人工智能实践：Tensorflow 笔记

第五讲

卷积神经网络

本节主要内容：讲解卷积神经网络，利用基础 CNN、LeNet、AlexNet、VGGNet、InceptionNet 和 ResNet 实现图像识别。

5.1 全连接网络回顾

√ 全连接 NN 特点：每个神经元与前后相邻层的每一个神经元都有连接关系。（可以实现分类和预测）

全连接网络的参数个数为：$\sum$（前层×后层＋后层）

如图 5-1 所示，针对一张分辨率仅为 28 * 28 的黑白图像（像素值个数为 28 * 28 * 1 = 784），全连接网络的参数总量就有将近 40 万个。

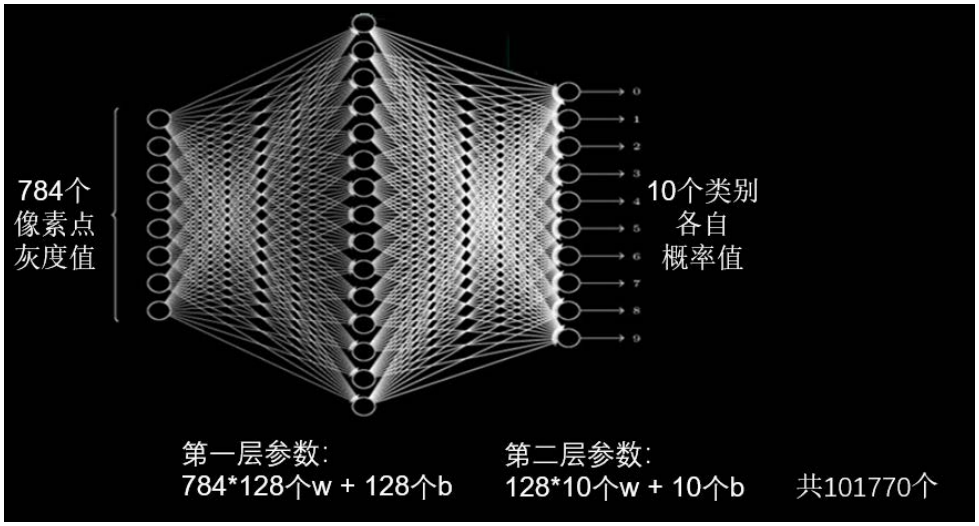


图 5-1 全连接网络的参数量

在实际应用中，图像的分辨率远高于此，且大多数是彩色图像，如图 5-2 所示。虽然全连接网络一般被认为是分类预测的最佳网络，但待优化的参数过多，容易导致模型过拟合。

灰度图单通道 **彩色图红绿蓝三通道**

图 5-2 灰度图与彩色图

为了解决参数量过大而导致模型过拟合的问题，一般不会将原始图像直接输入，而是先对图像进行特征提取，再将提取到的特征输入全连接网络，如图 5-3 所示，就是将汽车图片

经过多次特征提取后再喂入全连接网络。

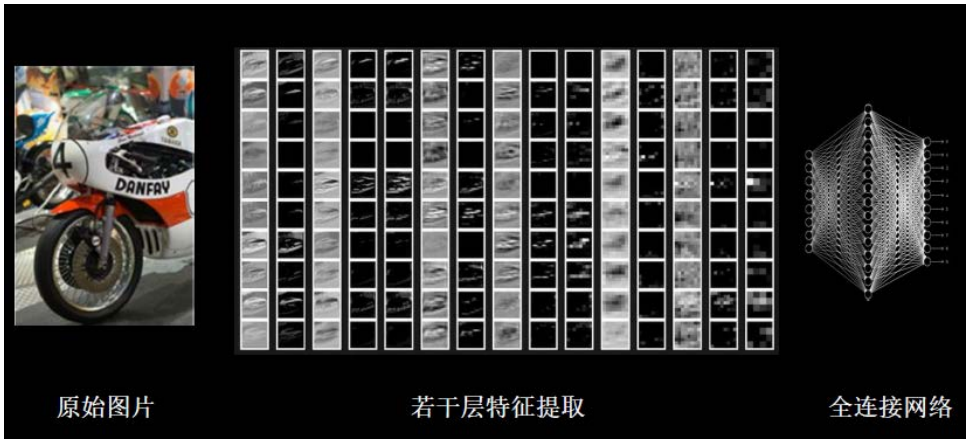


图 5-3 全连接网络的改进

5.2 卷积神经网络

√ 卷积的概念：卷积可以认为是一种有效提取图像特征的方法。一般会用一个正方形的卷积核，按指定步长，在输入特征图上滑动，遍历输入特征图中的每个像素点。每一个步长，卷积核会与输入特征图出现重合区域，重合区域对应元素相乘、求和再加上偏置项得到输出特征的一个像素点，如图 5-4 所示，利用大小为 3×3×1 的卷积核对 5×5×1 的单通道图像做卷积计算得到相应结果。

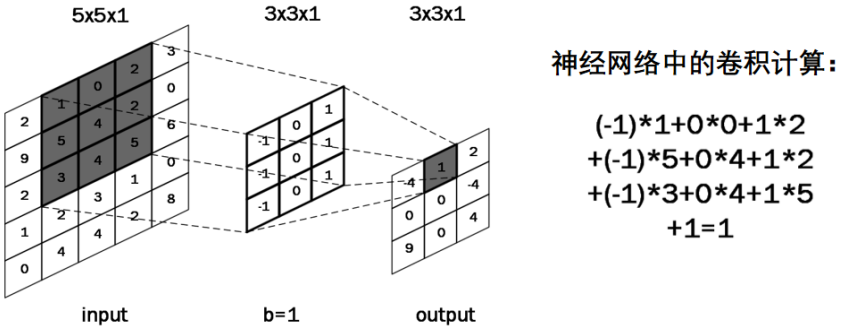


图 5-4 神经网络中的卷积计算

√ 对于彩色图像（多通道）来说，卷积核通道数与输入特征一致，套接后在对应位置上进行乘加和操作，如图 5-5 所示，利用三通道卷积核对三通道的彩色特征图做卷积计算。

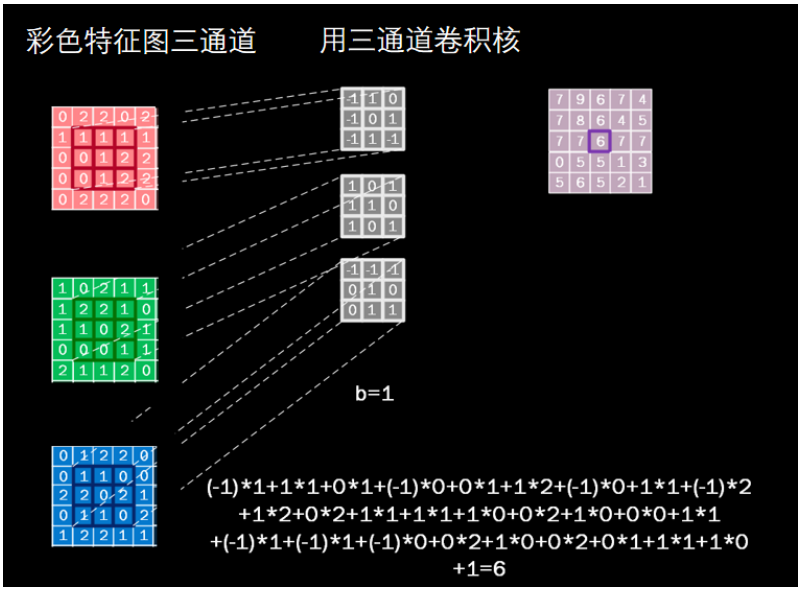


图 5-5 三通道彩色图像的卷积计算

√ 用多个卷积核可实现对同一层输入特征的多次特征提取，卷积核的个数决定输出层的通道（channels）数，即输出特征图的深度。

√ 感受野（Receptive Field）的概念：卷积神经网络各输出层每个像素点在原始图像上的映射区域大小，如图 5-7 所示。

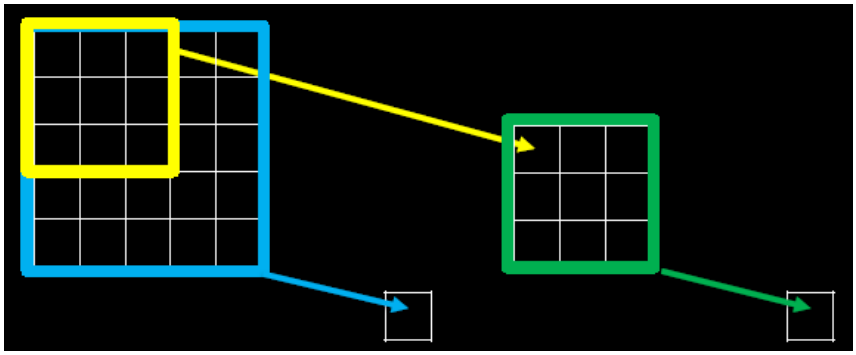


图 5-6 感受野示意图

当我们采用尺寸不同的卷积核时，最大的区别就是感受野的大小不同，所以经常会采用多层小卷积核来替换一层大卷积核，在保持感受野相同的情况下减少参数量和计算量，例如十分常见的用 2 层 3 * 3 卷积核来替换 1 层 5 * 5 卷积核的方法，如图 5-7 所示。

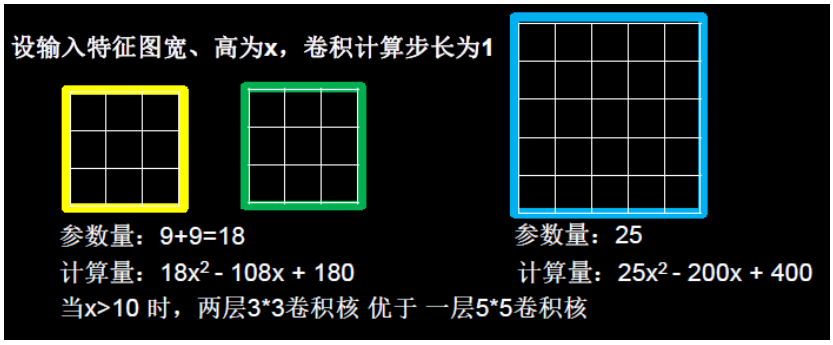


图 5-7 两层 3*3 卷积核与一层 5*5 卷积核的对比

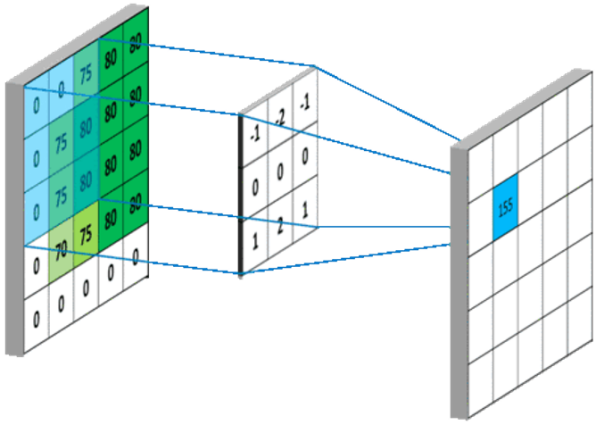
这里给出详细推导：不妨设输入特征图的宽、高均为 x，卷积计算的步长为 1，显然，两个 3 * 3 卷积核的参数量为 9 + 9 = 18，小于 5 * 5 卷积核的 25，前者的参数量更少。

