**第6章**

# 卷积神经网络

|  |  |
| --- | --- |
|  | 本 章 内 容 |
|  | ◎ 卷积神经网络简介  ◎ 卷积神经网络原理  ◎ 卷积神经网络结构  ◎ keras实现经典的卷积神经网络 |

## 6.1 卷积神经网络简介

卷积神经网络（Convolutional Neural Network,CNN）是一种常见的深度学习架构，其初期主要是用来解决图像识别的问题，但早期由于缺乏训练数据和计算能力，要在不产生过拟合的情况下训练高性能卷积神经网络是很困难的。近年来GPU的发展，人们给与卷积神经网络的关注一直在持续的到加强，因此也让卷积神经网络发展迅速，其表现的应用已经不仅仅只应用在图像方面了，还可用于能够表示张量的任意类型的数据，包括语音识别、自然语言处理等方面，在2014年，微软研究院就公开了一篇关于使用CNN来处理音频识别的文章。

ImageNet，一个目前最为常用的包含图像标注信息的数据库，2007年由斯坦福大学的华人教授李飞飞创办，目前大部门的网络都会采用该数据来验证自己的效果。ImageNet每年也会举行一场大规模的图像识别的比赛，简称ILSVRC，参赛的选手使用ImageNet提供的图像信息来做图像识别与分类，并采用top5与top1的分数来评估模型的好坏。卷积神经网络的出现，也迅速打破了该比赛的最好记录。

卷积神经网络受生物自然视觉认知机制启发而来，20世纪 90 年代，LeCun et al. 等人发表论文，确立了CNN的现代结构，后来又对其进行完善。他们设计了一种多层的人工神经网络，取名叫做LeNet-5，可以对手写数字做分类。2006年起，人们设计了很多方法，想要克服难以训练深度CNN的困难。其中，最著名的是 Krizhevsky et al.提出了一个经典的CNN 结构，并在图像识别任务上取得了重大突破。其方法的整体框架叫做 AlexNet，与 LeNet-5 类似，但要更加深一些。AlexNet 取得成功后，研究人员又提出了其他的完善方法，其中最著名的要数 VGGNet, GoogleNet和 ResNet这三种。从结构看，CNN 发展的一个方向就是层数变得更多，ILSVRC 2015 冠军 ResNet 是 AlexNet 的20 多倍，是 VGGNet 的8 倍多。通过增加深度，网络便能够利用增加的非线性得出目标函数的近似结构，同时得出更好的特性表征。但是，这样做同时也增加了网络的整体复杂程度，使网络变得难以优化，很容易过拟合,当然，研究人员们也提出了很多方法来解决这一问题。下图对比了目前各种卷积神经网络之间，复杂度和精度之间的关系。

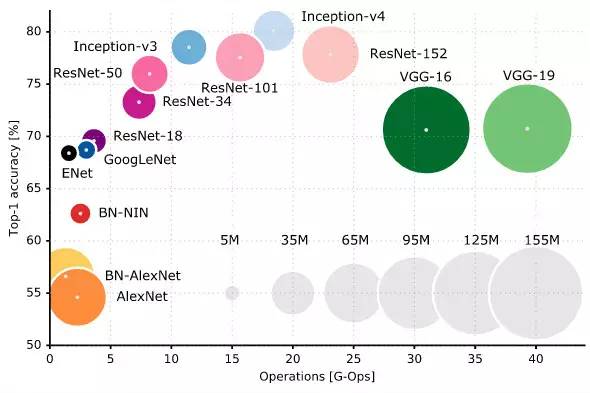


图 6.1-1 卷积神经网络复杂度与精度的关系

本章之后的内容将重点介绍卷积神经网络的原理，并讲解如何使用Keras来搭建卷积神经网络，并按照CNN的发展时间先后，分别介绍LeNet、AlexNet、VGGNet、ResNet这四个在在CNN发展过程中有重要意义的网络。

## 6.2卷积神经网络的原理

卷积神经网络通过卷积来模拟特征区分，并且通过卷积的权值共享及池化，来降低网络参数的数量级，最后通过传统神经网络完成分类等任务。

考虑一个问题，那么为什么不用传统的神经网络来做图像识别呢？从下面的例子就可以很容易的理解。如果我们采用传统的神经网络来处理一张1000\*1000像素的黑白图片，即只有一个颜色通道，那么一张图片就有100万个像素点，如果连接一个相同大小的隐藏层，那么将产生100万×100万=1万亿个连接，这还仅仅是一层全连接层，计算量就已经无法接受了。我们必须减少需要训练的权重数量，一是降低训练的复杂度，二是过多的连接很容易造成过拟合，减少连接可以降低模型的泛华能力。

图像在空间上是有组织结构的，每一个像素点在空间上和周围的像素点是紧密联系的，但是和太远的点就没有太大的关系了，因此，每一个神经元并不需要接受所有的像素点的信息，只需要接受局部的像素点作为输入，而后将这些局部信息综合起来就可以得到全局的信息。这样就把之前的全连接改变成了局部连接，如果我们取的局部信息大小是10×10，那么现在就只有10×10×100万=1亿个连接，相比之前的1万亿缩小了1万倍。

虽然我们从1万亿降低到了1亿，但是数量还是很多，因此，我们引入了这样一个想法，让每一个隐藏层的节点参数一样，所以我们的参数最终只有10×10=100个，这10×10的权重组成的矩阵就被称为卷积核，100即卷积核的大小，卷积核和原图像相乘即称为卷积操作。可以看到无论原图像有多大都是100个权重，这就是卷积核的作用。我们不需要担心有多少个隐藏节点，图像有多大，参数量只和卷积核的大小有关系，这就是权值共享。当然但是如果我们只采用一个卷积核显然是不够的，每一个卷积核只能提取出图像中的一种特征，如果我们引入了多个卷积核，即可提取图像中多种特征，好在图像中的特征并不多，每一张图像都是由最基础的点、线组成，当神经元接收到这些点线特征后，传到下一层再组合成更高级的特征，比如三角形，正方形，再继续抽象出眼睛、鼻子，最后五官组合成了一张脸，从而完成了图像识别。因此我们的问题就很好解决了，只需要提供更多的卷积核，提取出更多的特征，一般来说，我们把100个卷积核放在第一个卷积层就很充足了，这样的话，我们的参数就是100×100=1万，相比之前的1亿我们又降低了10000倍。因此依靠卷积操作，我们就可以高效的训练出一个局部连接的神经网络了，我们也把这样结构的网络称为卷积神经网络。

## 6.3卷积神经网络的结构

卷积神经网络一般由卷积层、池化层、全连接层组成。其中卷积层与池化层配合，组成多个卷积组，逐层提取特征，最终通过若干个全连接层完成对应的识别、分类等任务。

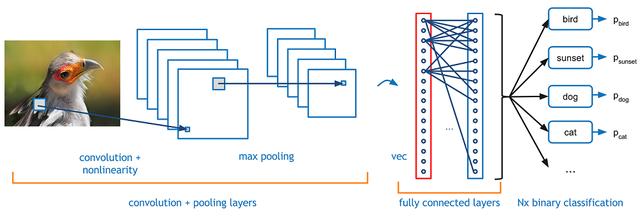


图 6.3-1 卷积神经网络的基本结构

**1.卷积层（Convolution）**

对图像进行卷积操作实际的操作过程是一个滑动的窗口对原始图像像素做点乘然后求和。如图所示，我们有一个5x5的图像，我们用一个3x3的卷积核做卷积操作，如果我们的滑动步长（卷积核每次移动的格数）是1，可得到：

