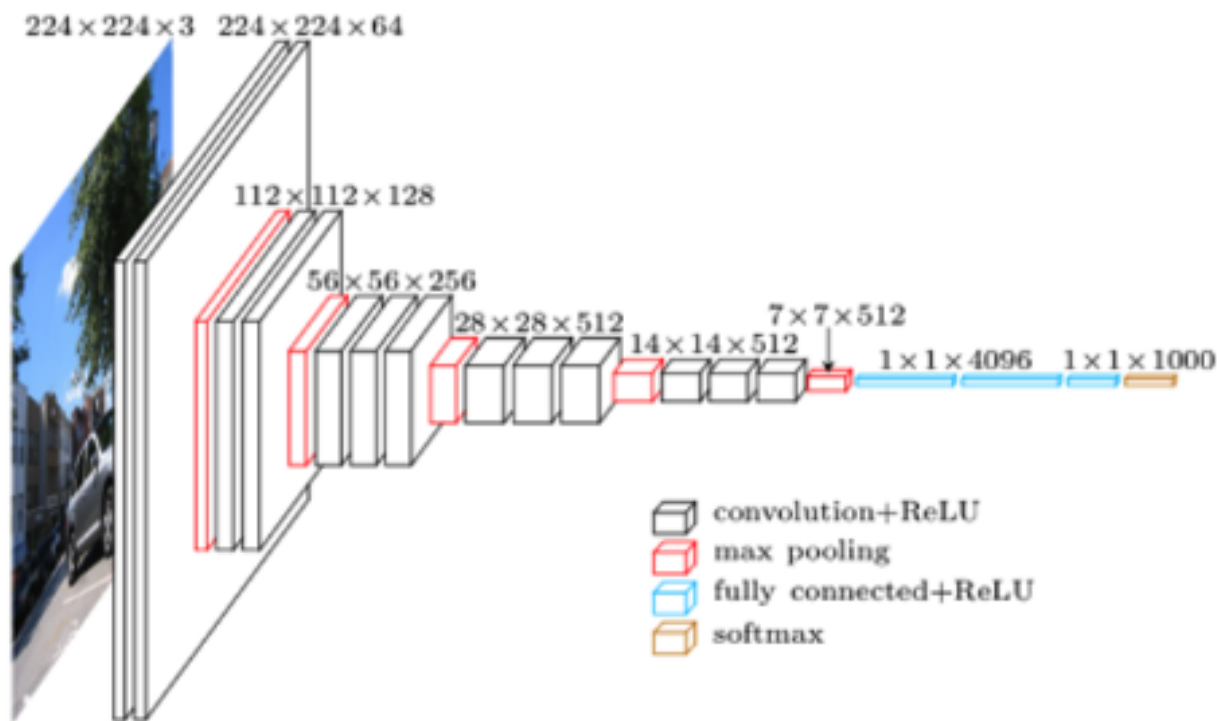
VGG16 是如何把不同尺寸的特征图都转化为最后一维的大小为 25088 的张量呢？这主要依赖于第 33 行所示的自适应平均池化层。该层对应的代码如下：

```
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 $7 \times 7 \times 512 = 25088$ 。也就是说，自适应平均池化层保证了 VGG16 可以接受不同尺寸图像的输入，而不需改变网络的结构。

5.2.3 VGGNET 网络

- conv的stride为1, padding为1
- maxpool的大小为2, stride为2



$$out_{size} = (in_{size} - F_{size} + 2P) / S + 1$$

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					



5.2 经典卷积神经网络的结构

5.2.4 GoogLeNet 网络与 1×1 卷积核

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： $w-1+1=w$

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

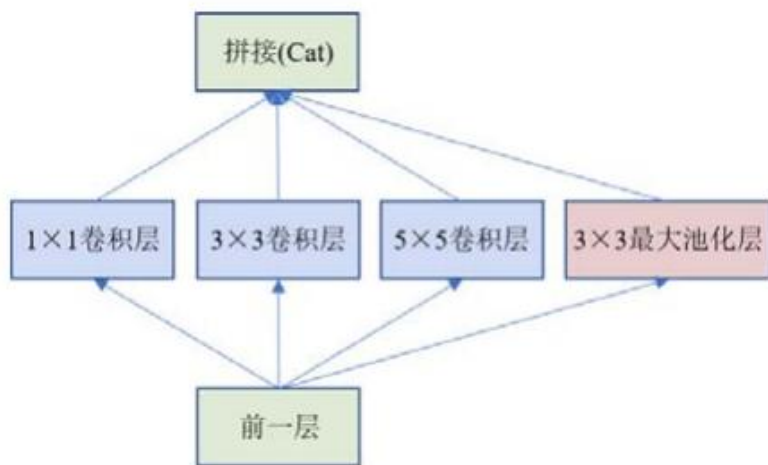
其中，默认步长为 1，无填充

GoogLeNet 这个名字可以理解为 Google+LeNet，意指是 Google 公司在 LeNet 的基础上发展出来的。GoogLeNet 有两个特点，一个是 GoogLeNet 有九个称为 Inception 的模块构成，另一个是有三个 softmax 输出层。

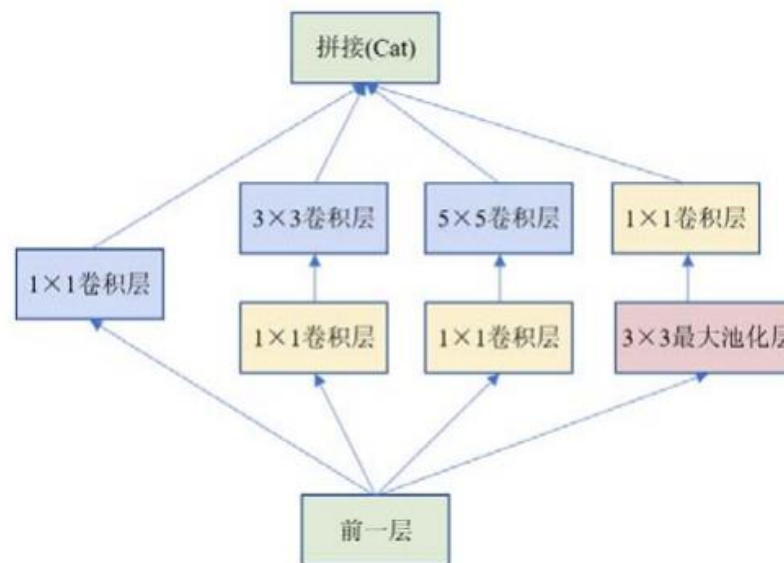
5.2 经典卷积神经网络的结构

5.2.4 GoogLeNet 网络与 1×1 卷积核

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 下图(a)和(b)所示。