在计算量上，根据图 5-8 所示的输出特征尺寸计算公式，对于 5 * 5 卷积核来说，输出特征图共有(x – 5 + 1)^2 个像素点，每个像素点需要进行 5 * 5 = 25 次乘加运算，则总计算量为 25 * (x – 5 + 1)^2 = 25x^2 – 200x + 400；

对于两个 3 * 3 卷积核来说，第一个 3 * 3 卷积核输出特征图共有(x – 3 + 1)^2 个像素点，每个像素点需要进行 3 * 3 = 9 次乘加运算，第二个 3 * 3 卷积核输出特征图共有(x – 3 + 1 – 3 + 1)^2 个像素点，每个像素点同样需要进行 9 次乘加运算，则总计算量为 9 * (x – 3 + 1)^2 + 9 * (x – 3 + 1 – 3 + 1)^2 = 18 x^2 – 108x + 180；

对二者的总计算量（乘加运算的次数）进行对比，18 x^2 – 200x + 400 < 25x^2 – 200x + 400，经过简单数学运算可得 x < 22/7 or x > 10，x 作为特征图的边长，在大多数情况下显然会是一个大于 10 的值（非常简单的 MNIST 数据集的尺寸也达到了 28 * 28），所以两层 3 * 3 卷积核的参数量和计算量，在通常情况下都优于一层 5 * 5 卷积核，尤其是当特征图尺寸比较大的情况下，两层 3 * 3 卷积核在计算量上的优势会更加明显。

√ 输出特征尺寸计算：在了解神经网络中卷积计算的整个过程后，就可以对输出特征图的尺寸进行计算，如图 5-8 所示，5×5 的图像经过 3×3 大小的卷积核做卷积计算后输出特征尺寸为 3×3。



输出图片边长=（输入图片边长 – 卷积核长 + 1）/ 步长　向上取整
此图：　（5 – 3 + 1）/ 1 = 3

图 5-8 输出特征尺寸计算

√ 全零填充（padding）：为了保持输出图像尺寸与输入图像一致，经常会在输入图像周围进行全零填充，如图 5-9 所示，在 5×5 的输入图像周围填 0，则输出特征尺寸同为 5×5。

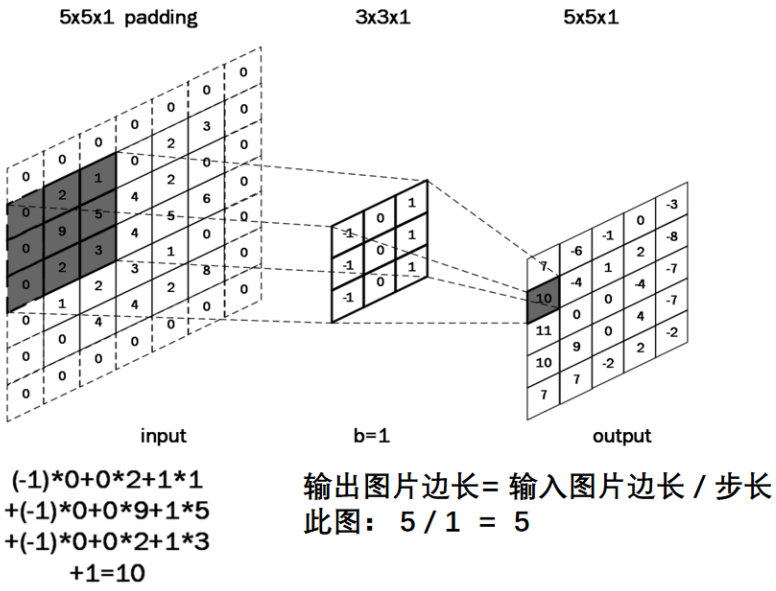


图 5-9 全零填充的输出特征尺寸计算

在 Tensorflow 框架中，用参数 padding = ‘SAME’或 padding = ‘VALID’表示是否进行全零填充，其对输出特征尺寸大小的影响如下：

$$padding\begin{cases} SAME & \frac{入长}{步长}\ （面积不变） \\ VALID（不全零填充） & \frac{入长-核长+1}{步长}\ （向上取整） \end{cases}$$

上下两行分别代表对输入图像进行全零填充或不进行填充，对于 5×5×1 的图像来说，当 padding = ‘SAME’时，输出图像边长为 5；当 padding = ‘VALID’时，输出图像边长为 3。

具备以上知识后，就可以在 Tensorflow 框架下利用 Keras 来构建 CNN 中的卷积层，使用的是 tf.keras.layers.Conv2D 函数，具体的使用方法如下：

```
tf.keras.layers.Conv2D(
input_shape = (高，宽，通道数)，  #仅在第一层有
filters =  卷积核个数,
kernel_size =  卷积核尺寸,
strides =  卷积步长,
padding = ‘SAME’ or ‘VALID’,
activation = ‘relu’ or ‘sigmoid’ or ‘tanh’ or ‘softmax’等   #如有 BN 则此处不用写
)
```

使用此函数构建卷积层时，需要给出的信息有：

A）输入图像的信息，即宽高和通道数；

B）卷积核的个数以及尺寸，如 filters = 16, kernel_size = (3, 3)代表采用 16 个大小为 3×3 的卷积核；

C）卷积步长，即卷积核在输入图像上滑动的步长，纵向步长与横向步长通常是相同的，默认值为 1；

D）是否进行全零填充，全零填充的具体作用上文有描述；

E）采用哪种激活函数，例如 relu、softmax 等，各种函数的具体效果在前面章节中有详细描

述；

这里需要注意的是，在利用 Tensorflow 框架构建卷积网络时，一般会利用 BatchNormalization 函数来构建 BN 层，进行批归一化操作，所以在 Conv2D 函数中经常不写 BN。BN 操作的具体含义和作用见下文。

√ Batch Normalization(批标准化)：对一小批数据在网络各层的输出做标准化处理，其具体实现方式如图 5-10 所示。（标准化：使数据符合 0 均值，1 为标准差的分布。）

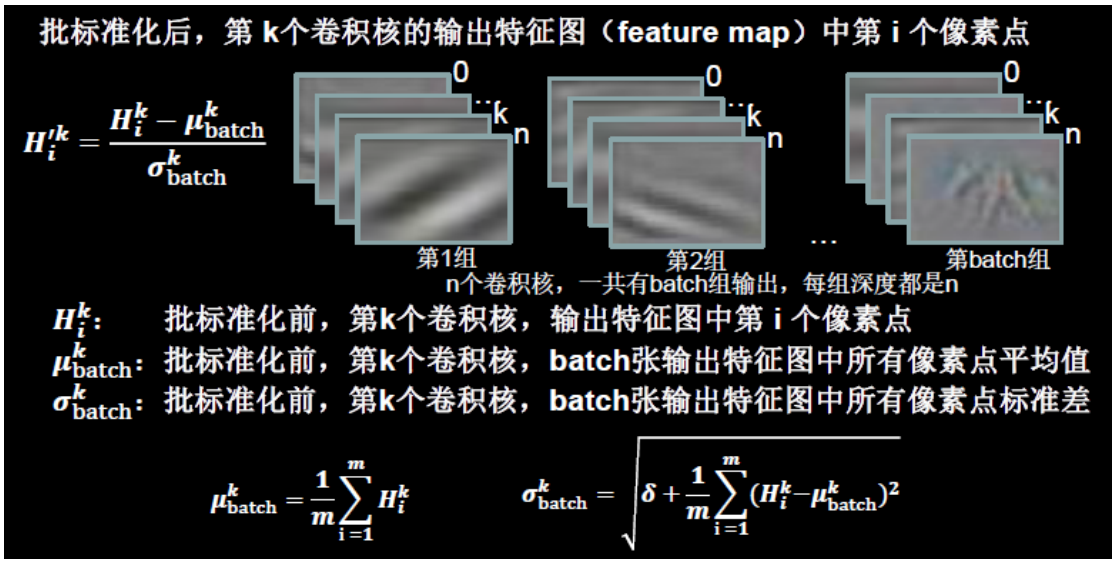


图 5-10 Batch Normalization 的实现

Batch Normalization 将神经网络每层的输入都调整到均值为 0，方差为 1 的标准正态分布，其目的是解决神经网络中梯度消失的问题，如图 5-11 所示。

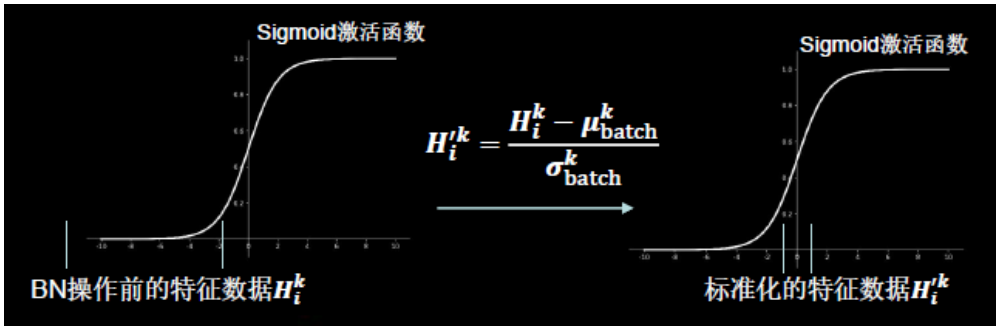


图 5-11 Batch Normalization 的作用（以 Sigmoid 激活函数为例）

BN 操作的另一个重要步骤是缩放和偏移，值得注意的是，缩放因子 γ 以及偏移因子 β 都是可训练参数，其作用如图 5-12 所示。

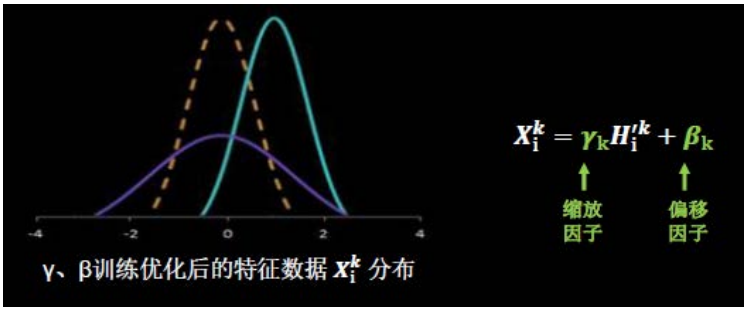


图 5-12 BN 中的缩放与平移

BN 操作通常位于卷积层之后，激活层之前，在 Tensorflow 框架中，通常使用 Keras 中

的 tf.keras.layers.BatchNormalization 函数来构建 BN 层。

在调用此函数时，需要注意的一个参数是 training，此参数只在调用时指定，在模型进行前向推理时产生作用，当 training = True 时，BN 操作采用当前 batch 的均值和标准差；当 training = False 时，BN 操作采用滑动平均（running）的均值和标准差。在 Tensorflow 中，通常会指定 training = False，可以更好地反映模型在测试集上的真实效果。

滑动平均（running）的解释：滑动平均，即通过一个个 batch 历史的叠加，最终趋向数据集整体分布的过程，在测试集上进行推理时，滑动平均的参数也就是最终保存的参数。

此外，Tensorflow 中的 BN 函数其实还有很多参数，其中比较常用的是 momentum，即动量参数，与 sgd 优化器中的动量参数含义类似但略有区别，具体作用为滑动平均 running = momentum * running + (1 – momentum) * batch，一般设置一个比较大的值，在 Tensorflow 框架中默认为 0.99。