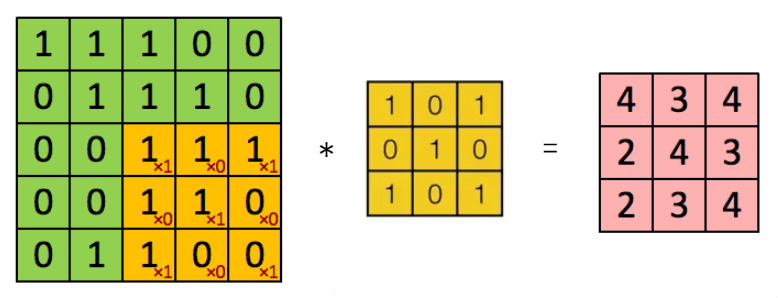


图6.3-2 卷积操作

假设输入图像大小为卷积核大小为步长为，则输出图像大小为：

但是这样做卷积运算是有一个缺点的，卷积图像的大小会不断缩小，另外图像的角落的元素只被一个输出所使用，所以在图像边缘的像素在输出中采用较少，也就意味着会丢失掉很多图像边缘的信息，为了解决这两个问题，就引入了padding操作，即在图像卷积操作之前，沿着图像边缘用0进行图像填充，就可以保证输出图像和输入图像一样大，

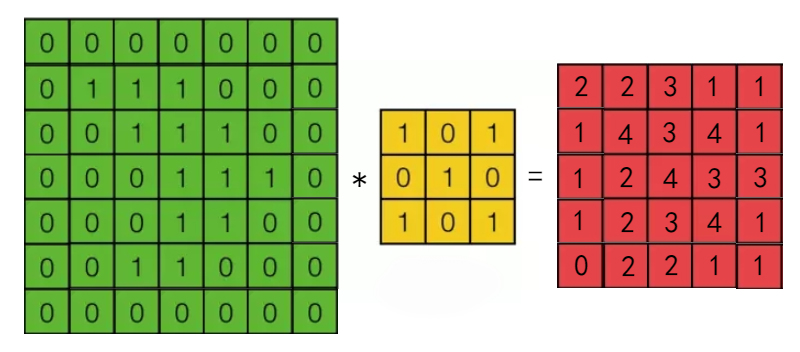


图6.3-3 引入padding的卷积操作

假设输入图像大小为过滤器大小为步长为引入的padding为，则输出图像大小为：

在keras中，与卷积操作相关的API均有特定的参数padding来控制是否为图像填充0，它的取值可以是SAME或者VALID，其中VALID代表只进行有效的卷积，即对边界数据不处理。SAME代表保留边界处的卷积结果，自动为图像填充0，通常会导致输出shape与输入shape相同。

对于单色道的黑白图片，我们只需要一个信道，如果想要在RGB图像上进行卷积操作，卷积核的大小就不在是而是，最后的3对应为信道数。卷积时就是图像中每个的卷积核对应的值与图像的值进行相乘累加。

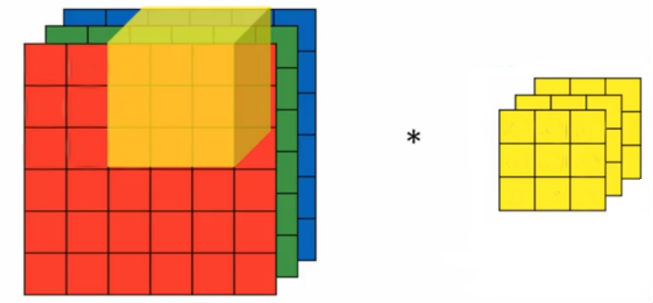


图6.3-4 RGB图像的卷积

在上文中也提到过，在具体应用中我们往往会检测多种特征，例如同时检测垂直边缘、水平边缘、45度边缘等等，也就是多个卷积核的问题。对于上面的情况，在只有一个卷积核的时候最后生成的图片是二维的，如果有个卷积核，最终生成图像为三维的的张量。

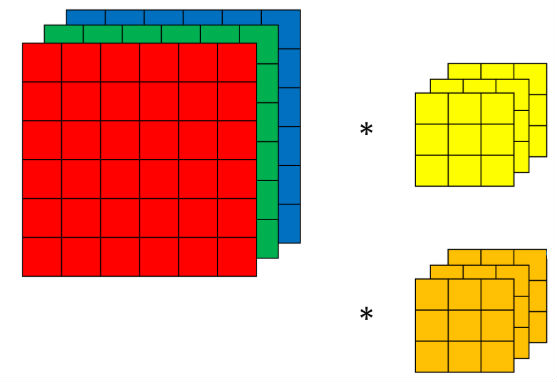


图6.3-5 多卷积核

**2.池化层 （Pooling）**

池化层是CNN的重要组成部分，通过减少卷积层之间的连接，降低运算复杂程度，从而减少过拟合，并通过减小输入的尺寸来提高性能。其次池化层也可用于对输入进行降采样，并且保留了较为关键的信息，卷积层也可以做类似的操作，但是其效率会低很多。池化层一般有两种形式，最大池化层和平均池化层。

**2.1最大池化层（max-pooling）**

最大池化思想很简单，以下图为例，把4×4的图像分割成4个不同的区域，然后输出每个区域的最大值，这就是最大池化所做的事情。这里当然也可以理解为选择了2×2的过滤器，步长为2，但不是进行卷积操作，而是变成了选取最大值的操作。当输入数据的灰度与图像中的重要性相关时，这种池化操作非常有用，在一幅真正的图像中提取最大值也可能意味着提取了某些特定特征，比如垂直边缘、一只眼睛等等。最大池化层通常是利用2×2的过滤器完成对应操作的，使用2×2的原因一般在于它是能够进行降采样的最小单位，如果使用1×1的过滤器，那么输入和输出相同。

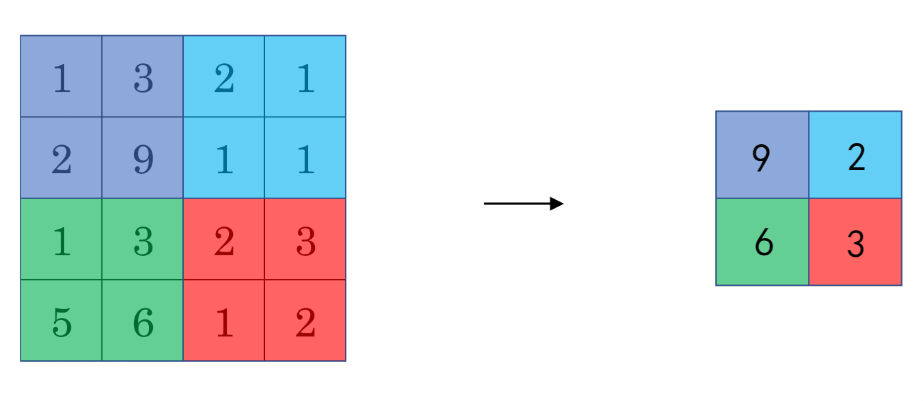


图6.3-6 最大池化层

**2.2平均池化层（average-pooling）**

平均池化和最大池化唯一的不同是，它计算的是区域内的平均值而最大池化层计算的是最大值。当整个卷积核的值都非常重要时，如果想实现池化操作，平均池化层也是非常有用的，但在日常应用中使用最多的还是最大池化。