(a) Inception 的初始版本



(b) Inception v1 (改进版本)

5.2 经典卷积神经网络的结构

5.2.5 ResNet 网络

- 残差网络的思想源于《Deep Residual Learning for Image Recognition》
- 残差：数理统计中是指实际观察值与估计值（拟合值）之间的差。“残差”蕴含了有关模型基本假设的重要信息。如果回归模型正确的话，我们可以将残差看作误差的观测值。它应符合模型的假设条件，且具有误差的一些性质。
- 残差与误差的区别：
 - The error term is the difference between the observed value for the dependent variable and its theoretical value, while a model is applied on overall population. We don't actually calculate it.
 - Residual is the practically calculated term during modeling exercise; It is the difference between the actual value in the sample and predicated value in the sample.
 - 误差是衡量观测值和真实值之间的差距，残差是指观测值和预测值之间的差距。

5.2 经典卷积神经网络的结构

5.2.5 ResNet 网络

- 在VGG中——卷积网络达到了19层，GoogLeNet——达到了22层。
- 问：网络的精度会随着网络的层数增多而增多吗？
- 答：随着层数增加一般会出现**退化（degradation）现象**——训练集loss逐渐下降，然后趋于饱和，当再增加网络深度时，训练集loss反而会增大。
- 设想：因此当网络退化时，浅层网络能够达到比深层网络更好的训练效果，这时如果把低层的特征传到高层，那么效果应该至少不比浅层的网络效果差。