√ 池化（pooling）：池化的作用是减少特征数量（降维）。最大值池化可提取图片纹理，均值池化可保留背景特征，如图 5-13 所示。

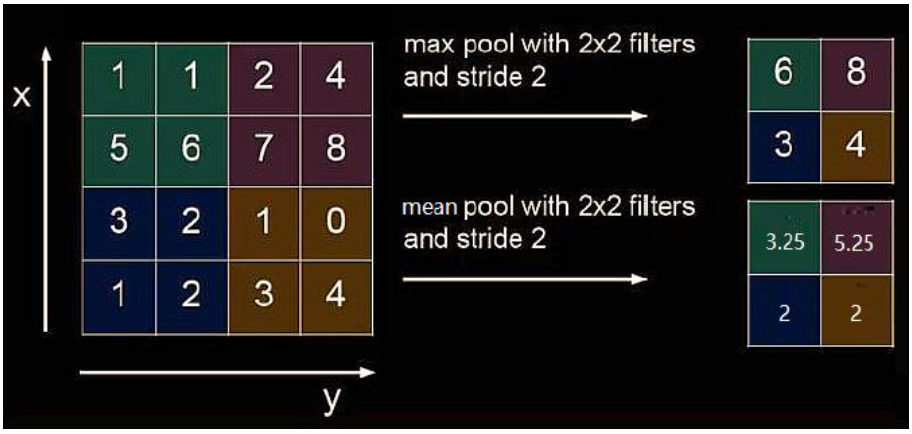


图 5-13 最大值池化与均值池化

在 Tensorflow 框架下，可以利用 Keras 来构建池化层，使用的是 tf.keras.layers.MaxPool2D 函数和 tf.keras.layers.AveragePooling2D 函数，具体的使用方法如下：

```
tf.keras.layers.MaxPool2D(
pool_size = 池化核尺寸,
strides = 池化步长,
padding = ‘SAME’ or ‘VALID’
)
tf.keras.layers.AveragePooling2D(
pool_size = 池化核尺寸,
strides = 池化步长,
padding = ‘SAME’ or ‘VALID’
)
```

√ 舍弃（Dropout）：在神经网络的训练过程中，将一部分神经元按照一定概率从神经网络中暂时舍弃，使用时被舍弃的神经元恢复链接，如图 5-14 所示。

图 5-14 舍弃（Dropout）示意图

在 Tensorflow 框架下，利用 tf.keras.layers.Dropout 函数构建 Dropout 层，参数为舍弃的概率（大于 0 小于 1）。

利用上述知识，就可以构建出基本的卷积神经网络（CNN）了，其核心思路为在 CNN 中利用卷积核（kernel）提取特征后，送入全连接网络。

√CNN 模型的主要模块：一般包括上述的卷积层、BN 层、激活函数、池化层以及全连接层，如图 5-15 所示。

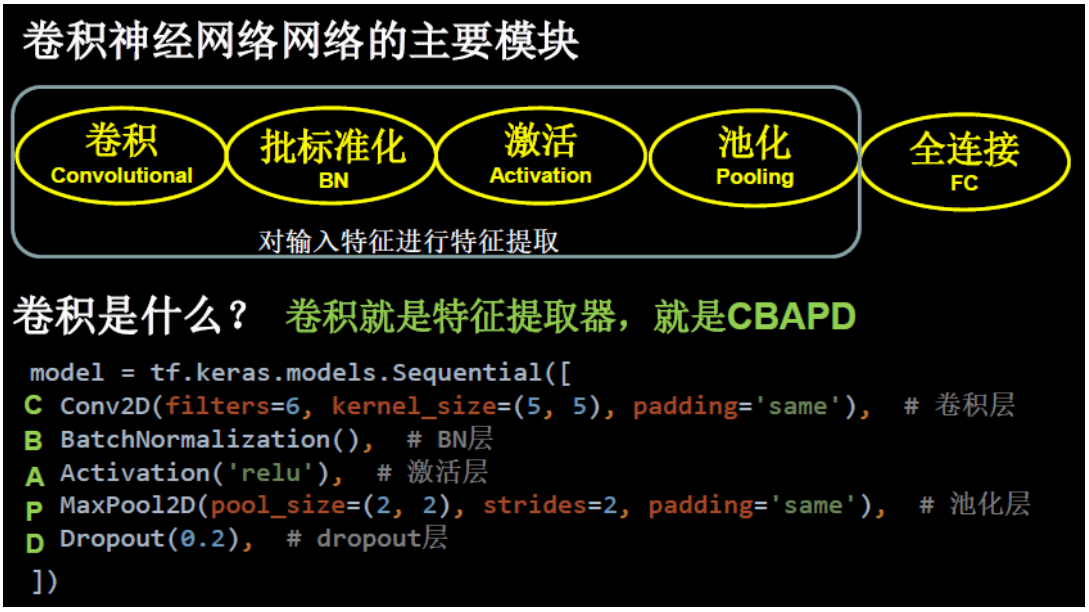


图 5-15 CNN 模型的主要模块

在此基础上，可以总结出在 Tensorflow 框架下，利用 Keras 来搭建神经网络的“八股”套路，在主干的基础上，还可以添加其他内容，来完善神经网络的功能，如利用自己的图片和标签文件来自制数据集；通过旋转、缩放、平移等操作对数据集进行数据增强；保存模型文件进行断点续训；提取训练后得到的模型参数以及准确率曲线，实现可视化等。

构建神经网络的“八股”套路：

A）import 引入 tensorflow 及 keras、numpy 等所需模块。

B）读取数据集，课程中所利用的 MNIST、cifar10 等数据集比较基础，可以直接从 sklearn 等模块中引入，但是在实际应用中，大多需要从图片和标签文件中读取所需的数据集。

C）搭建所需的网络结构，当网络结构比较简单时，可以利用 keras 模块中的 tf.keras.Sequential 来搭建顺序网络模型；但是当网络不再是简单的顺序结构，而是有其它特殊结构出现时（例如 ResNet 中的跳连结构），便需要利用 class 来定义自己的网络结构。前者使用起来更加方便，但实际应用中往往需要利用后者来搭建网络。

D）对搭建好的网络进行编译（compile），通常在这一步指定所采用的优化器（如 Adam、sgd、RMSdrop 等）以及损失函数（如交叉熵函数、均方差函数等），选择哪种优化器和损

失函数往往对训练的速度和效果有很大的影响，至于具体如何进行选择，前面的章节中有比较详细的介绍。

E）将数据输入编译好的网络来进行训练（model.fit），在这一步中指定训练轮数 epochs 以及 batch_size 等信息，由于神经网络的参数量和计算量一般都比较大，训练所需的时间也会比较长，尤其是在硬件条件受限的情况下，所以在这一步中通常会加入断点续训以及模型参数保存等功能，使训练更加方便，同时防止程序意外停止导致数据丢失的情况发生。

F）将神经网络模型的具体信息打印出来（model.summary），包括网络结构、网络各层的参数等，便于对网络进行浏览和检查。

十分类彩色图片数据集 cifar10 数据集介绍：该数据集共有 60000 张彩色图像，每张尺寸为 32 * 32，分为 10 类，每类 6000 张。训练集 50000 张，分为 5 个训练批，每批 10000 张；从每一类随机取 1000 张构成测试集，共 10000 张，剩下的随机排列组成训练集，如图 5-16 所示。

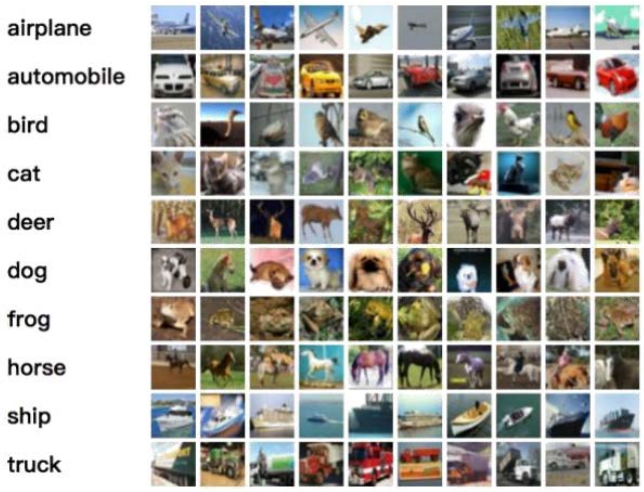


图 5-16 cifar10 数据集

cifar10 数据集的读取：

√ 数据集下载：

cifar10 = tf.keras.datasets.cifar10

√ 导入训练集和测试集：

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

√ 打印训练集与测试集的数据维度，打印结果为：

```
x_train.shape: (50000, 32, 32, 3)
x_test.shape: (10000, 32, 32, 3)
y_train.shape: (50000, 1)
y_test.shape: (10000, 1)
```

显然，cifar10 是一个用于图像分类的数据集，共分 10 类，相较于 mnist 数据集会更复杂一些，训练难度也更大，但是图像尺寸较小，仅为 32 * 32，仍然属于比较基础的数据集，利用一些 CNN 经典网络结构（如 VGGNet、ResNet 等，下一小节会具体介绍）进行训练的话准确率很容易就能超过 90%，很适合初学者用来练习。目前学术界对于 cifar10 数据集的分类准确率已经达到了相当高的水准，图 5-17 中为 Github 网站上 cifar10 数据集分类准确率的排行榜。

参考网址：http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

Result	Method	Venue
96.53%	Fractional Max-Pooling	arXiv 2015
95.59%	Striving for Simplicity: The All Convolutional Net	ICLR 2015
94.16%	All you need is a good init	ICLR 2016
94%	Lessons learned from manually classifying CIFAR-10	unpublished 2011
93.95%	Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree	AISTATS 2016
93.72%	Spatially-sparse convolutional neural networks	arXiv 2014
93.63%	Scalable Bayesian Optimization Using Deep Neural Networks	ICML 2015
93.57%	Deep Residual Learning for Image Recognition	arXiv 2015
93.45%	Fast and Accurate Deep Network Learning by Exponential Linear Units	arXiv 2015
93.34%	Universum Prescription: Regularization using Unlabeled Data	arXiv 2015
93.25%	Batch-normalized Maxout Network in Network	arXiv 2015
93.13%	Competitive Multi-scale Convolution	arXiv 2015

图 5-17 cifar10 数据集分类准确率排行

Cifar10 数据集相关代码：p24_cifar10_datasets.py

掌握了利用 tf.keras 来搭建神经网络的八股之后，就可以搭建自己的神经网络来对数据集进行训练了，这里提供一个实例，利用一个结构简单的基础卷积神经网络（CNN）来对 cifar10 数据集进行训练，网络结构如图 5-18 所示。

图 5-18 网络结构

利用 tf.keras.Sequential 模型以及 class 定义两种方式都可以构建出图 5-18 中的基础 CNN 网络，在此例中二者的效果是完全相同的，前者看起来会更简洁一些，但后者在实际应用中更加常用，因为这仅仅是一个非常基础的网络，而一些复杂的网络经常会有 Sequential 模型无法表达的结构或设计，所以在这里采用后者，如图 5-19 所示。

图 5-19 卷积神经网络搭建示例

CNN 训练 cifar10 数据集的 baseline 源码：p27_cifar10_baseline.py

上述源码中，除了 CNN 网络的搭建以外，还包含 cifar10 数据集读取，模型参数保存与读取，loss 及准确率曲线绘制等功能，在之后的代码中也会经常用到。

5.3 CNN 经典网络

在卷积神经网络的发展历程中，出现过许多经典的网络结构，这些 CNN 经典网络的提出都曾极大地促进了领域的发展，这里对 5 个经典的 CNN 网络结构做一个介绍，从 1998 年由 Yann LeCun 提出的 LeNet 直至 2015 年由何恺明提出的 ResNet，如图 5-20 所示。

值得一提的是，除了卷积网络的“开篇之作”LeNet 以外，AlexNet、VGGNet、InceptionNet 以及 ResNet 这四种经典网络全部是在当年的 ImageNet 竞赛中问世的，它们作为深度学习的经典代表，使得 ImageNet 数据集上的错误率逐年降低。下面将会对这五种经典网络逐一进行介绍。

图 5-20 5 个 CNN 经典网络

附：CNN 经典网络论文出处

LeNet-5:

Yann Lecun, Leon Bottou, Y. Bengio, Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, 1998.