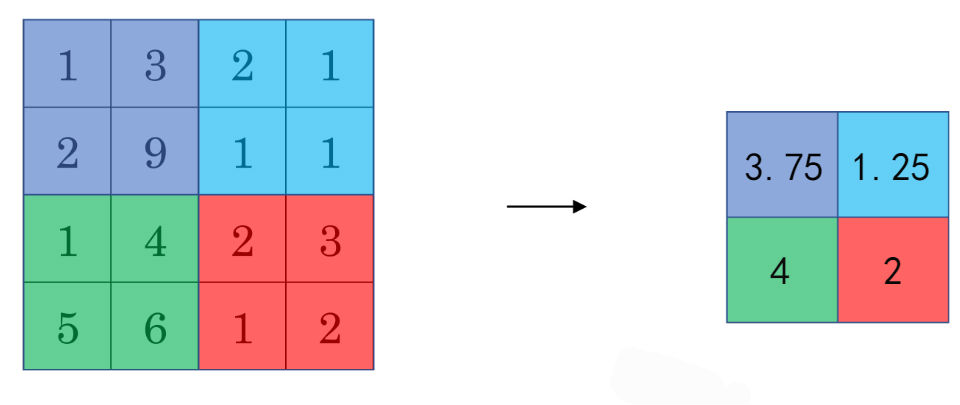


图6.3-7 平均池化层

**3.激活层 （Activation）**

在经过池化层后，通常会对输出进行一个非线性映射，因为卷积计算和池化计算是一种线性计算，如果不引入非线性映射的话，无论有多少层神经网络，输出都是输入的线性组合，这与一层隐藏层的效果相当。激活函数是否可用，需要考虑以下两点。

* 该函数是否是单调的，这样输出便会随着输入的增长而增长，从而在使用梯度下降时寻找局部极值成为可能。
* 该函数是否可微，以保证该函数定义域内的任意一点上导数都存在，从而使得梯度下降法能够正常使用来自这类激活函数的输出。

任何满足以上两点额函数都可以用做激活函数，常见的激活函数有relu、tanh、sigmoid等，在卷积神经网络中，更加常用的是relu。

rule即修正线性单元，也被称为斜坡函数，relu是分段线性的，当输入为负数时，输出均为0，当输入为非负数时，输出与输入相同。在反向传播计算梯度中，使用relu求导明显会比tanh和sigmoid简单，也因此减少了计算量。同时，使用tanh和sigmoid，当层数较多时容易导致梯度消失，因为tanh和sigmoid的导数均小于1（可参考激活函数的导数公式），当我们神经网络有多层的时候，每层都要乘以这个小于1的导数，就有可能接近于0，这就是所谓的梯度消失。而使用relu求导，若输出不为0时，导数均为1，可以有效避免梯度消失问题。另外，relu还会将小于0的映射为0，使得网络较为稀疏，减少神经元之间的依赖，避免过拟合。当然，对应的缺点就是如果学习率使用的太大，很难收敛。

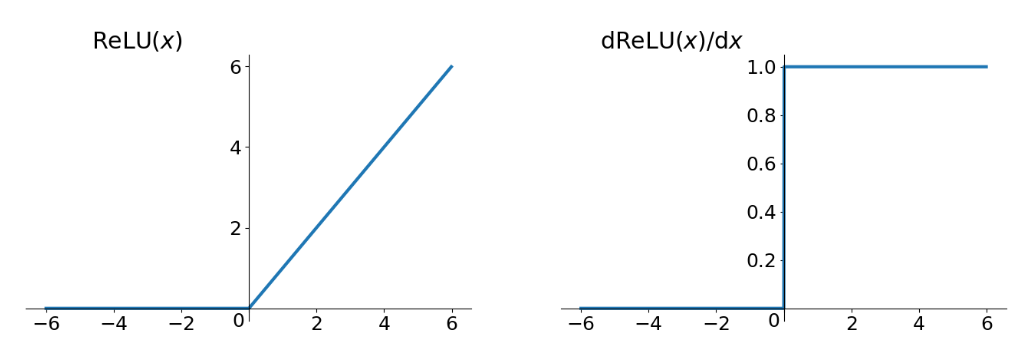


图6.3-8 relu激活函数

**4.全连接层 （Fully Connect）**

当获取到足以用来识别图片的特征后，接下来的就是如何进行分类。全连接层就可以用来将最后的输出映射到线性可分的空间。通常卷积网络的最后会将末端得到的高维数据平摊为一维的向量，并送入全连接层配合sigmoid层或softmax层进行最后的分类。

这里也体现了深度神经网络或deep learning之所以称deep的一个原因：模型将特征抓取层和分类层合在了一起，负责特征抓取的卷积层主要是用来学习“如何观察”。

## 6.4 Keras实现经典的卷积神经网络

本章将介绍四种经典的卷积神经网络，分别是LeNet、AlexNet、VGGNet与ResNet，这四种网络依照出现的先后顺序排列，深度和复杂度也一次递增。这四个卷积神经网络都在各自的年代使用了先进的网络结构，对于深度学习来说都有很大的推进作用，也象征着这几年神经网络的快速发展。

**1. Keras实现LeNet**

在了解完CNN的基本概念之后，本节会带领读者用keras来实现一个简单的卷积神经网络LeNet，LeNet是一个用来识别手写数字的最经典的卷积神经网络，是Yann LeCun在1998年设计并提出的，也是早期卷积神经网络中最有代表性的。本节使用的数据是也是手写数字数据集MNIST，MNIST 数据集来自美国国家标准与技术研究所, 数据集由来自 250 个不同人手写的数字构成,包括0-9共10个数字，我们的任务是正确识别出手写数字。本节将会构建一个非常简单并且有代表性的卷及神经网络，预期可达到99%的准确率，读者可通过该例子掌握keras搭建卷及神经网络的要点。

首先载入MNIST数据集，这里直接采用keras内置的获取mnist数据的方法，该方法会把数据下载到对应的目录中，因此执行以下方法时读者需等待片刻。当然读者也可自行下载，地址为：http://yann.lecun.com/exdb/mnist/ 数据读取后我们对label做一个one-hot处理，因为是10分类问题，所以num\_classes这个参数的值为10。

|  |  |
| --- | --- |
| 1 | from keras.datasets import mnist |
| 2 | from keras.utils import to\_categorical |
| 3 | (train\_X, train\_y), (test\_X, test\_y) = mnist.load\_data() |
| 4 | y\_train = to\_categorical(y\_train, num\_classes=10) |
| 5 | y\_test = to\_categorical(y\_test, num\_classes=10) |

数据读取完成后，我们对数据的维度进行查看。

|  |  |
| --- | --- |
| 1 | train\_X.shape |
| 2 |  |
| 3 | out: |
| 4 | (55000, 784) |

可以看到，我们读取的数据是二维的，第一个维度表示的是图片的数量，第二个维度是图片的像素集，我们需要把这一维的数据转换为一个28×28×1的数据集，其中28表示图片的长宽，由于图片是单色道即黑白图片，因此第三个维度是1。我们这里使用reshape方法来改变数据维度。

|  |  |
| --- | --- |
| 1 | train\_X = train\_X.reshape(-1, 28, 28, 1) |
| 2 | test\_X = test\_X.reshape(-1, 28, 28, 1) |

接下来就是建模的过程，由于模型比较简单，我们这里采用序贯模型Sequential。首先，我们创建第一个卷基层，其中filters表示卷积核的个数，kernel\_size表示卷积核的大小，padding表示填充方式，其中包括“valid”与“same”，“valid”代表只进行有效的卷积，即对边界数据不处理，“same”代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同，input\_shape表示输入数据的维度，注意，如果是第一层卷基层，必须提供该参数。接下来我们引入一些非线性的变化操作添加一个relu激活函数。最后，使用一个2×2的最大池化层对卷积的输出结果做池化操作，其中pool\_size表示池化层的大小。

|  |  |
| --- | --- |
| 1 | from keras.layers import Activation, MaxPooling2D, Dropout, Flatten, Dense, Conv2D |
| 2 | from keras.models import Sequential |
| 3 | model = Sequential() |
| 4 | model.add(Conv2D(filters=32,kernel\_size=(5,5),padding='valid',  input\_shape=(28, 28, 1))) |
| 5 | model.add(Activation('relu')) |
| 6 | model.add(MaxPooling2D(pool\_size=(2, 2))) |