5.2 经典卷积神经网络的结构

5.2.5 ResNet 网络

残差网络：由多个残差模块组成，右图是一个残差模块的结构。

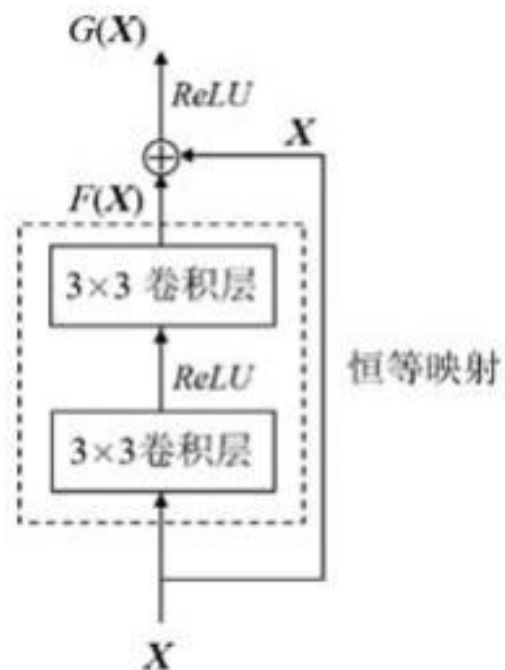
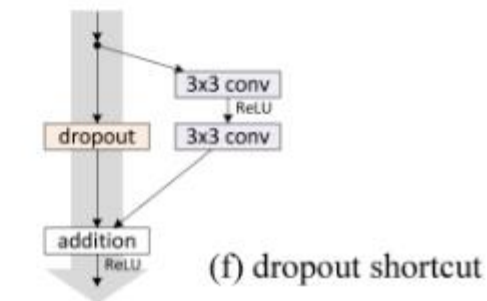
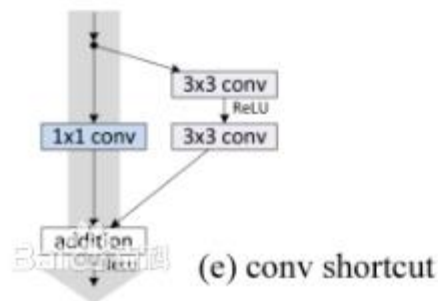
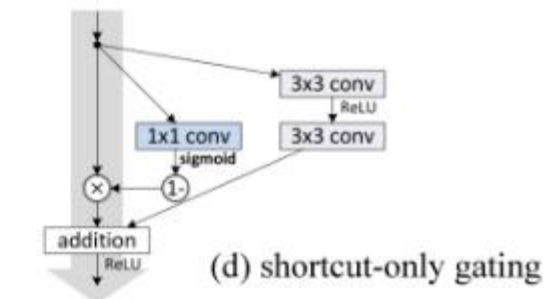
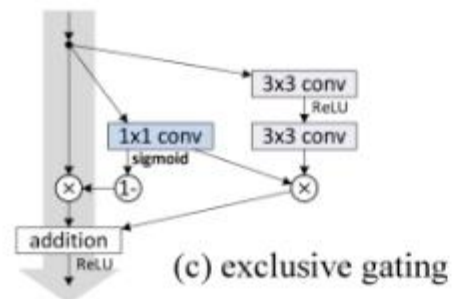
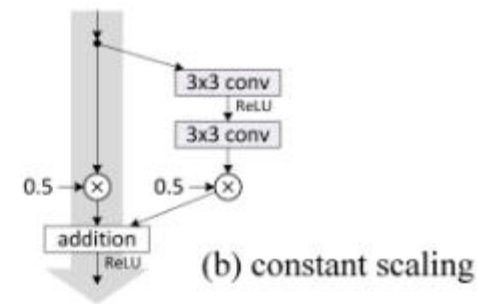
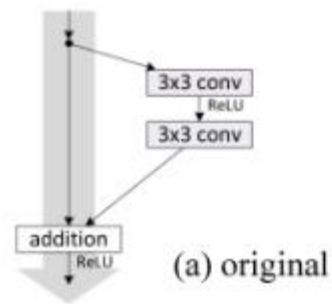


图 5-3 一个残差模块





5.2 经典卷积神经网络的结构

5.2.5 ResNet 网络

令 $G(X)$ 表示残差模块的输出，则有：

$$G(X) = F(X) + X$$

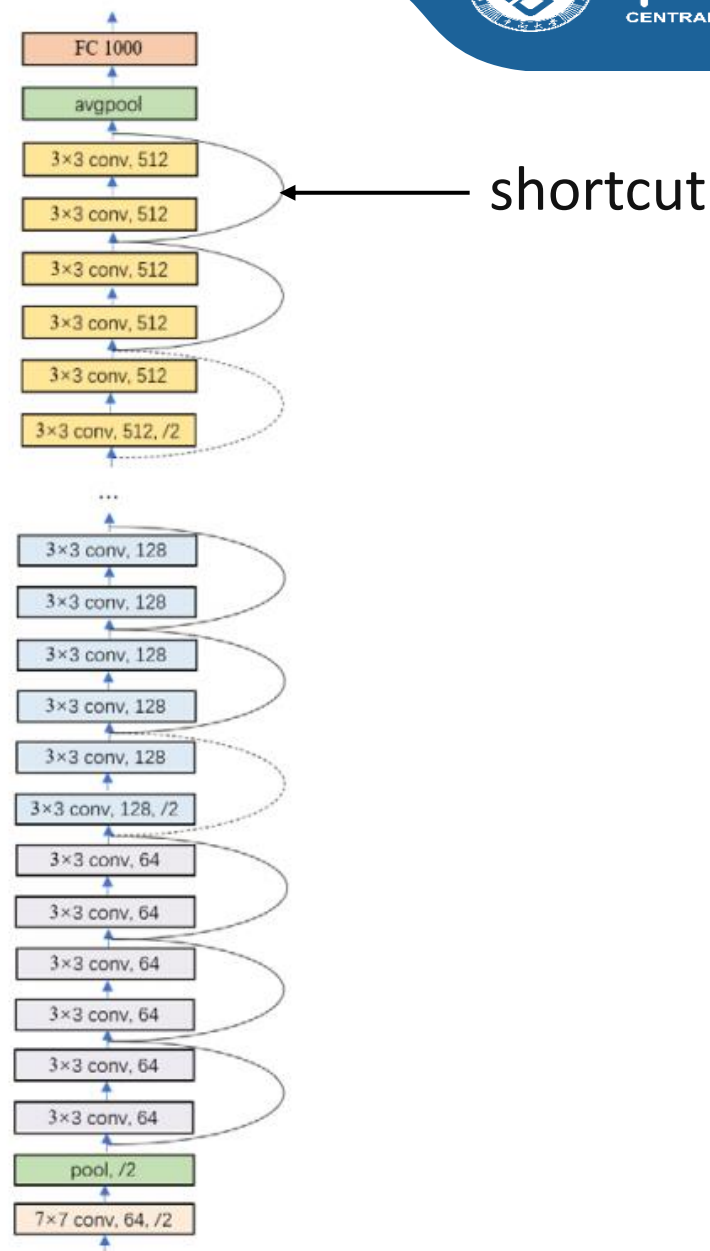
由上式也可以得到， $F(X) = G(X) - X$ 。也就是说，可以把两个卷积层的输出 $F(X)$ 看成是 $G(X)$ 和 X 之间的误差估计，其中 X 是输入的数据张量，没有学习参数， $F(X)$ 看成是两个卷积层构成的子网，有学习参数。通过对 $F(X)$ 中参数的学习，使得在使用误差 $F(X)$ 修正 X 后，修正的结果 $F(X) + X$ 更接近 $G(X)$ 。

5.2 经典卷积神经网络的结构

5.2.5 ResNet 网络

由多个残差模块堆叠而形成的残差网络（ResNet）的架构可用右图表示。

- 一个方框表示一个网络层；
- 一条弧线表示一个恒等映射；
- 虚线画的弧线表示需要对特征图做一些变换，如调整特征图的尺寸和通道数等，以保证 $F(X)$ 和 X 能够按位相加；
- 一条弧线及其跨越的两个表示卷积层的方框，共同构成了一个残差模块。



5.2 经典卷积神经网络的结构

5.2.5 ResNet 网络