AlexNet:

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In NIPS, 2012.

VGG16:

K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition.In ICLR, 2015.

Inception-v1:

Szegedy C, Liu W, Jia Y, et al. Going Deeper with Convolutions. In CVPR, 2015.

ResNet:

Kaiming He, Xiangyu Zhang, Shaoqing Ren. Deep Residual Learning for Image Recognition. In

CPVR, 2016.

5.3.1 LeNet

模型实现代码：p31_cifar10_lenet5.py
借鉴点：共享卷积核，减少网络参数。

LeNet 即 LeNet5，由 Yann LeCun 在 1998 年提出，做为最早的卷积神经网络之一，是许多神经网络架构的起点，其网络结构如图 5-21 所示。

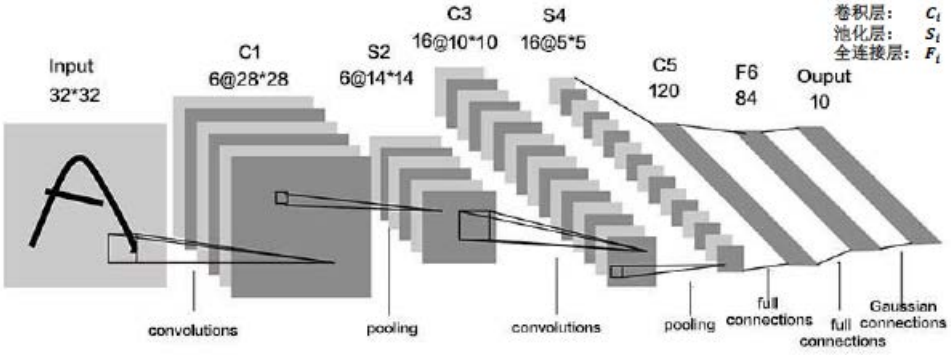


图 5-21 LeNet5 网络结构

根据以上信息，就可以根据上一节所总结出来的方法，在 Tensorflow 框架下利用 tf.Keras 来构建 LeNet5 模型，如图 5-22 所示。

图 5-22 实现 LeNet5 模型

图中紫色部分为卷积层，红色部分为全连接层，模型图与代码一一对应，模型搭建具体流程如下（各步骤的实现函数在 5.2 节中均有介绍）：

A）输入图像大小为 32 * 32 * 3，三通道彩色图像输入；

B）进行卷积，卷积核大小为 5 * 5，个数为 6，步长为 1，不进行全零填充；

C）将卷积结果输入 sigmoid 激活函数（非线性函数）进行激活；

D）进行最大池化，池化核大小为 2 * 2，步长为 2；

```
self.c1 = Conv2D(filters=6, kernel_size=(5, 5),
                 padding='valid', input_shape=(32, 32, 3), activation='sigmoid')
self.p1 = MaxPool2D(pool_size=(2, 2), strides=2)
```

E）进行卷积，卷积核大小为 5 * 5，个数为 16，步长为 1，不进行全零填充；

F）将卷积结果输入 sigmoid 激活函数进行激活；

G）进行最大池化，池化核大小为 2 * 2，步长为 2；

```
self.c2 = Conv2D(filters=16, kernel_size=(5, 5),
                 padding='valid', activation='sigmoid')
self.p2 = MaxPool2D(pool_size=(2, 2), strides=2)
```

H）输入三层全连接网络进行 10 分类。

```
self.flatten = Flatten()
self.f1 = Dense(120, activation='sigmoid')
self.f2 = Dense(84, activation='sigmoid')
self.f3 = Dense(10, activation='softmax')
```

与最初的 LeNet5 网络结构相比，这里做了一点微调，输入图像尺寸为 32 * 32 * 3，以适应 cifar10 数据集（此数据集在 5.2 节中也有具体介绍）。模型中采用的激活函数有 sigmoid 和 softmax，池化层均采用最大池化，以保留边缘特征。

总体上看，诞生于 1998 年的 LeNet5 与如今一些主流的 CNN 网络相比，其结构可以说是相当简单，不过它成功地利用“卷积提取特征→全连接分类”的经典思路解决了手写数字识别的问题，对神经网络研究的发展有着很重要的意义。

5.3.2 AlexNet

模型实现代码：p34_cifar10_alexnet8.py

借鉴点：激活函数使用 Relu，提升训练速度；Dropout 防止过拟合。

AlexNet 网络诞生于 2012 年，其 ImageNet Top5 错误率为 16.4 %，可以说 AlexNet 的出现使得已经沉寂多年的深度学习领域开启了黄金时代。

AlexNet 的总体结构和 LeNet5 有相似之处，但是有一些很重要的改进：

A）由五层卷积、三层全连接组成，输入图像尺寸为 224 * 224 * 3，网络规模远大于 LeNet5；

B）使用了 Relu 激活函数；

C）进行了舍弃（Dropout）操作，以防止模型过拟合，提升鲁棒性；

D）增加了一些训练上的技巧，包括数据增强、学习率衰减、权重衰减（L2 正则化）等。

AlexNet 的网络结构如图 5-23 所示。

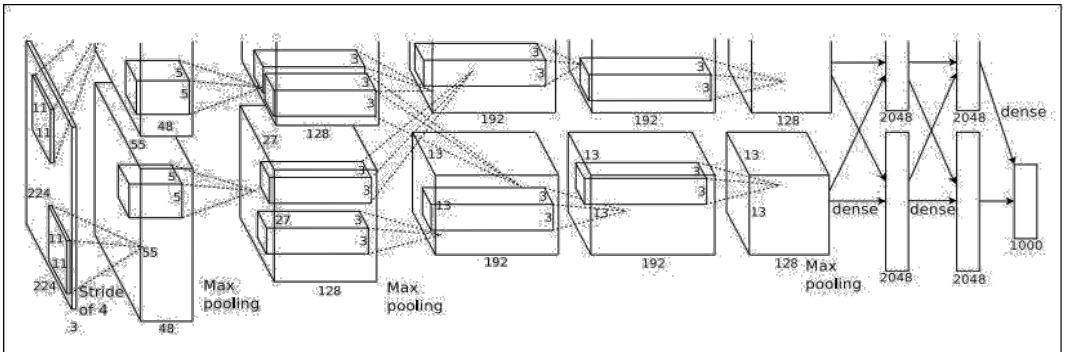


图 5-23 AlexNet 网络结构

可以看到，图 5-20 所示的网络结构将模型分成了两部分，这是由于当时用于训练 AlexNet 的显卡为 GTX 580（显存为 3GB），单块显卡运算资源不足的原因。

在 Tensorflow 框架下利用 Keras 来搭建 AlexNet 模型，这里做了一些调整，将输入图像尺寸改为 32 * 32 * 3 以适应 cifar10 数据集，并且将原始的 AlexNet 模型中的 11 * 11、7 * 7、5 * 5 等大尺寸卷积核均替换成了 3 * 3 的小卷积核，如图 5-24 所示。

图 5-24 Keras 实现 AlexNet 模型

图中紫色块代表卷积部分，可以看到卷积操作共进行了 5 次：

A）第 1 次卷积：共有 96 个 3 * 3 的卷积核，不进行全零填充，进行 BN 操作，激活函数为 Relu，进行最大池化，池化核尺寸为 3 * 3，步长为 2；

```
self.c1 = Conv2D(filters=96, kernel_size=(3, 3), input_shape=(32, 32, 3))
self.b1 = BatchNormalization()
self.a1 = Activation('relu')
self.p1 = MaxPool2D(pool_size=(3, 3), strides=2)
```

B）第 2 次卷积：与第 1 次卷积类似，除卷积核个数由 96 增加到 256 之外几乎相同；

```
self.c2 = Conv2D(filters=256, kernel_size=(3, 3))
self.b2 = BatchNormalization()
self.a2 = Activation('relu')
self.p2 = MaxPool2D(pool_size=(3, 3), strides=2)
```

C）第 3 次卷积：共有 384 个 3 * 3 的卷积核，进行全零填充，激活函数为 Relu，不进行 BN 操作以及最大池化；

```
self.c3 = Conv2D(filters=384, kernel_size=(3, 3), padding='same',
                 activation='relu')
```

D）第 4 次卷积：与第 3 次卷积几乎完全相同；

```
self.c4 = Conv2D(filters=384, kernel_size=(3, 3), strides=1, padding='same',
                 activation='relu')
```

E）第 5 次卷积：共有 96 个 3 * 3 的卷积核，进行全零填充，激活函数为 Relu，不进行 BN 操作，进行最大池化，池化核尺寸为 3 * 3，步长为 2。

```
self.c5 = Conv2D(filters=256, kernel_size=(3, 3), strides=1, padding='same',
                 activation='relu')
self.p3 = MaxPool2D(pool_size=(3, 3), strides=2)
```

图中红色块代表全连接部分，共有三层：

A）第一层共 2048 个神经元，激活函数为 Relu，进行 0.5 的 dropout；

```
self.flatten = Flatten()
self.f1 = Dense(2048, activation='relu')
self.d1 = Dropout(0.5)
```

B）第二层与第一层几乎完全相同；

```
self.f2 = Dense(2048, activation='relu')
self.d2 = Dropout(0.5)
```

C）第三层共 10 个神经元，进行 10 分类。

```
self.f3 = Dense(10, activation='softmax')
```

可以看到，与结构类似的 LeNet5 相比，AlexNet 模型的参数量有了非常明显的提升，卷积运算的层数也更多了，这有利于更好地提取特征；Relu 激活函数的使用加快了模型的训练速度；Dropout 的使用提升了模型的鲁棒性，这些优势使得 AlexNet 的性能大大提升。

5.3.3 VGGNet

模型实现代码：p36_cifar10_vgg16.py

借鉴点：小卷积核减少参数的同时，提高识别准确率；网络结构规整，适合并行加速。

在 AlexNet 之后，另一个性能提升较大的网络是诞生于 2014 年的 VGGNet，其 ImageNet Top5 错误率减小到了 7.3 %。

VGGNet 网络的最大改进是在网络的深度上，由 AlexNet 的 8 层增加到了 16 层和 19 层，更深的网络意味着更强的表达能力，这得益于强大的运算能力支持。VGGNet 的另一个显著特点是仅使用了单一尺寸的 3 * 3 卷积核，事实上，3 * 3 的小卷积核在很多卷积网络中都被大量使用，这是由于在感受野相同的情况下，小卷积核堆积的效果要优于大卷积核，同时参数量也更少。VGGNet 就使用了 3 * 3 的卷积核替代了 AlexNet 中的大卷积核（11 * 11、7 * 7、5 * 5），取得了较好的效果（事实上课程中利用 Keras 实现 AlexNet 时已经采取了这种方式），VGGNet16 的网络结构如图 5-25 所示。

VGGNet16 和 VGGNet19 并没有本质上的区别，只是网络深度不同，前者 16 层（13 层卷积、3 层全连接），后者 19 层（16 层卷积、3 层全连接）。