然后定义我们的第二个池化层，这个卷基层和第一层类似，只是卷积核的数量改为了64。

|  |  |
| --- | --- |
| 1 | model.add(Conv2D(filters=64,kernel\_size=(5,5),padding='valid')) |
| 2 | model.add(Activation('relu')) |
| 3 | model.add(MaxPooling2D(pool\_size=(2, 2))) |

在经过上面两个卷基层之后，我们的图片大小最后为5×5，由于最后一个卷基层的卷积核是64，所以最后拿到的输出结果维度是5×5×64。为了方便加入后面的分类，这里加入一个Flatten层，Flatten层用来将输入“压平”，即把多维的输入一维化，常用在从卷积层到全连接层的过渡。然后连接一个全连接层，并加入relu激活函数，为了防止过拟合，添加一个Dropout层，然后连接一个10维的全连接层，最后把得到的结果输入到softmax层，从而得到最后的概率。

|  |  |
| --- | --- |
| 1 | model.add(Flatten()) |
| 2 | model.add(Dense(128)) |
| 4 | model.add(Activation('relu')) |
| 5 | model.add(Dropout(0.5)) |
| 6 | model.add(Dense(10)) |
| 7 | model.add(Activation('softmax')) |

模型建立完成之后，来定义损失函数，优化算法与评估指标，其中损失函数采用多分类常使用的croos entropy，优化方法采用adam，评估指标是准确率。

|  |  |
| --- | --- |
| 1 | model.compile(loss='categorical\_crossentropy',  optimizer='adam',  metrics=['accuracy']) |

下面就是训练过程了，其中batch\_size表示mini-batch的大小，epochs表示训练的迭代次数，verbose表示是否打印日志，0为不在标准输出流输出日志信息，1为输出进度条记录，2为每个epoch输出一行记录，为了方便查看训练进度，可把verbose设置为1。

|  |  |
| --- | --- |
| 1 | model.fit(train\_X, train\_y, batch\_size=128, epochs=10, verbose=1) |

训练完成后，即可在测试集上进行测试，得到最终的测试结果。注意，对于正常的流程，通常会保留一个验证集，这里的测试集其实就可以理解为实际应用中的验证集。

|  |  |
| --- | --- |
| 1 | score= model.evaluate(test\_X, test\_y, batch\_size=128, verbose=0) |
| 2 | print('Test accuracy:', score[1]) |
| 3 |  |
| 4 | out: |
| 5 | Test accuracy: 0.9923 |

score的结果是一个二维的数组，其中第一个值表示的是该输入数据是负样本的概率，第二个值表示的是该输入数据是正样本的概率，这里需要查看的是准确率，所以是第二个值。可以看到，这么简单的一个卷积神经网络，准确率已经高达99.37%，可见卷积层对图像的特征提取是十分有用的，并且依靠卷积核的参数共享，训练效率得到了很大的提升。在后面的文章中，会给大家介绍一些更加复杂且效果更好的卷积神经网络。

**2. Keras实现AlexNet**

2012年，ImageNet的比赛ILSVRC冠军的model——AlexNet（以第一作者alex命名）。这个模型的意义很大，首先它证明了CNN在复杂模型下的有效性，GPU实现使得训练在可接受的时间范围内得到结果，确实让CNN和GPU都大火了一把，顺便推动了有监督深度学习的发展。

相比于LeNet来说，AlexNet结构更加复杂了，其次也新增了以下的优点：

* AlexNet使用ReLu代替了sigmoid，其能更快的训练，同时解决了sigmoid在训练较深的网络中出现的梯度消失的问题。
* 使用dropout随机断开一些神经元的连接，避免过拟合。
* 在以前的CNN中普遍使用平均池化层average pooling, AlexNet全部使用最大池化层 max pooling，避免了平均池化层的模糊化的效果，并且步长比池化的核的尺寸小，这样池化层的输出之间有重叠，提升了特征的丰富性。
* 提出LRN层，局部响应归一化，对局部神经元创建了竞争的机制，使得其中响应较大的值变得更大，响应小的值变得更小，增强模型的泛华能力，但是效果并不明显，且对前向传播与反向传播的速度有大大的影响，目前除了AlexNet并没有其他卷积神经网络采用LRN层，下文的代码中也将不会添加LRN层。

AlexNet其模型结构见下图，虽然只有寥寥八层，但是它有60M以上的参数总量。

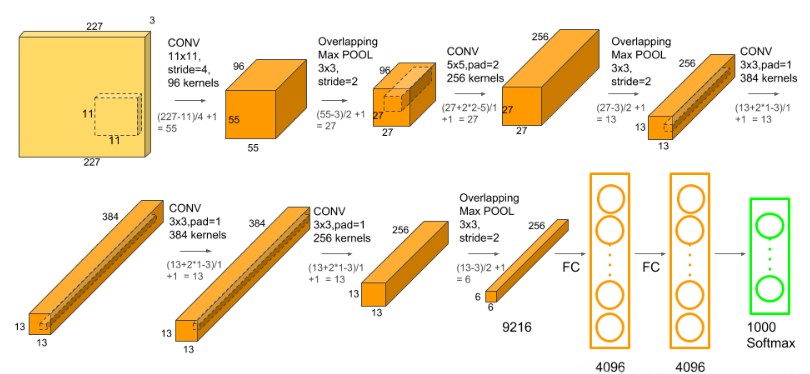


图 6.4-1 AlexNet结构

从图中可以看到，AlexNet一共有八个卷积层，三个全连接层，详细看一下每一层做了些什么操作。

第一个层 conv\_1：

* 输入的图片大小为227×227×3
* 有96个大小为11×11的卷积核，步长为4，无padding
* 输出大小为55×55×96
* 池化层采用3×3的最大池化层，步长为2

第二个层 conv\_2:

* 输入的tensor为27×27×96
* 有256个大小为5×5的卷积核，步长为1，padding为2
* 输出大小为27×27×256
* 池化层采用3×3的最大池化层，步长为2

第三个层 conv\_3:

* 输入tensor为13×13×256
* 有384个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为13×13×384
* 无池化层

第四个层 conv\_4:

* 输入tensor为13×13×384
* 有384个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为13×13×384
* 无池化层

第五个层 conv\_5:

* 输入tensor为13×13×384
* 有256个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为13×13×258
* 池化层采用3×3的最大池化层，步长为2

第六个层 fc\_1:

* 输入tensor为9216个元素的一维向量
* 4096个节点的全连接层

第七个层 fc\_2:

* 输入tensor为4096个元素的一维向量
* 4096个节点的全连接层

第8个层 fc\_3:

* 输入tensor为4096个元素的一维向量
* 输出为1000个元素的一维向量
* 最后加入一层softmax，输出为1000个种类的各类率值

接下来就会带领读者用keras来实现AlexNet，并采用cifar-10这个数据集来对结果做一个验证，cifar-10这个数据集共有60000张彩色图像，每张图像都是32×32，分为10个类，每类6000张图，也就是说这是一个10分类的图像识别问题。