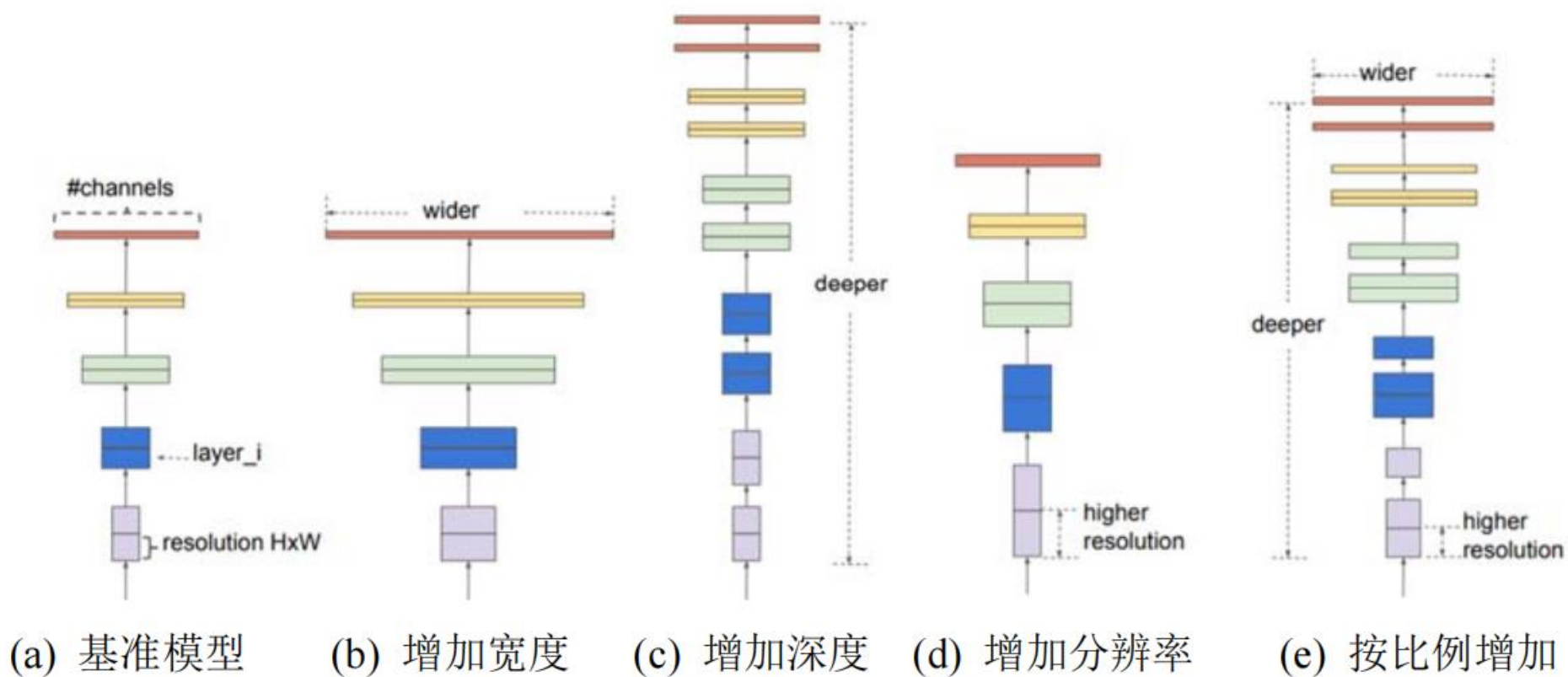
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

5.2.5 EfficientNet 网络

提高神经网络对图像的处理能力，通过三种途径来实现：

1. 增加网络的**宽度**：实际上就是增加每个网络层上卷积核的个数，以提取更多的特征，其原理如图 5-5(b)所示；
2. 增加网络的**深度**：即增加网络的层数，提高卷积核的感受野及其提取抽象特征的能力，如 VGG、ResNet 等，如图 5-5(c)所示；
3. 增加图像的**分辨率**：提高图像本身蕴含的信息量，如图 5-5(d)所示。

5.2.5 EfficientNet 网络





03

预训练模型的迁移方法



5.3.1 预训练网络迁移的基本原理

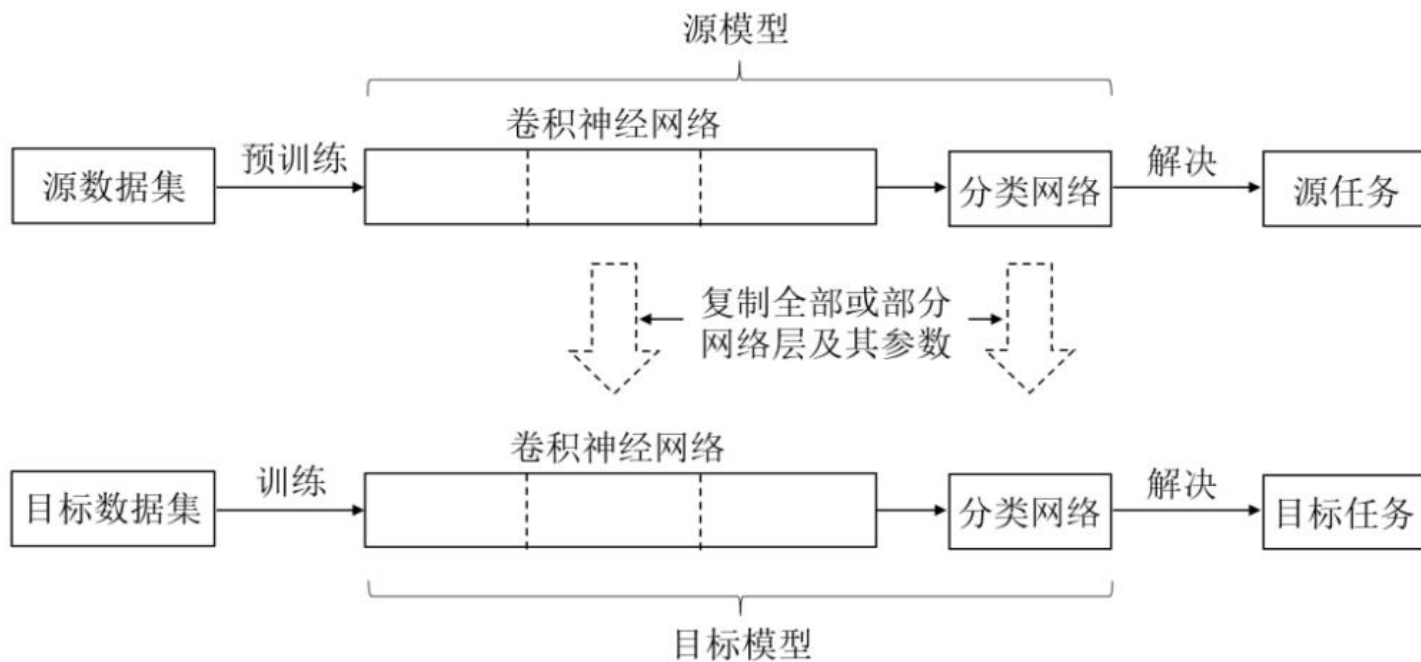
迁移学习 (Transfer learning) 引入:

假设在一个样本足够多的数据集（源数据集）上训练（预训练）出一个好的模型（源模型，对下游任务而言称为预训练模型），能够完美地解决给定的任务（源任务）。此后，将源模型中全部或部分网络层及其参数迁移过来，并在适当添加新网络层的基础上重新构造一个新模型（目标模型），然后在一个数据量较少的数据集（目标数据集）上进行训练，以用于解决新的任务（目标任务）。在这个训练过程中，源模型中迁移过来的参数一般不参与训练（被冻结了），而只是训练因新增加网络层而产生的少量参数。

5.3.1 预训练网络迁移的基本原理

在迁移方法当中，**微调**

(fine tuning) 是常采用的一种迁移方法。微调一般是指通过调整分类网络最后一个输出层的来构建新网络的方法（而其他网络层全部复制过来）。当然，由于这种调整而产生的新参数是需要重新训练的，而其他参数不需要重新训练。



5.3.2 VGG16 的迁移案例

【例 5.2】 从 VGG16 迁移出若干个网络层来构建新的网络。

假设需要对 224×224 的灰度图像进行二分类，要求使用 VGG16 中第 3、第 5 和第 31 行，见下面代码所示的卷积层。