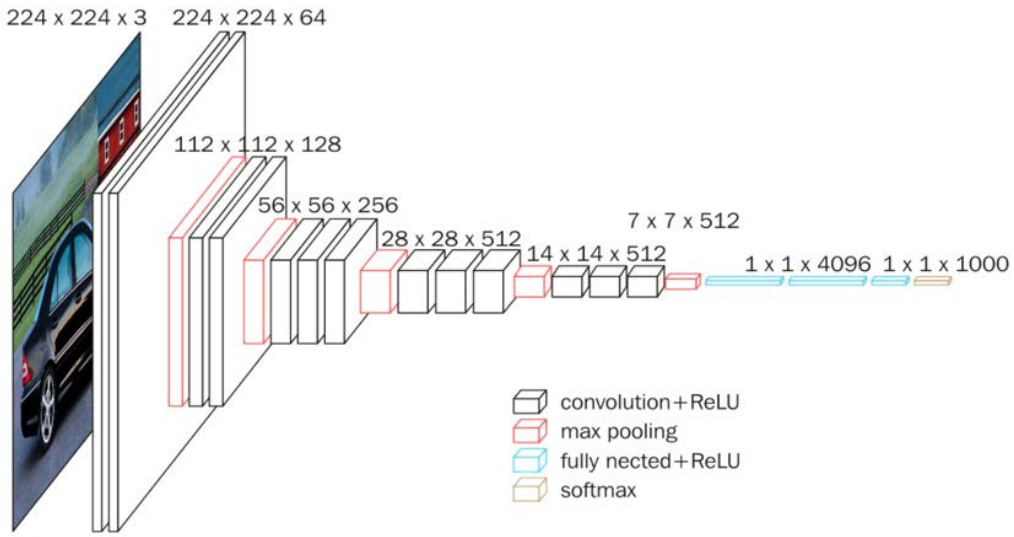


图 5-25 VGGNet16 网络结构图

在 Tensorflow 框架下利用 Keras 来实现 VGG16 网络，为适应 cifar10 数据集，将输入图像尺寸由 224 * 244 * 3 调整为 32 * 32 * 3，如图 5-26 所示。

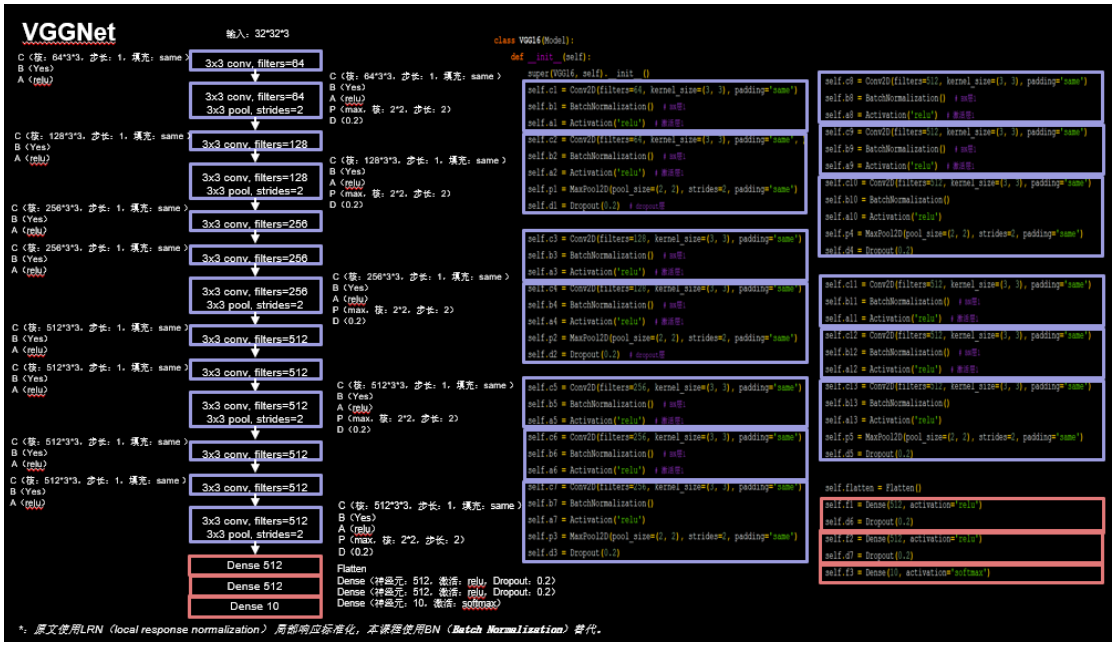


图 5-26 Keras 实现 VGG16 模型

根据特征图尺寸的变化，可以将 VGG16 模型分为六个部分（在 VGG16 中，每进行一次池化操作，特征图的边长缩小为 1/2，其余操作均未影响特征图尺寸）：

A）第一部分：两次卷积（64 个 3 * 3 卷积核、BN、Relu 激活）→最大池化→Dropout

```
self.c1 = Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='same', input_shape=(32, 32, 3))
self.b1 = BatchNormalization()  # BN层1
self.a1 = Activation('relu')  # 激活层1
self.c2 = Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='same', input_shape=(32, 32, 3))
self.b2 = BatchNormalization()  # BN层1
self.a2 = Activation('relu')  # 激活层1
self.p1 = MaxPool2D(pool_size=(2, 2), strides=2, padding='same')
self.d1 = Dropout(0.2)  # dropout层
```

B）第二部分：两次卷积（128 个 3 * 3 卷积核、BN、Relu 激活）→最大池化→Dropout

```
self.c3 = Conv2D(128, kernel_size=(3, 3), strides=1, padding='same')
self.b3 = BatchNormalization()  # BN层1
self.a3 = Activation('relu')  # 激活层1
self.c4 = Conv2D(128, kernel_size=(3, 3), strides=1, padding='same')
self.b4 = BatchNormalization()  # BN层1
self.a4 = Activation('relu')  # 激活层1
self.p2 = MaxPool2D(pool_size=(2, 2), strides=2, padding='same')
self.d2 = Dropout(0.2)  # dropout层
```

C）第三部分：三次卷积（256 个 3 * 3 卷积核、BN、Relu 激活）→最大池化→Dropout

```
self.c5 = Conv2D(256, kernel_size=(3, 3), strides=1, padding='same')
self.b5 = BatchNormalization()  # BN层1
self.a5 = Activation('relu')  # 激活层1
self.c6 = Conv2D(256, kernel_size=(3, 3), strides=1, padding='same')
self.b6 = BatchNormalization()  # BN层1
self.a6 = Activation('relu')  # 激活层1
self.c7 = Conv2D(256, kernel_size=(3, 3), padding='same')
self.b7 = BatchNormalization()
self.a7 = Activation('relu')
self.p3 = MaxPool2D(pool_size=(2, 2), strides=2, padding='same')
self.d3 = Dropout(0.2)
```

D）第四部分：三次卷积（512 个 3 * 3 卷积核、BN、Relu 激活）→最大池化→Dropout

```
self.c8 = Conv2D(512, kernel_size=(3, 3), strides=1, padding='same')
self.b8 = BatchNormalization()  # BN层1
self.a8 = Activation('relu')  # 激活层1
self.c9 = Conv2D(512, kernel_size=(3, 3), strides=1, padding='same')
self.b9 = BatchNormalization()  # BN层1
self.a9 = Activation('relu')  # 激活层1
self.c10 = Conv2D(512, kernel_size=(3, 3), padding='same')
self.b10 = BatchNormalization()
self.a10 = Activation('relu')
self.p4 = MaxPool2D(pool_size=(2, 2), strides=2, padding='same')
self.d4 = Dropout(0.2)
```

E）第五部分：三次卷积（512 个 3 * 3 卷积核、BN、Relu 激活）→最大池化→Dropout

```
self.c11 = Conv2D(512, kernel_size=(3, 3), strides=1, padding='same')
self.b11 = BatchNormalization()  # BN层1
self.a11 = Activation('relu')  # 激活层1
self.c12 = Conv2D(512, kernel_size=(3, 3), strides=1, padding='same')
self.b12 = BatchNormalization()  # BN层1
self.a12 = Activation('relu')  # 激活层1
self.c13 = Conv2D(512, kernel_size=(3, 3), padding='same')
self.b13 = BatchNormalization()
self.a13 = Activation('relu')
self.p5 = MaxPool2D(pool_size=(2, 2), strides=2, padding='same')
self.d5 = Dropout(0.2)
```

F）第六部分：全连接（512 个神经元）→Dropout→全连接（512 个神经元）→Dropout→全连接（10 个神经元）

```
self.flatten = Flatten()
self.f1 = Dense(512, activation='relu')
self.d6 = Dropout(0.2)
self.f2 = Dense(512, activation='relu')
self.d7 = Dropout(0.2)
self.f3 = Dense(10, activation='softmax')
```

总体来看，VGGNet 的结构是相当规整的，它继承了 AlexNet 中的 Relu 激活函数、Dropout 操作等有效的方法，同时采用了单一尺寸的 3 * 3 小卷积核，形成了规整的 C（Convolution，卷积）、B（Batch normalization）、A（Activation，激活）、P（Pooling，池化）、D（Dropout）结构，这一典型结构在卷积神经网络中的应用是非常广的。

5.3.4 InceptionNet

模型实现代码：p40_cifar10_inception26.py

借鉴点：一层内使用不同尺寸的卷积核，提升感知力（通过 padding 实现输出特征面积一致）；使用 1 * 1 卷积核，改变输出特征 channel 数（减少网络参数）。

InceptionNet 即 GoogLeNet，诞生于 2015 年，旨在通过增加网络的宽度来提升网络的能力，与 VGGNet 通过卷积层堆叠的方式（纵向）相比，是一个不同的方向（横向）。

显然，InceptionNet 模型的构建与 VGGNet 及之前的网络会有所区别，不再是简单的纵向堆叠，要理解 InceptionNet 的结构，首先要理解它的基本单元，如图 5-27 所示。

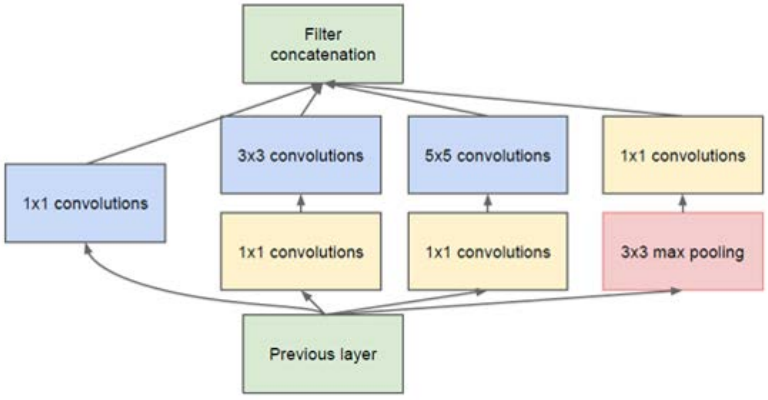


图 5-27 InceptionNet 基本单元

可以看到，InceptionNet 的基本单元中，卷积部分是比较统一的 C、B、A 典型结构，即卷积→BN→激活，激活均采用 Relu 激活函数，同时包含最大池化操作。

在 Tensorflow 框架下利用 Keras 构建 InceptionNet 模型时，可以将 C、B、A 结构封装在一起，定义成一个新的 ConvBNRelu 类，以减少代码量，同时更便于阅读。

```
class ConvBNRelu(Model):
    def __init__(self, ch, kernelsz=3, strides=1, padding='same'):
        super(ConvBNRelu, self).__init__()
        self.model = tf.keras.models.Sequential([
            Conv2D(ch, kernelsz, strides=strides, padding=padding),
            BatchNormalization(),
            Activation('relu')
        ])

    def call(self, x, training=None):
        x = self.model(x, training=training)
        return x
```