原版的AlexNet的输入图像大小为227×227，如果采用一样的参数在接下来要使用的数据集cifar-10上是行不通的，本节的关键点还是要让读者掌握AlexNet的结构与其优点所在，因此本节的代码将对卷积核做一些简单的修改，把卷积核的大小都改为3×3，并减少卷积核的数量，池化层大小都改为3×3。当然，有兴趣的读者可以自行下载ImageNet数据来对原版AlexNet进行测试，这里为大家提供下载地址：

http://academictorrents.com/details/a306397ccf9c2ead27155983c254227c0fd938e2

首先读取数据集，这里先对图像数据做一个归一化的处理，因为每一个像素点的最大值是255，所以对原数据除以255即是归一化操作。对数据做归一化能加快模型收敛。

|  |  |
| --- | --- |
| 1 | from keras.datasets import cifar10 |
| 2 | from keras.utils import to\_categorical |
| 3 | (x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data() |
| 4 | x\_train = x\_train / 255 |
| 5 | x\_test = x\_test / 255 |
| 6 | y\_train = to\_categorical(y\_train, num\_classes=10) |
| 7 | y\_test = to\_categorical(y\_test, num\_classes=10) |

模型搭建阶段依旧采用序贯模型Sequential，整体的结构和上文有略微差别，主要在于卷积核大小和池化层大小。

|  |  |
| --- | --- |
| 1 | from keras.models import Sequential |
| 2 | from keras.layers import Dense, Flatten, Dropout |
| 3 | from keras.layers.convolutional import Conv2D, MaxPooling2D |
| 4 | from keras.utils.np\_utils import to\_categorical |
| 5 |  |
| 6 | model = Sequential() |
| 7 | model.add(Conv2D(96, (3, 3), input\_shape=(32, 32, 3), |
| 8 | padding='valid', activation='relu')) |
| 9 | model.add(MaxPooling2D(pool\_size=(3, 3))) |
| 10 |  |
| 11 | model.add(Conv2D(256, (3, 3), padding='same', activtion='relu')) |
| 12 | model.add(MaxPooling2D(pool\_size=(3, 3))) |
| 13 |  |
| 14 | model.add(Conv2D(384,(3, 3), padding='same', activation='relu')) |
| 15 | model.add(Conv2D(384,(3, 3), padding='same', activation='relu')) |
| 16 | model.add(Conv2D(256,(3, 3), padding='same', activation='relu')) |
| 17 | model.add(MaxPooling2D(pool\_size=(3, 3))) |
| 18 |  |
| 19 | model.add(Flatten()) |
| 20 | model.add(Dense(1024, activation='relu')) |
| 21 | model.add(Dropout(0.5)) |
| 22 | model.add(Dense(512, activation='relu')) |
| 23 | model.add(Dropout(0.5)) |
| 24 | model.add(Dense(10, activation='softmax')) |

定义损失函数、优化方法、评估指标。

|  |  |
| --- | --- |
| 1 | model.compile(loss='categorical\_crossentropy',optimizer='adam', metrics=['accuracy']) |
| 2 | model.summary() |

这里需要特别提一下model.summary()，这个函数可以打印出整个卷积神经网络的结构，可根据该图来判断网络的参数、结构是否正确，具体如图：

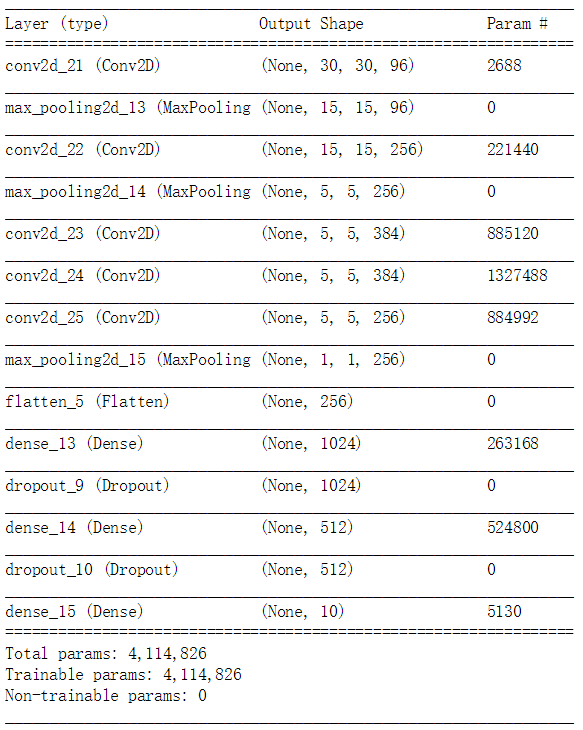


图6.4-2 summary结构图

模型准备完成后即可以开始训练，这里引入了EarlyStopping，EarlyStopping可以保证模型在自定义的某个标准不再提升或下降的前提下提前结束训练，其中monitor参数表示的是停止时参考的指标，这里采用的是准确率，patience的意思是多少轮指标没有改变模型训练就停止，这里使用的是5，也就是说如果训练五轮，准确率依旧没有提升即停止训练。

|  |  |
| --- | --- |
| 1 | from keras.callbacks import EarlyStopping |
| 2 | early\_stopping=EarlyStopping(monitor='acc',patience=5,  verbose=1) |
| 3 | model.fit(x\_train,y\_train,batch\_size=64,epochs=200, callbacks=[early\_stopping]) |
| 4 | score = model.evaluate(x\_test, y\_test, batch\_size=64) |
| 5 | print('Test accuracy:', score[1]) |
| 6 |  |
| 7 | out: |
| 8 | Epoch 1/200  50000/50000 [==============================] - 35s 698us/step - loss: 1.6917 - acc: 0.3423 |
| 9 | Epoch 2/200  50000/50000 [==============================] - 34s 681us/step - loss: 1.1557 - acc: 0.5864 |
| 10 | Epoch 3/200  50000/50000 [==============================] - 34s 682us/step - loss: 0.9287 - acc: 0.6797 |
| 11 | …… |
| 12 | Epoch 40/200  50000/50000 [==============================] - 34s 674us/step - loss: 0.1209 - acc: 0.9272 |
| 13 | Epoch 00040: early stopping  10000/10000 [==============================] - 2s 220us/step |
| 14 | Test accuracy: 0.7427 |

训练时计划是训练200轮，由于EarlyStopping的关系，在第40轮的时候就停止了，训练集的准确率为92.72%，测试集的准确率为74.27%，可见，模型存在一定的过拟合，读者也可以自行对上文的AlexNet的参数进行调整或者通过数据增强的方式，对原始图像做旋转平移等操作来增加训练数据的数量，从而提高模型的泛华能力与最终效果。

**3.Keras实现VGGNet**

VGGNet是牛津大学计算机视觉组和Google DeepMind的研究人员共同研发出来的卷积神经网络，其在2014年的ILSVRC比赛中取得了第二名的好成绩。VGGNet可以看成是加深版本的AlexNet，都是由卷积层、全连接层两大部分构成，但是VGGNet通过反复堆叠3×3的小型卷积核和2×2的最大池化层来构建网络，其整体结构十分简洁，探索了卷积神经网络的深度与其性能之间的关系。VGGNet一共有六种不同的网络结构，每种结构都有含有５组卷积操作，深度在16-19层之间。VGGNet相比于之前的网络，准确率更高，泛化能力更强，迁移到其他图片上效果也很好，依然是目前用来做图像特征提取的利器。