```
vgg16 = models.vgg16(pretrained=True).to(device)
conv1 = nn.Conv2d(1, 3, 3) #(1, 3), 新定义
conv2 = vgg16.features[0] #(3, 64), 来自 VGG16, 参数需要冻结
conv3 = vgg16.features[2] #(64, 64), 来自 VGG16, 参数需要冻结
conv4 = nn.Conv2d(64, 512, 3) #(64, 512), 新定义
conv5 = vgg16.features[28] #(512, 512), 来自 VGG16, 参数需要冻结
```



5.3.2 VGG16 的迁移案例

然后冻结来自 VGG16 的网络层的参数：

```
L = [conv2,conv3,conv5] #对这些网络层上的参数进行冻结
for layer in L:
    for param in layer.parameters():
        param.requires_grad = False
```

接着定义全连接网络层，构造分类网络：

```
self.fc1 = nn.Linear(512*6*6, 2048)
self.fc2 = nn.Linear(2048, 1024)
self.fc3 = nn.Linear(1024, 2)
```

5.3.2 VGG16 的迁移案例

最后编写测试代码和参数统计代码，核心代码如下：

```
import torch
import torch.nn as nn
from torchvision import models
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vgg16 = models.vgg16(pretrained=True).to(device)
conv1 = nn.Conv2d(1, 3, 3) #(1, 3), 新定义
conv2 = vgg16.features[0] #(3, 64), 来自 VGG16, 参数需要冻结
conv3 = vgg16.features[2] #(64, 64), 来自 VGG16, 参数需要冻结
conv4 = nn.Conv2d(64, 512, 3) #(64, 512), 新定义
conv5 = vgg16.features[28] #(512, 512), 来自 VGG16, 参数需要冻结
L = [conv2, conv3, conv5] #对这些网络层上的参数进行冻结
for layer in L:
    for param in layer.parameters():
        param.requires_grad = False
```

该程序的核心代码（全部代码见教材P134）

5.3.2 VGG16 的迁移案例

执行上述代码，输入结果如下：

该模型的参数总数为：42544992，其中可训练的参数总数为：40146464，占的百分比为：
94.36%

输入和输出的形状分别为： torch.Size([16, 1, 224, 224]) torch.Size([16, 2])

从这个例子中，读者不难举一反三，总结从 VGG16 中迁移任意若干个网络层来构造新网络的方法。

5.3.3 预训练网络迁移的基本原理

【例 5.3】GoogLeNet 的迁移案例。

该例使用已训练好的 GoogLeNet 模型来对数据集 flower_photos 进行分类。该数据集经常用于图像分类教学，下载自 http://download.tensorflow.org/example_images/flower_photos.tgz。下载并解压后，产生五个目录，保存于 ./data/flower_photos 目录下，

如右图所示。这五个目录名分别表示雏菊花、蒲公英、玫瑰花、葵花和郁金香五种花。这些花的图像文件分别保存在相应的目录下，文件数量分别为 633、898、641、699 和 799，总数为 3670。



图 5-7 数据集 flower_photos 的目录结构

5.3.3 预训练网络迁移的基本原理

该程序的代码编写步骤如下。

(1) 编写加载数据及打包数据的代码基本思路：

- ① 先定义函数 `getFileLabel(tmp_path)`，其作用是：读取每个文件的相对路径（含文件名）及其类别（分别用 0、1、2、3、4 对类别编号），形成以二元组(路径, 类别编号)为元素的列表。
- ② 然后，划分训练集和测试集，并通过定义数据集类 `FlowerDataSet(Dataset)`，以训练集和测试集作为输入，将它们分别映射为该类的实例 `train_dataset` 和 `test_dataset`。
- ③ 最后，用 `DataLoader()`类对 `train_dataset` 和 `test_dataset` 进行打包，形成两个实例 `train_loader` 和 `test_loader`。具体代码见随后列出的程序代码。



5.3.3 预训练网络迁移的基本原理

(2) 用下列语句下载已训练好的 GoogLenet 模型:

```
googlenet_base = models.googlenet(num_classes=5, init_weights=True)
```

其中, num_classes=5 表示模型的类别个数为 5。显然, num_classes=1000 表示下载模型的类别为 1000。init_weights=True 表示同时下载参数, 否则模型将随机初始化参数。该模型比较大, 建议先 torch.save() 函数将模型保存到磁盘, 以后调试时利用 torch.load() 函数从磁盘中加载模型, 否则每次调试都花费时间等待。

5.3.3 预训练网络迁移的基本原理

(3) 更新模型 googlenet_base 的参数。为此，先从下列地址下载模型的参数文件：

<https://download.pytorch.org/models/googlenet-1378be20.pth>。该参数文件保存了至目前为止最好的参数

(比初始参数要好得多)，因此最好用该文件中的参数更新上面下载的模型的参数。但该文件默认适用于类别为 1000 的模型，而上面下载的模型的类别为 5。我们先 `print(googlenet_base)` 语句查看该模型的层次结构，结果如右。

```
(aux1): InceptionAux(
  (conv): BasicConv2d(
    (conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (fc1): Linear(in_features=2048, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=5, bias=True)
)
(aux2): InceptionAux(
  (conv): BasicConv2d(
    (conv): Conv2d(528, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (fc1): Linear(in_features=2048, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=5, bias=True)
)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(dropout): Dropout(p=0.2, inplace=False)
(fc): Linear(in_features=1024, out_features=5, bias=True)
)
```

5.3.3 预训练网络迁移的基本原理

下划线的三行代码都是表示输出类别为 5 的全连接输出层，它们的结构跟参数文件 googlenet-1378be20.pth 的结构不匹配，因此该参数文件不能更新这个全连接层。所以，我们用该参数文件更新这三个全连接层以外的其他网络层的参数，代码如下：