参数 ch 代表特征图的通道数，也即卷积核个数；kernelsz 代表卷积核尺寸；strides 代表卷积步长；padding 代表是否进行全零填充。

完成了这一步后，就可以开始构建 InceptionNet 的基本单元了，同样利用 class 定义的方式，定义一个新的 InceptionBlk 类，如 5-28 所示。

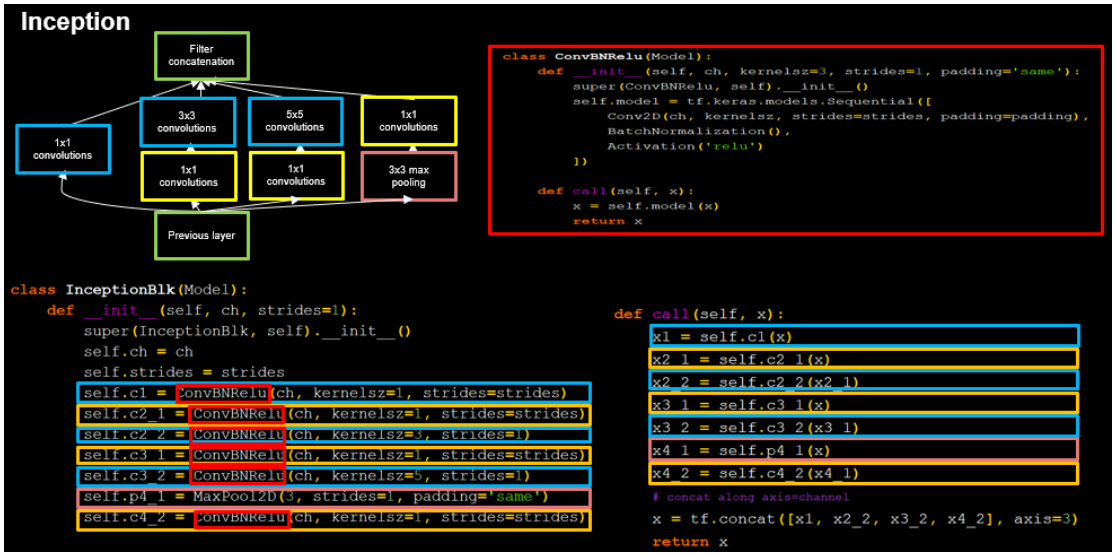


图 5-28 Inception 基本单元的实现

参数 ch 仍代表通道数，strides 代表卷积步长，与 ConvBNRelu 类中一致；tf.concat 函数将四个输出连接在一起，x1、x2_2、x3_2、x4_2 分别代表图 5-27 中的四列输出，结合结构图和代码很容易看出二者的对应关系。

可以看到，InceptionNet 的一个显著特点是大量使用了 1 * 1 的卷积核，事实上，最原始的 InceptionNet 的结构是不包含 1 * 1 卷积的，如图 5-29 所示。

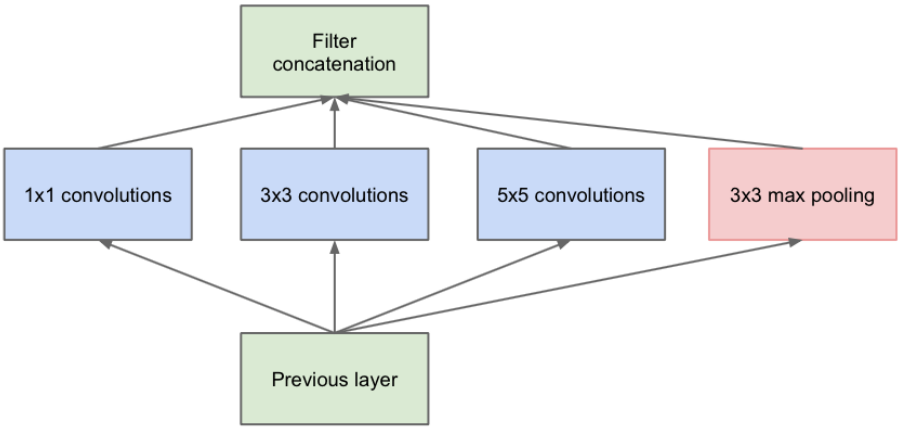


图 5-29 InceptionNet 最原始的基本单元

由图 5-29 可以更清楚地看出 InceptionNet 最初的设计思想，即通过不同尺寸卷积层和池化层的横向组合（卷积、池化后的尺寸相同，通道可以相加）来拓宽网络深度，可以增加网络对尺寸的适应性。但是这样也带来一个问题，所有的卷积核都会在上一层的输出上直接做卷积运算，会导致参数量和计算量过大（尤其是对于 5 * 5 的卷积核来说）。因此，InceptionNet 在 3 * 3、5 * 5 的卷积运算前、最大池化后均加入了 1 * 1 的卷积层，形成了图 5-24 中的结构，这样可以降低特征的厚度，一定程度上避免参数量过大的问题。

那么 1 * 1 的卷积运算是如何降低特征厚度的呢？下面以 5 * 5 的卷积运算为例说明这个问题。假设网络上一层的输出为 100 * 100 * 128（H * W * C），通过 32 * 5 * 5（32 个大小为 5 * 5 的卷积核）的卷积层（步长为 1、全零填充）后，输出为 100 * 100 * 32，卷积层的参数量为 32 * 5 * 5 * 128 = 102400；如果先通过 32 * 1 * 1 的卷积层（输出为 100 * 100 * 32），再通过 32 * 5 * 5 的卷积层，输出仍为 100 * 100 * 32，但卷积层的参数量变为 32 * 1 * 1 * 128 + 32 * 5 * 5 * 32 = 29696，仅为原参数量的 30 %左右，这就是小卷积核的降维作用。

InceptionNet 网络的主体就是由其基本单元构成的，其模型结构如图 5-30 所示。

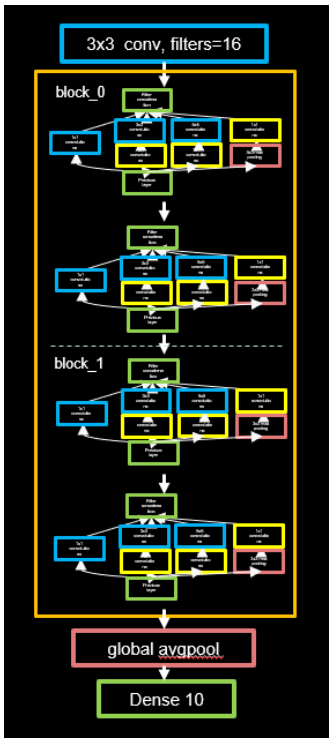


图 5-30 InceptionNet v1 模型结构图

图中橙色框内即为 InceptionNet 的基本单元，利用之前定义好的 InceptionBlk 类堆叠而成，模型的实现代码如下。

```
class Inception10(Model):
    def __init__(self, num_blocks, num_classes, init_ch=16, **kwargs):
        super(Inception10, self).__init__(**kwargs)
        self.in_channels = init_ch
        self.out_channels = init_ch
        self.num_blocks = num_blocks
        self.init_ch = init_ch
        self.c1 = ConvBNRelu(init_ch)
        self.blocks = tf.keras.models.Sequential()
        for block_id in range(num_blocks):
            for layer_id in range(2):
                if layer_id == 0:
                    block = InceptionBlk(self.out_channels, strides=2)
                else:
                    block = InceptionBlk(self.out_channels, strides=1)
                self.blocks.add(block)
            # enlarger out_channels per block
            self.out_channels *= 2
        self.p1 = GlobalAveragePooling2D()
        self.f1 = Dense(num_classes, activation='softmax')

    def call(self, x):
        x = self.c1(x)
        x = self.blocks(x)
        x = self.p1(x)
        y = self.f1(x)
        return y

model = Inception10(num_blocks=2, num_classes=10)
```

参数 num_layers 代表 InceptionNet 的 Block 数，每个 Block 由两个基本单元构成，每经过一个 Block，特征图尺寸变为 1/2，通道数变为 2 倍；num_classes 代表分类数，对于 cifar10 数据集来说即为 10；init_ch 代表初始通道数，也即 InceptionNet 基本单元的初始卷积核个数。

InceptionNet 网络不再像 VGGNet 一样有三层全连接层（全连接层的参数量占 VGGNet 总参数量的 90 %），而是采用“全局平均池化+全连接层”的方式，这减少了大量的参数。

这里介绍一下全局平均池化，在 tf.keras 中用 GlobalAveragePooling2D 函数实现，相比于平均池化（在特征图上以窗口的形式滑动，取窗口内的平均值为采样值），全局平均池化不再以窗口滑动的形式取均值，而是直接针对特征图取平均值，即每个特征图输出一个值。通过这种方式，每个特征图都与分类概率直接联系起来，这替代了全连接层的功能，并且不产生额外的训练参数，减小了过拟合的可能，但需要注意的是，使用全局平均池化会导致网络收敛的速度变慢。

总体来看，InceptionNet 采取了多尺寸卷积再聚合的方式拓宽网络结构，并通过 1 * 1 的卷积运算来减小参数量，取得了比较好的效果，与同年诞生的 VGGNet 相比，提供了卷积神经网络构建的另一种思路。但 InceptionNet 的问题是，当网络深度不断增加时，训练会十分困难，甚至无法收敛（这一点被 ResNet 很好地解决了）。

5.3.5 ResNet

模型实现代码：p46_cifar10_resnet18.py

借鉴点：层间残差跳连，引入前方信息，减少梯度消失，使神经网络层数变身成为可能。

ResNet 即深度残差网络，由何恺明及其团队提出，是深度学习领域又一具有开创性的工作，通过对残差结构的运用，ResNet 使得训练数百层的网络成为了可能，从而具有非常强大的表征能力，其网络结构如图 5-31 所示。

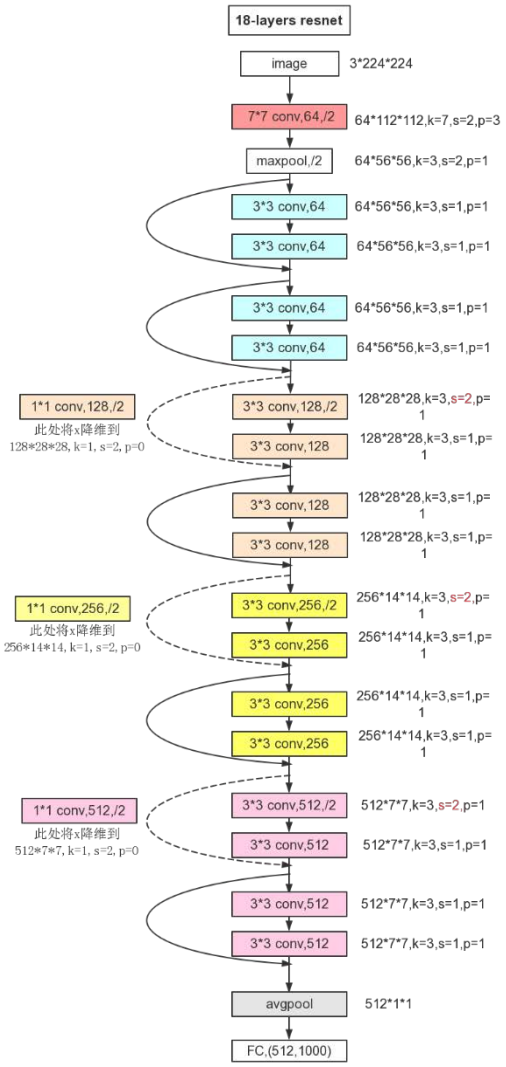


图 5-31 ResNet18 网络结构图

ResNet 的核心是残差结构，如图 5-32 所示。在残差结构中，ResNet 不再让下一层直接拟合我们想得到的底层映射，而是令其对一种残差映射进行拟合。若期望得到的底层映射为 $H(x)$，我们令堆叠的非线性层拟合另一个映射 $F(x) := H(x) - x$，则原有映射变为 $F(x) + x$。对这种新的残差映射进行优化时，要比优化原有的非相关映射更为容易。不妨考虑极限情况，如果一个恒等映射是最优的，那么将残差向零逼近显然会比利用大量非线性层直接进行拟合更容易。