VGGNet的版本比较多，比较出名的是VGG-16和VGG-19，最常用的是VGG-16，这里给出各版本的结构图。

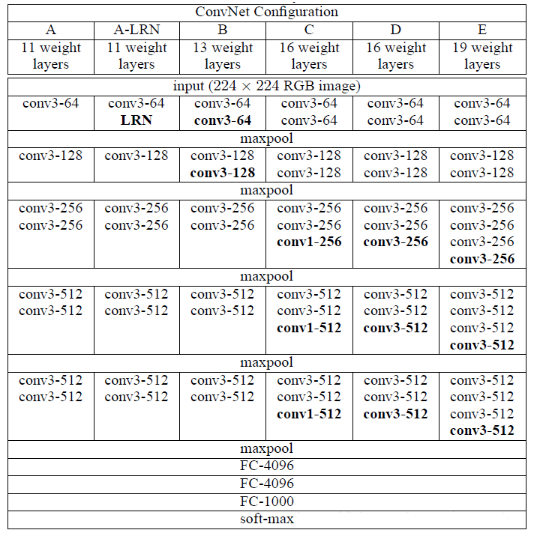


图 6.4-3 VGG各版本结构

其中，网络结构D就是著名的VGG16，下文也会围绕VGG16来讲解，VGG16的结构如下图所示。

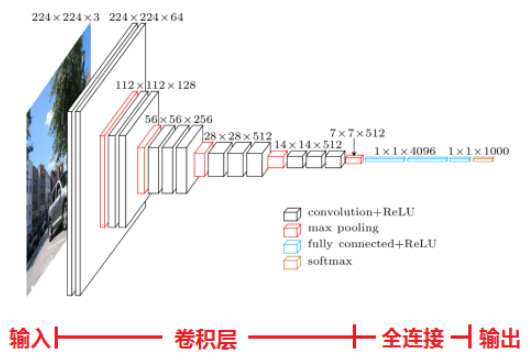


图 6.4-4 VGG16结构

可以看到VGG16由13个卷积层和3个全连接层构成，并且所有的卷积核的大小都是3×3，详细分析一下每层的结构。

第一个层 conv\_1：

* 输入的图片大小为224×224×3
* 有64个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为224×224×64

第二个层 conv\_2：

* 输入的图片大小为224×224×64
* 有64个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为224×224×64
* 池化层采用2×2的最大池化层，步长为2，池化后的尺寸变为112x112x64

第三个层 conv\_3：

* 输入的图片大小为112x112x64
* 有128个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为112x112x64

第四个层 conv\_4：

* 输入的图片大小为112x112x64
* 有128个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为112x112x64
* 池化层采用2×2的最大池化层，步长为2，池化后的尺寸变为56x56x128

第五个层 conv\_5：

* 输入的图片大小为56x56x128
* 有256个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为56x56x256

第六个层 conv\_6：

* 输入的图片大小为56x56x128
* 有256个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为56x56x256

第七个层 conv\_7：

* 输入的图片大小为56x56x128
* 有256个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为56x56x256
* 池化层采用2×2的最大池化层，步长为2，池化后的尺寸变为28x28x256

第八个层 conv\_8：

* 输入的图片大小为28x28x256
* 有512个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为28x28x512

第九个层 conv\_9：

* 输入的图片大小为28x28x256
* 有512个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为28x28x512

第十个层 conv\_10：

* 输入的图片大小为28x28x256
* 有512个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为28x28x512
* 池化层采用2×2的最大池化层，步长为2，池化后的尺寸变为14x14x512

第十一个层 conv\_11：

* 输入的图片大小为14x14x512
* 有512个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为14x14x512

第十二个层 conv\_12：

* 输入的图片大小为14x14x512
* 有512个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为14x14x512

第十三个层 conv\_13：

* 输入的图片大小为14x14x512
* 有512个大小为3×3的卷积核，步长为1，padding为1
* 输出大小为14x14x512
* 池化层采用2×2的最大池化层，步长为2，池化后的尺寸变为7x7x512

第十四个层 fc\_14:

* 输入tensor为25088个元素的一维向量
* 4096个节点的全连接层

第十五个层 fc\_15:

* 输入tensor为4096个元素的一维向量
* 4096个节点的全连接层

第十六个层 fc\_16:

* 输入tensor为4096个元素的一维向量
* 1000个节点的全连接层，最后再添加一层softmax，输出为1000个种类的各类概率值。

从上面的过程可以看出VGG网络结构其实并不复杂，十分简洁明了，均是由小卷积核、小池化核组合而成，并且逐次增加卷积核的数量，其简化图如下：

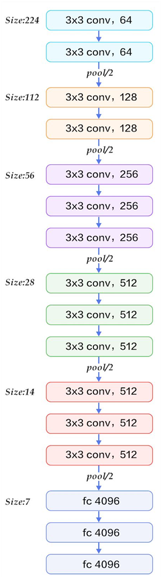


图 6.4-5 VGG16卷积层参数

本节依旧采用cifar-10数据集来验证VGG-16效果。对于VGG-16来说，网络比较深，而使用的图像像素仅有32×32，因此这里提供一个精简版本的VGG，把第八层到第十四层移除，最后的全连接层与softmax层保留。

首先读取数据集，并做归一化处理。

|  |  |
| --- | --- |
| 1 | from keras.datasets import cifar10 |
| 2 | from keras.utils import to\_categorical |
| 3 | (x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data() |
| 4 | x\_train = x\_train / 255 |
| 5 | x\_test = x\_test / 255 |
| 6 | y\_train = to\_categorical(y\_train, num\_classes=10) |
| 7 | y\_test = to\_categorical(y\_test, num\_classes=10) |

模型搭建阶段上文已经提过，这里不再赘述，具体代码如下。

|  |  |
| --- | --- |
| 1 | from keras.models import Sequential |
| 2 | from keras.layers import Dense, Flatten, Dropout |
| 3 | from keras.layers.convolutional import Conv2D, MaxPooling2D |
| 4 | from keras.optimizers import SGD |
| 5 | mport numpy as np |
| 6 | import keras |
| 7 |  |
| 8 | model = Sequential() |
| 9 | model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3), padding='same')) |
| 10 | model.add(Conv2D(32, (3, 3), activation='relu', padding='same')) |
| 11 | model.add(MaxPooling2D(pool\_size=(2, 2))) |
| 12 | model.add(Dropout(0.25)) |
| 13 | model.add(Conv2D(64, (3, 3), activation='relu', padding='same')) |
| 14 | model.add(Conv2D(64, (3, 3), activation='relu', padding='same')) |
| 15 | model.add(MaxPooling2D(pool\_size=(2, 2))) |
| 16 | odel.add(Dropout(0.25)) |
| 17 |  |
| 18 | model.add(Conv2D(128,(3, 3), activation='relu', padding='same')) |
| 19 | model.add(Conv2D(128,(3, 3), activation='relu', padding='same')) |
| 20 | model.add(Conv2D(128,(3, 3), activation='relu', padding='same')) |
| 21 | model.add(MaxPooling2D(pool\_size=(2, 2))) |
| 22 | model.add(Dropout(0.25)) |
| 23 |  |
| 24 | model.add(Flatten()) |
| 25 | model.add(Dense(512, activation='relu')) |
| 26 | model.add(Dropout(0.5)) |
| 27 | model.add(Dense(256, activation='relu')) |
| 28 | model.add(Dropout(0.5)) |
| 29 | model.add(Dense(10, activation='softmax')) |

定义损失函数、优化方法、评估指标，并引入EarlyStopping。

|  |  |
| --- | --- |
| 1 | from keras.callbacks import EarlyStopping |
| 2 | model.compile(loss='categorical\_crossentropy',  optimizer='adam',  metrics=['accuracy']) |
| 3 | early\_stopping=EarlyStopping(monitor='acc',patience=5,verbose=1) |
| 4 | model.fit(x\_train,y\_train,batch\_size=64,epochs=200, callbacks=[early\_stopping]) |