```
model_dict = googlenet_base.state_dict()  
#从磁盘加载最新的参数文件  
pretrain_model = torch.load(f"./pre_models/googlenet-1378be20.pth")  
#googlenet-1378be20.pth 类别数量为 1000,此处为 5,故不能更新这几个网络层的参数  
del_list = ["aux1.fc2.weight", "aux1.fc2.bias",  
            "aux2.fc2.weight", "aux2.fc2.bias",  
            "fc.weight", "fc.bias"] #不能被更新的参数  
pretrain_dict = {k: v for k, v in pretrain_model.items() if k not in del_list}  
#用 googlenet-1378be20.pth 中的参数值更新模型参数  
googlenet_base.model_dict.update(pretrain_dict)
```



5.3.3 预训练网络迁移的基本原理

(4) 冻结部分网络层的参数。为此，先冻结所有的参数：

```
for param in googlenet_base.parameters(): #先冻结所有的参数
    param.requires_grad = False
```

然后解冻五个全连接层的参数，表示这些参数是待学习参数：

```
layers = [ googlenet_base.aux1.fc1, googlenet_base.aux1.fc2,
            googlenet_base.aux2.fc1, googlenet_base.aux2.fc2,
            googlenet_base.fc]
for layer in layers:
    for param in layer.parameters():
        param.requires_grad = True
```

5.3.3 预训练网络迁移的基本原理

模型训练和测试的代码：

在训练模式下，调用模型后返回值的类型是 `torchvision.models.googlenet.GoogLeNetOutputs`，不是张量，因此需要用 `logits` 属性获得返回值的张量，相应代码如下：

```
pre_y = googlenet_model(x)
pre_y = pre_y.logits
```

而在测试模式下（`googlenet_model.eval()`），`googlenet_model(x)`返回的是张量，就不能用第二条语句了。

该程序的部分代码如下：

5.3.3 预训练网络迁移的基本原理

```
#下载模型需要时间，笔者已经将模型下载并放在指定的目录（./pre_models）了
googlenet_base = models.googlenet(num_classes=5,init_weights=True)
torch.save(googlenet_base,'./pre_models/googlenet_base5')
'''
#加载已下载的模型，类别个数为 5
googlenet_base = torch.load('./pre_models/googlenet_base5')
model_dict = googlenet_base.state_dict()
#加载最新的模型参数
pretrain_model = torch.load(f'./pre_models/googlenet-1378be20.pth')
#googlenet-1378be20.pth 对应类别数量为 1000，此处为 5，
#故不能更新这几个网络层的参数
del_list = ["aux1.fc2.weight", "aux1.fc2.bias",
            "aux2.fc2.weight", "aux2.fc2.bias",
            "fc.weight", "fc.bias"]
pretrain_dict = {k: v for k, v in pretrain_model.items() if k not in del_list}
#用 googlenet-1378be20 中的参数值更新模型 googlenet_base
model_dict.update(pretrain_dict)
googlenet_base.load_state_dict(model_dict)
#-----
#先冻结所有的参数
for param in googlenet_base.parameters():
```

```
    param.requires_grad = False
#再解冻五个全连接层的参数：
layers = [googlenet_base.aux1.fc1, googlenet_base.aux1.fc2,
          googlenet_base.aux2.fc1, googlenet_base.aux2.fc2,
          googlenet_base.fc]
for layer in layers:
    for param in layer.parameters():
        param.requires_grad = True
googlenet_model = googlenet_base.to(device)
optimizer = optim.Adam(googlenet_model.parameters())
```

该程序的核心代码（全部代码见教材P138）

5.3.3 预训练网络迁移的基本原理

执行上述代码，结果如下（部分）：

```
... ..  
9 0.4279835522174835  
9 0.22218433022499084  
运行时间： 1.9098375598589579 分钟  
在测试集上的准确率： 0.8746594190597534
```

该结果表明，只在运行 10 代的情况下，即可达到 0.87 的准确率。这说明，上述迁移方法对此类数据集是相对有效的。

5.3.4 ResNet 的迁移案例

【例 5.4】ResNet 的迁移案例：

加载预训练模型 ResNet50 的代码如下：

```
resnet50 =
```

```
models.resnet50(pretrained=True)
```

但我们目前不知道该模型长成什么样子，不知从何入手对其结构进行更改。为此，一般的做法是打印该模型的层次结构（`print(resnet50)`），结果如右图所示。

```
(2): Bottleneck(
  (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=2048, out_features=5, bias=True)
)
```

5.3.4 ResNet 的迁移案例

从上图中可以看出，最后一层是全连接层，我们可以对该层进行微调（当然，也可以对其他有关的网络层进行修改，但一般不建议这么做），并冻结除了该层以外的其他层参数，代码如下：