值得一提的是，这里的相加与 InceptionNet 中的相加是有本质区别的，Inception 中的相加是沿深度方向叠加，像“千层蛋糕”一样，对层数进行叠加；ResNet 中的相加则是特征图对应元素的数值相加，类似于 python 语法中基本的矩阵相加。

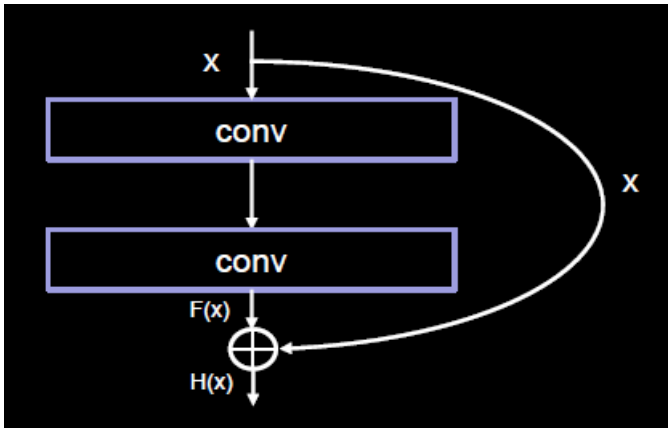


图 5-32 ResNet 中的残差结构

ResNet 引入残差结构最主要的目的是解决网络层数不断加深时导致的梯度消失问题，从之前介绍的 4 种 CNN 经典网络结构我们也可以看出，网络层数的发展趋势是不断加深的。这是由于深度网络本身集成了低层/中层/高层特征和分类器，以多层首尾相连的方式存在，所以可以通过增加堆叠的层数（深度）来丰富特征的层次，以取得更好的效果。

模型名称	网络层数
LeNet	5
AlexNet	8
VGG	16 / 19
InceptionNet v1	22

但如果只是简单地堆叠更多层数，就会导致梯度消失（爆炸）问题，它从根源上导致了函数无法收敛。然而，通过标准初始化（normalized initialization）以及中间标准化层（intermediate normalization layer），已经可以较好地解决这个问题了，这使得深度为数十层的网络在反向传播过程中，可以通过随机梯度下降（SGD）的方式开始收敛。

但是，当深度更深的网络也可以开始收敛时，网络退化的问题就显露了出来：随着网络深度的增加，准确率先是达到瓶颈（这是很常见的），然后便开始迅速下降。需要注意的是，这种退化并不是由过拟合引起的。对于一个深度比较合适的网络来说，继续增加层数反而会导致训练错误率的提升，图 5-33 就是一个例子。

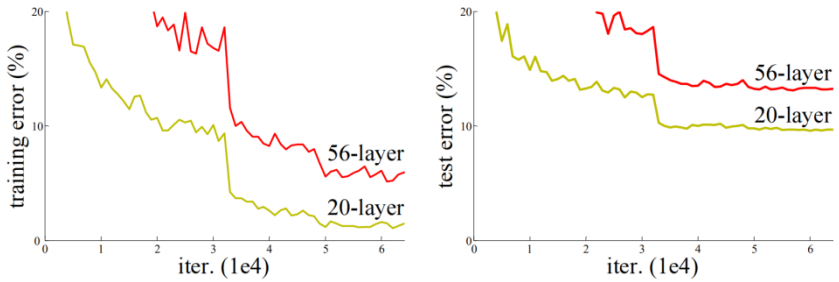


图 5-33 cifar10 数据集训练集错误率（左）及测试集错误率（右）

ResNet 解决的正是这个问题，其核心思路为：对一个准确率达到饱和的浅层网络，在它后面加几个恒等映射层（即 y = x，输出等于输入），增加网络深度的同时不增加误差。这使得神经网络的层数可以超越之前的约束，提高准确率。图 5-34 展示了 ResNet 中残差结构的具体用法。

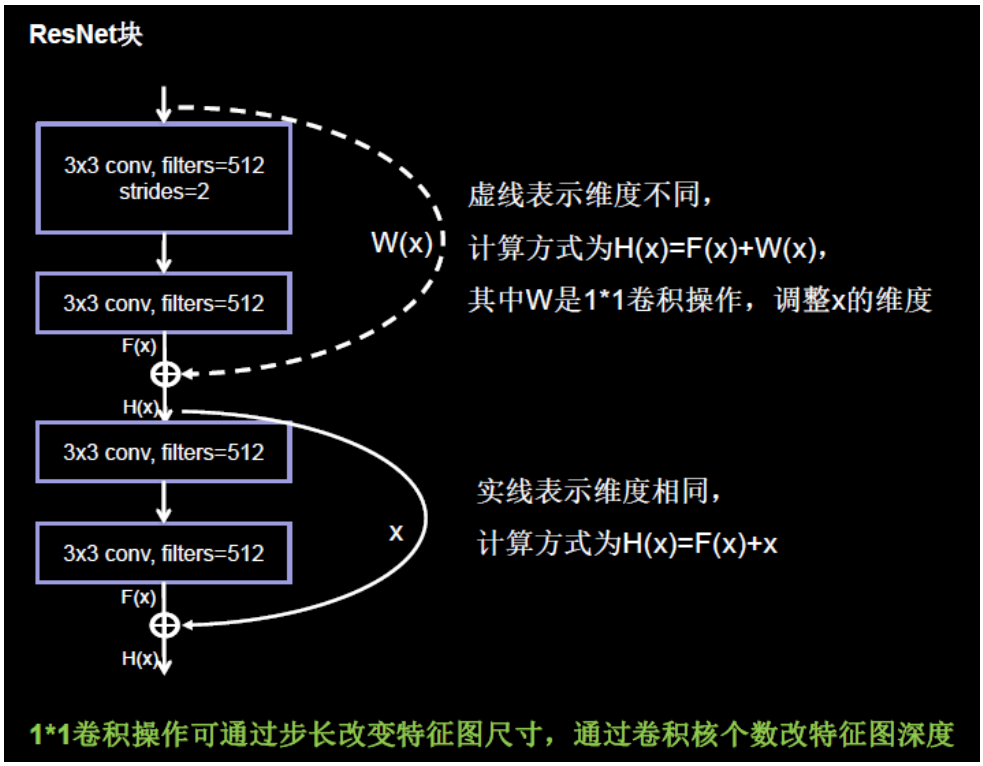


图 5-34 ResNet 中的残差结构

上图中的实线和虚线均表示恒等映射，实线表示通道相同，计算方式为 H(x) = F(x) + x；虚线表示通道不同，计算方式为 H(x) = F(x) + Wx，其中 W 为卷积操作，目的是调整 x 的维度（通道数）。

我们同样可以借助 tf.keras 来实现这种残差结构，定义一个新的 ResnetBlock 类。

```
class ResnetBlock(Model):

    def __init__(self, filters, strides=1, residual_path=False):
        super(ResnetBlock, self).__init__()
        self.filters = filters
        self.strides = strides
        self.residual_path = residual_path

        self.c1 = Conv2D(filters, (3, 3), strides=strides, padding='same', use_bias=False)
        self.b1 = BatchNormalization()
        self.a1 = Activation('relu')

        self.c2 = Conv2D(filters, (3, 3), strides=1, padding='same', use_bias=False)
        self.b2 = BatchNormalization()

        # residual_path为True时，对输入进行下采样，即用1x1的卷积核做卷积操作，保证x能和F(x)维度相同，顺利相加
        if residual_path:
            self.down_c1 = Conv2D(filters, (1, 1), strides=strides, padding='same', use_bias=False)
            self.down_b1 = BatchNormalization()

        self.a2 = Activation('relu')

    def call(self, inputs):
        residual = inputs  # residual等于输入值本身，即residual=x
        # 将输入通过卷积、BN层、激活层，计算F(x)
        x = self.c1(inputs)
        x = self.b1(x)
        x = self.a1(x)

        x = self.c2(x)
        y = self.b2(x)

        if self.residual_path:
            residual = self.down_c1(inputs)
            residual = self.down_b1(residual)

        out = self.a2(y + residual)  # 最后输出的是两部分的和，即F(x)+x或F(x)+Wx,再过激活函数
        return out
```

卷积操作仍然采用典型的 C、B、A 结构，激活采用 Relu 函数；为了保证 F(x)和 x 可以顺利相加，二者的维度必须相同，这里利用的是 1 * 1 卷积来实现（1 * 1 卷积改变输出维度的作用在 InceptionNet 中有具体介绍）。

利用这种结构，就可以利用 tf.keras 来构建出 ResNet 模型，如图 5-35 所示。

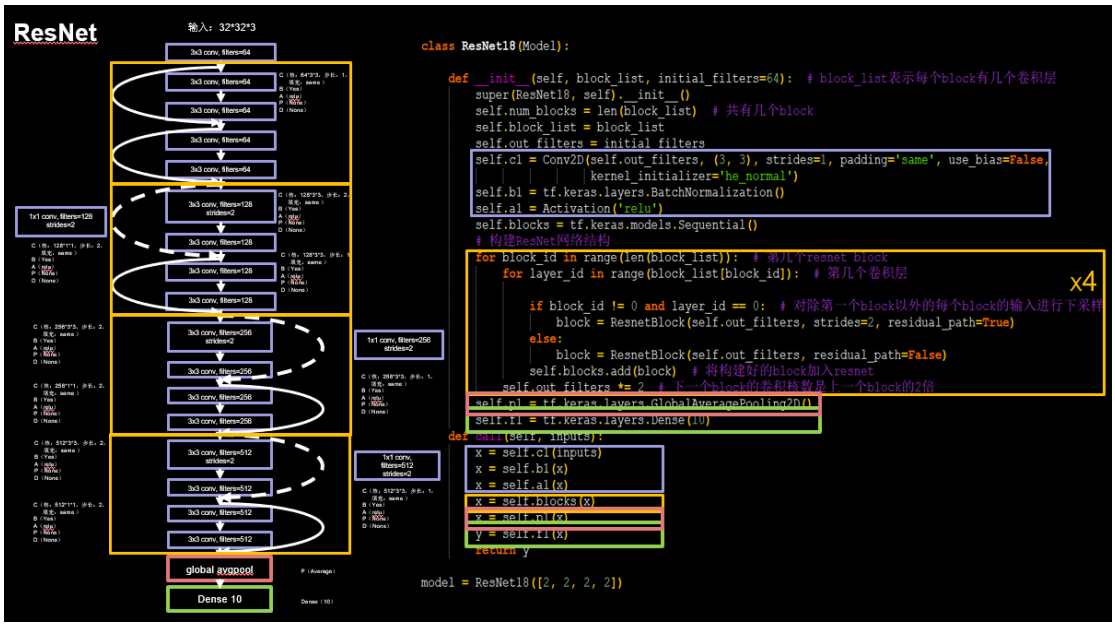


图 5-35 ResNet18 模型的代码实现

参数 block_list 表示 ResNet 中 block 的数量；initial_filters 表示初始的卷积核数量。可以看到该模型同样使用了全局平均池化的方式来替代全连接层（关于全局平均池化的作用 InceptionNet 中有介绍）。