训练阶段我们计划是训练200轮，由于EarlyStopping的关系，在第55轮的时候就停止了，最后在训练集的准确率在81.96%，测试集的准确率在79.28%，当然了，读者也可以自行对上文的VGG参数进行调整来提高模型的最终效果。

|  |  |
| --- | --- |
| 1 | score = model.evaluate(x\_test, y\_test, batch\_size=64) |
| 2 | print('Test accuracy:', score[1]) |
| 3 |  |
| 4 | out: |
| 5 | Epoch 1/200  50000/50000 [==============================] - 19s 378us/step - loss: 1.8772 - acc: 0.2659 |
| 6 | Epoch 2/200  50000/50000 [==============================] - 17s 349us/step - loss: 1.4498 - acc: 0.4635 |
| 7 | …… |
| 8 | Epoch 54/200  50000/50000 [==============================] - 17s 346us/step - loss: 0.5502 - acc: 0.8173 |
| 9 | Epoch 55/200  50000/50000 [==============================] - 17s 346us/step - loss: 0.5462 - acc: 0.8196 |
| 10 | Epoch 00055: early stopping  10000/10000 [==============================] - 2s 169us/step |
| 11 | Test accuracy: 0.7928 |

到此VGG-16就搭建完成了，VGG系列的网络相比AlexNet进步了很多，且有以下优点：

* 使用了更小的卷积核和更深的网络，两个3×3卷积核的堆叠相对于5×5卷积核的视野，不但参数的个数减少了，也拥有了更多的线性变换，提高了提取图像特征的能力。
* 训练时，先训练较浅的一个网络，然后再用该较浅的网络的参数来初始化化复杂的网络的权重，从而加快模型的收敛。
* 采用了数据增强的方法来增加训练数据，该方法能有效防止模型过拟合，并提升准确率。

因此VGG也成了卷积神经网络的代表，直到目前用的也十分广泛。

**4. Keras实现ResNet**

ResNet（Residual Neural Network）是由微软何凯明等4人华人提出，该网络在ILSVRC 2015年的比赛中取得了冠军，其参数比VGGNet少，但是效果却非常突出。随着计算性能的提升，目前神经网络变得越来越复杂，从几层到几十层甚至一百多层的网络都有。深层网络的主要的优势是可以表达非常复杂的函数，它可以从不同抽象程度的层里学到特征，比如在比较低层次里学习到边缘特征而在较高层里可以学到复杂的特征。然而使用深层网络并非总是奏效，一方面可能会出现Degradation的问题，即随着准确率不断上升达到饱和后，再增加层数可能会导致准确率下降，其次训练过程困难，收敛速度慢。下图左边是普通的网络，右边是残差网络，较细的线代表验证误差，较粗的线则代表训练误差。可以看到普通的网络存在梯度退化的现象，即34层网络的训练和验证误差都大于18层的网络，而残差网络中则不存在这个现象，可见残差网络解决了梯度退化的问题。那么ResNet到底是怎么解决这个问题的呢？在下文中，将会为读者详细介绍ResNet的神奇之处。

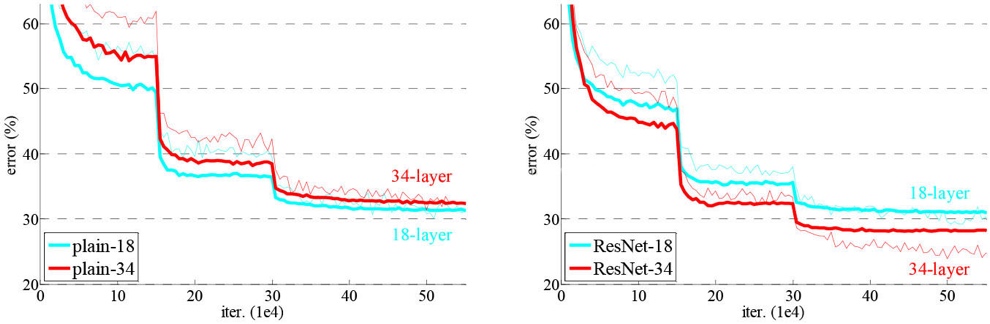


图 6.4-6 ResNet与普通卷积神经网络对比

ResNet最重要的是引入了残差学习模块，正是因为这个模块的引入才使得更深层网络成为可能。假定某神经网络的输入是，输出是，正常情况下模型需要学习的是一个到的映射关系，但是当网络较深时，输入到输出的变化可能极小，最终学习到的参数本身肯定也是较小的，如果直接把输入传到输出作为初始结果，那么此时模型需要学习的目标就是，ResNet相当于将学习目标改变了，不再是学习一个完整的输出，只是学习输出和输入的差别，并且没有增加网络的运算复杂度。传统的学习过程与ResNet提出的残差学习过程表达的效果虽然相同，但是优化的难度却并不相同，这一想法也是源于图像处理中的残差向量编码，通过一个reformulation，将一个问题分解成多个尺度直接的残差问题，能够很好的起到优化训练的效果。如图3-15所示，这是一个ResNet的残差学习单元，即残差。

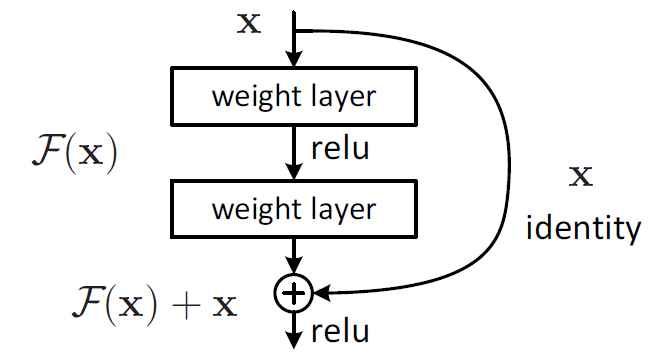


图 6.4-7 ResNet残差学习模块

传统的卷积层在信息传递的过程中，难免会出现信息的丢失，ResNet把输入直接放到了输出，保留了原有信息的完整性，一定程度上也缓解了这个问题，而学习的内容也只是输入与输出的差别的那一个部分，简化了学习的目标和难度从而提升了学习效率。

在ResNet中，除了上文提到的两层的残差学习模块，论文的作者还提出了一种三层结构的残差学习模块，如图3-16所示，其中两层的残差模块由于都是3×3的卷积核，所以没法对图像做维度上面的变化，而三层的残差模块，第一层与第三层的卷积核都是1×1的，方便对图像数据进行降维与升维，另外，对于输入与输出维度不同的情况，也可以对做一个线性映射来变换维度。

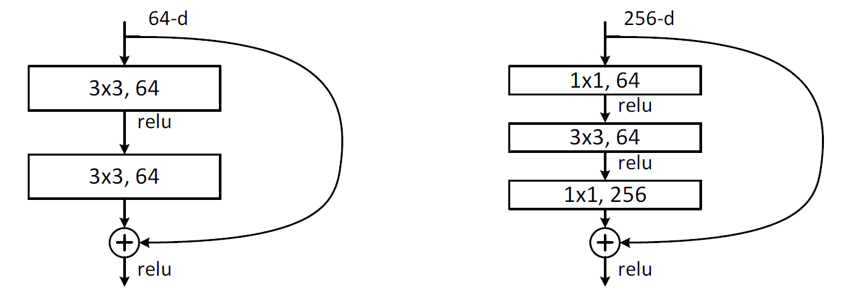


图 6.4-8 ResNet残差学习模块结构

简单看一下ResNet的结构，其整体结构图示如下，其中包括了ResNet-18、ResNet-34、 ResNet-50，ResNet-101，ResNet-152的结构。