```
resnet50.fc = nn.Linear(2048, 5) #改为最后一层有 5 个输出节点， 因为是 5 分类
for param in resnet50.parameters(): #先冻结全部参数
    param.requires_grad = False
for param in resnet50.fc.parameters(): #在解冻最后一层的参数
    param.requires_grad = True
```

5.3.4 ResNet 的迁移案例

模型创建和调整已经完成。此后，训练和测试的代码跟例 5.3 相似。程序核心代码如下所示：

```
#给定数据集，测试在其上的准确率：
def getAccOnadataset(data_loader):
    resnet50.eval()
    correct = 0
    with torch.no_grad():
        for i, (x, y) in enumerate(data_loader):
            x, y = x.to(device), y.to(device)
            pre_y = resnet50(x)
            pre_y = torch.argmax(pre_y, dim=1)
            t = (pre_y == y).long().sum()
            correct += t
    correct = 1. * correct / len(data_loader.dataset)
    resnet50.train()
return correct.item()
start=time.time() #开始计时
resnet50.train()
```

该程序的核心代码（全部代码见教材P141）

5.3.4 ResNet 的迁移案例

运行该程序，输入结果如下：

```
... ..  
9 0.26216524839401245  
9 0.05788884684443474  
运行时间： 2.4728100061416627 分钟  
在测试集上的准确率： 0.893733024597168
```

可见，该程序获得 0.89 的准确率，相对比较高。这说明，该迁移方法对此类数据集是比较有效的。



5.3.5 EfficientNet 的迁移案例

【例 5.5】EfficientNet 的迁移案例：

该例子使用 EfficientNet 作为预训练模型，解决的问题也跟例 5.3 一样，都是对数据集 flower_photos 进行分类。类似地，其加载和打包数据的代码也跟例 5.3 完全一样，故在此不做介绍。

导入 EfficientNet-B7，这是一种非常优秀的 EfficientNet 网络模型：

```
from efficientnet_pytorch import EfficientNet  
effi_model = EfficientNet.from_pretrained('efficientnet-b7').to(device)
```

注意，系统提示可以导入 EfficientNet 网络模型包括：efficientnet-b0, efficientnet-b1, efficientnet-b2, efficientnet-b3, efficientnet-b4, efficientnet-b5, efficientnet-b6, efficientnet-b7, efficientnet-b8, efficientnet-l2，但笔者只成功导入 efficientnet-b7。

5.3.5 EfficientNet 的迁移案例

为了解模型的结构，用 `print(model)` 打印出它的层次结构，然后查看哪些网络层可以利用和修改。比如，上述导入的模型的层次结构如下图所示。

```
(_conv_head): Conv2dStaticSamePadding(
  640, 2560, kernel_size=(1, 1), stride=(1, 1), bias=False
  (static_padding): Identity()
)
(_bn1): BatchNorm2d(2560, eps=0.001, momentum=0.010000000000000009,
(_avg_pooling): AdaptiveAvgPool2d(output_size=1)
(_dropout): Dropout(p=0.5, inplace=False)
(_fc): Linear(in_features=2560, out_features=1000, bias=True)
(_swish): MemoryEfficientSwish()
)
```

5.3.5 EfficientNet 的迁移案例

从上图中可以看出，最后一个全连接层名为“_fc”，其输入节点数为 2560，输出节点数为 1000。

因此，我们可以修改这个网络层，以适合本例 5 分类任务。同时，我们在修改该层网络之后，再增加两个全连接层。程序核心代码如下：

```
model = EfficientNet.from_pretrained('efficientnet-b7').to(device)
for param in model.parameters():
    param.requires_grad = False
feature = model._fc.in_features
model._fc = nn.Linear(in_features=feature,out_features=4096,bias=True) #改变输出层
fc1 = nn.Linear(4096, 2048)
fc2 = nn.Linear(2048, 5)
class EfficientNet(nn.Module):
def __init__(self, model_name='tf_efficientnet_b3_ns', pretrained=True):
    super().__init__()
    self.model = model #利用预训练模型
    self.fc1 = fc1 #增加两个全连接层
    self.fc2 = fc2
def forward(self, x):
    o = x
    o = self.model(o)
    o = nn.ReLU(inplace=True)(o)
    o = nn.Dropout(p=0.5, inplace=False)(o)
    o = self.fc1(o)
    o = nn.ReLU(inplace=True)(o)
    o = nn.Dropout(p=0.5, inplace=False)(o)
    o = self.fc2(o)
    return o
efficient_model = EfficientNet().to(device)
optimizer = optim.Adam(efficient_model.parameters())
```

该程序的核心代码（全部代码见教材P143）

5.3.5 EfficientNet 的迁移案例

该程序执行了 20 代，输出结果如下：

... ..

19 0.15419840812683105

19 0.8684183955192566

运行时间： 11.100697712103527 分钟

在测试集上的准确率： 0.856494128704071

该程序在测试集上获得的准确率为 0.85，略低于前面两个程序。这也许说明，好的深度模型未必在所有的数据集上都能获得绝对的好结果，这还需要丰富的调参经验为指导。你们不妨试着修改上面的模型，看看怎么改进才能获得更好的结果。

本章内容：

- 几种经典的卷积神经网络预训练模型
- VGG16, GoogLeNet, ResNet, EfficientNet
- 各个网络的迁移学习