对于 ResNet 的残差单元来说，除了这里采用的两层结构外，还有一种三层结构，如图 5-36 所示。

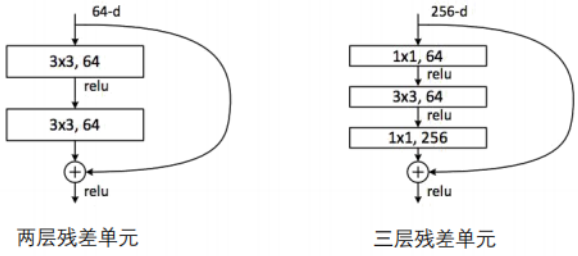


图 5-36 两层/三层残差单元

两层残差单元多用于层数较少的网络，三层残差单元多用于层数较多的网络，以减少计算的参数量。

总体上看，ResNet 取得的成果还是相当巨大的，它将网络深度提升到了 152 层，于 2015 年将 ImageNet 图像识别 Top5 错误率降至 3.57 %。

模型	AlexNet	ZF Net	GoogLeNet	ResNet
时间（年）	2012	2013	2014	2015
层数（层）	8	8	22	152
Top 5 错误率	15.4%	11.2%	6.7%	3.57%
数据增强	√	√	√	√
Dropout	√	√		
批量归一化				√

对上述的 5 种 CNN 经典结构进行总结，如图 5-37 所示。

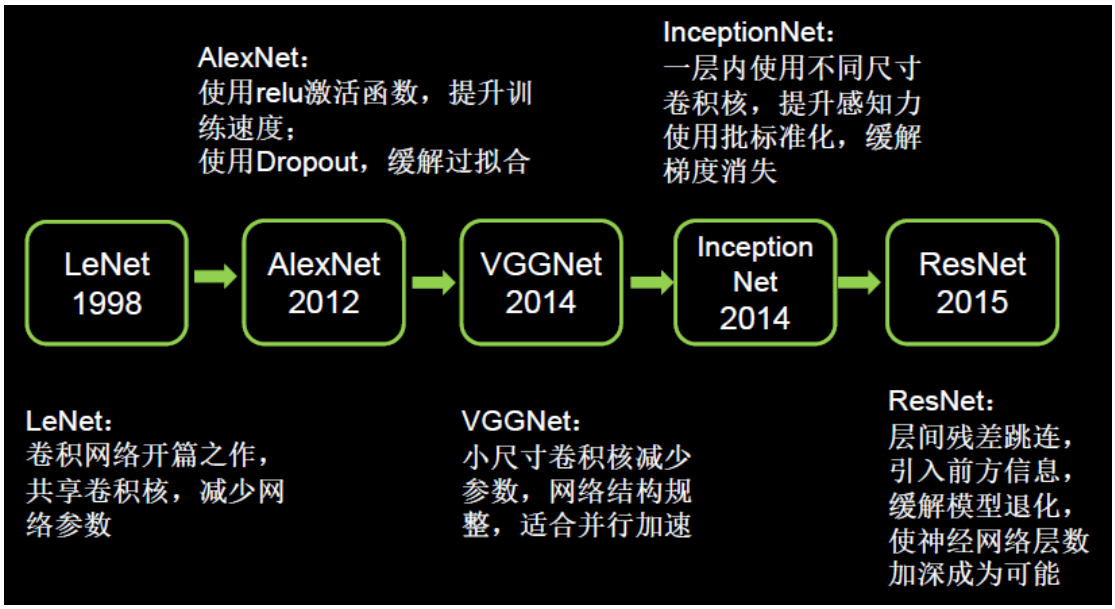
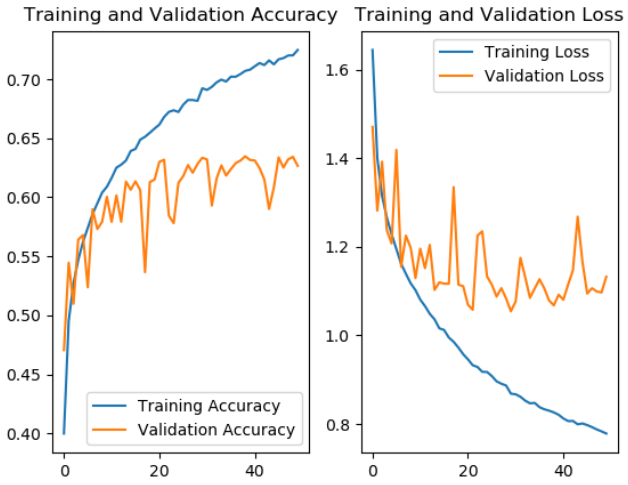


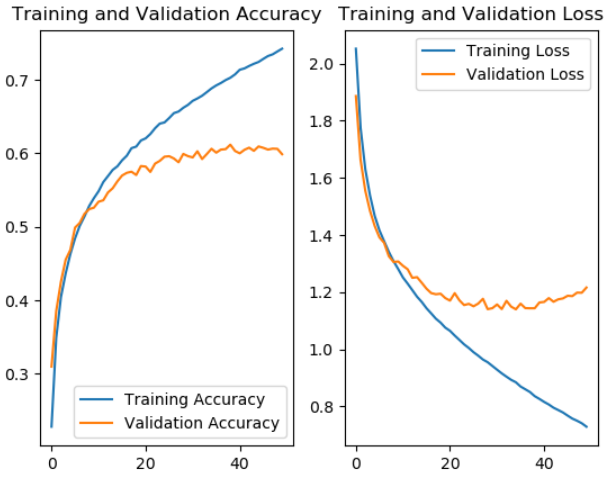
图 5-37 5 种经典网络结构的借鉴点

对于这五种网络（加上最基本的 baseline 共 6 种），课堂上均给出了代码实现，其测试集准确率曲线及 Loss 曲线如下。

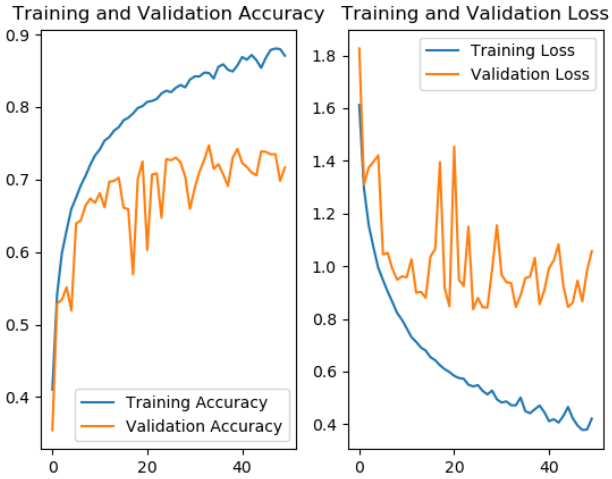
Baseline3：



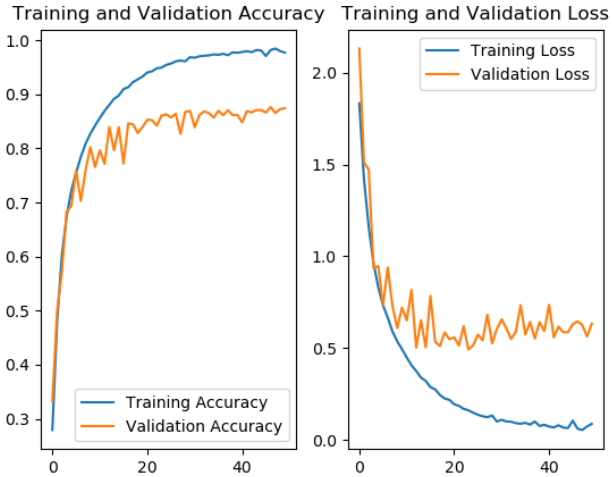
LeNet5：



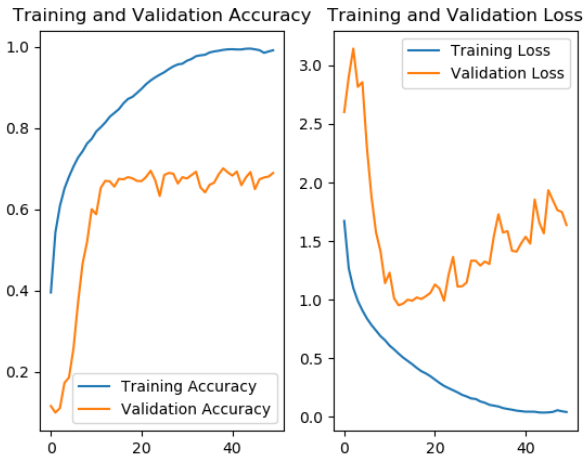
AlexNet8：



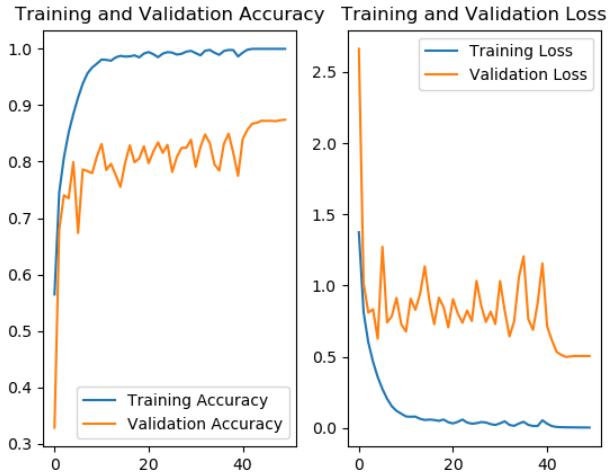
VGGNet16：



Inception10：



ResNet18：



可以看到，随着网络复杂程度的提高，以及 Relu、Dropout、BN 等操作的使用，利用各个网络训练 cifar10 数据集的准确率基本上是逐步上升的。五个网络当中，InceptionNet 的训练效果是最不理想的，首先其本身的设计理念是采用不同尺寸的卷积核，提供不同的感受野，但 cifar10 只是一个单一的分类任务，二者的契合度并不高，另外，由于本身结构的原因，InceptionNet 的参数量和计算量都比较大，训练需要耗费的资源比较多，所以课堂上仅仅搭建了一个深度为 10 的精简版本（完整的 InceptionNet v1，即 GoogLeNet 有 22 层，训练难度很大），主要目的是诠释 InceptionNet 的思路，并非单单追求 cifar10 数据集的准确率。

另外，需要指出的是，在利用这些网络训练 cifar10 数据集时，课程给出的源码并未包含其它的一些训练技巧，例如数据增强（对训练集图像进行旋转、偏移、翻转等多种操作，目的是增强训练集的随机性）、学习率策略（一般的策略是在训练过程中逐步减小学习率）、Batch size 的大小设置（每个 batch 包含训练集图片的数量）、模型参数初始化的方式等等。然而实际上，一些训练方法和超参数的设定对模型训练结果的影响是相当显著的，以 ResNet18 为例，如果采取合适的训练技巧，cifar10 的识别准确率是足以突破 90 %的。所以，在神经网络的训练中，除了选择合适的模型以外，如何更好地训练一个模型也是一个非常值得探究的问题。