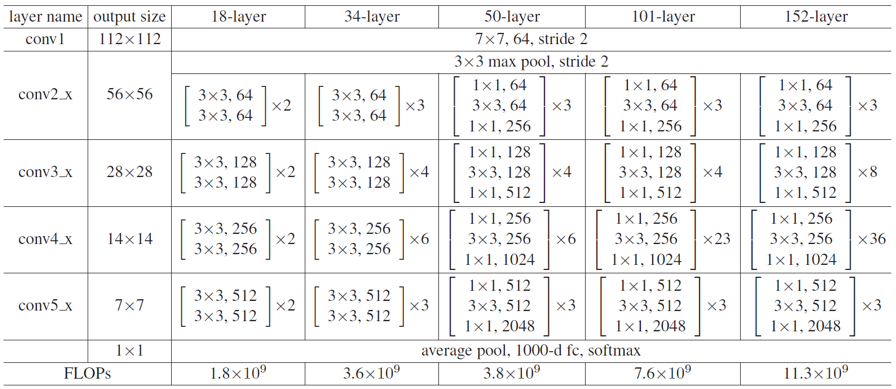


图 6.4-9 ResNet结构

下文将会基于18层的ResNet进行讲解与编码，先详细看下其内部结构。首先，作者把18层网络又分为了5大层与结尾的一个池化层加卷积层，其中conv1包括一个卷积层与一个最大池化层，conv2\_x到conv5\_x均是2个2层的残差模块堆叠在一起，一共16层，加上平均池化层、sofxmax层一共18层。代码部分将会继续采用上文提到的cifar-10数据来训练模型，重复的内容不再赘述，不清楚的读者请参阅上文。

读取数据的步骤和上文相同。

|  |  |
| --- | --- |
| 1 | from keras.datasets import cifar10 |
| 2 | from keras.utils import to\_categorical |
| 3 | (x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data() |
| 4 | x\_train = x\_train / 255 |
| 5 | x\_test = x\_test / 255 |
| 6 | y\_train = to\_categorical(y\_train, num\_classes=10) |
| 7 | y\_test = to\_categorical(y\_test, num\_classes=10) |

首先定义第一层即conv1，Functional模型需要根据输入数据定义一个Input，这里用到的图片大小是32×32×3，所以参数shape对应的值是32,32,3，接下来需要添加一个ZeroPadding2D层，正如其名，该层能给图片在边缘补零，因为使用到额图片较小，这里添加一个3×3的ZeroPadding2D层，之后就是比较熟悉的卷积层了，卷积层之后又添加了BatchNormalization层，BatchNormalization能加速收敛，控制过拟合并允许使用较大的学习率，参数axis指需要规范化的轴，通常为特征轴，最后再加上激活函数与最大池化层。

|  |  |
| --- | --- |
| 1 | import numpy as np |
| 2 | from keras.layers import Dense, Flatten |
| 3 | from keras.layers import Conv2D, MaxPooling2D, add, Input, ZeroPadding2D, BatchNormalization, Activation, AveragePooling2D |
| 4 |  |
| 5 | input\_data = Input(shape=(32,32,3)) |
| 6 | x = ZeroPadding2D((3, 3))(input\_data) |
| 7 | x = Conv2D(64, (2, 2), strides=(1, 1))(x) |
| 8 | x = BatchNormalization(axis=3)(x) |
| 9 | x = Activation('relu')(x) |
| 10 | x = MaxPooling2D((3, 3), strides=(2, 2))(x) |

接下来定义残缺模块，下文将会把残缺模块封装成一个方法方便后面调用。注意，此处在把卷积之后的结果和原数据进行相加前，给原数据添加了卷积层，可以看到，其卷积核的大小是1×1，其目的主要是为了改变原数据的维度，从而能让数据能够进行相加的操作。

|  |  |
| --- | --- |
| 1 | def identity\_Block(input\_data, n\_filter, kernel\_size=(3, 3), strides=(1, 1)): |
| 2 | x = Conv2D(n\_filter, kernel\_size,padding='same',  strides=strides)(input\_data) |
| 3 | x = BatchNormalization(axis=3)(x) |
| 4 | x = Activation('relu')(x) |
| 5 |  |
| 6 | x=Conv2D(n\_filter,kernel\_size,padding='same',  strides=strides)(x) |
| 7 | x = BatchNormalization(axis=3)(x) |
| 8 | x = Activation('relu')(x) |
| 9 |  |
| 10 | shortcut=Conv2D(n\_filter,kernel\_size=(1,1), padding='same')(input\_data) |
| 11 | shortcut = BatchNormalization(axis=3)(shortcut) |
| 12 | x = add([x, shortcut]) |
| 13 | x = Activation('relu')(x) |
| 14 | return x |

定义conv2\_x到conv5\_x的卷积层

|  |  |
| --- | --- |
| 1 | dx = identity\_Block(x, n\_filter=64) |
| 2 | x = identity\_Block(x, n\_filter=64) |
| 3 |  |
| 4 | x = identity\_Block(x, n\_filter=128) |
| 5 | x = identity\_Block(x, n\_filter=128) |
| 6 |  |
| 7 | x = identity\_Block(x, n\_filter=256) |
| 8 | x = identity\_Block(x, n\_filter=256) |
| 9 |  |
| 10 | x = identity\_Block(x, n\_filter=512) |
| 11 | x = identity\_Block(x, n\_filter=512) |

添加平均池化层与sofxmax层。

|  |  |
| --- | --- |
| 1 | from keras.models import Model |
| 2 | x = AveragePooling2D(pool\_size=(3, 3))(x) |
| 3 | x = Flatten()(x) |
| 4 | x = Dense(10, activation='softmax')(x) |
| 5 |  |
| 6 | model = Model(inputs=input\_data, outputs=x) |

添加early\_stopping并进行数据训练与测试。

|  |  |
| --- | --- |
| 1 | from keras.callbacks import EarlyStopping |
| 2 | model.compile(loss='categorical\_crossentropy',  optimizer='adam',  metrics=['accuracy']) |
| 3 | early\_stopping=EarlyStopping(monitor='acc',patience=3,verbose=1) |
| 4 | model.fit(x\_train,y\_train,batch\_size=64,epochs=200, callbacks=[early\_stopping]) |
| 5 |  |
| 6 | out: |
| 7 | Epoch 1/200  50000/50000 [==============================] - 342s 7ms/step–  loss: 2.3734 - acc: 0.2878 |
| 8 | Epoch 2/200  50000/50000 [==============================] - 329s 7ms/step –  loss: 1.9178 - acc: 0.3842 |
| 9 | …… |
| 10 | Epoch 39/200  50000/50000 [==============================] - 332s 7ms/step –  loss: 0.0253 - acc: 0.9913 |
| 11 | Epoch 40/200  50000/50000 [==============================] - 330s 7ms/step –  loss: 0.0269 - acc: 0.9910 |
| 12 | Epoch 00040: early stopping |

|  |  |
| --- | --- |
| 1 | score = model.evaluate(x\_test, y\_test, batch\_size=64) |
| 2 | print('Test accuracy:', score[1]) |
| 3 |  |
| 4 | out: |
| 5 | 10000/10000 [==============================] - 22s 2ms/step  Test accuracy: 0.8001 |

到此就讲解完了ResNet的原理与结构，ResNet是卷积神经网络发展历史中重要的一环，其解决了神经网络较深引起的一系列问题。目前的一些热门卷积神经网络如Google的Inception Net也都受其启发，并把其残缺模块融入到了自身之中，并取得了非常突出的效果。

本章主要讲解了卷积神经网络发展史中几个重要的网络，相信读者一定对卷积神经网络有了一个全面的认识。神经网络发展迅速，可以看到，自2012年AlexNet提出后，每隔几个月就会有新的技术提出，这些新的技术改变着网络的结构，并在图像领域不停的刷新着新的记录，上文中介绍的也都是一些比较出名的网络结构，目前已经涌现了大量效果极佳且参数量较少的卷积神经网络，Keras中也实现了许多经典的卷积神经网络，这些网络在各大比赛中都取得了优异的成绩，读者可借鉴这些网络的优点来搭建自己的神经网络，当然也可以采用迁移学习的方法直接使用Keras预先训练好的模型来完成自己的图像识别任务。在第七章，也会带领读者学习使用迁移学习，并完成几个有意思的图像识别项目。