5

深度神经网络

在本章中，我们将研究深度神经网络。这些网络在 ImageNet、CIFAR10[1]和 CIFAR100 等更具挑战性的数据集上的分类精度方面表现出色。为了简洁起见，我们将只关注两个网络。ResNet[2][4] 和 DenseNet[5]。虽然我们会讨论更多的细节，但有必要花点时间介绍一下这些网络。

ResNet 引入了残差学习的概念，通过解决深度卷积网络中的梯度消失问题（在第 2 节中讨论），使其能够建立非常深入的网络。

DenseNet 进一步改进了 ResNet，允许每个卷积都能直接访问输入，以及低层的特征图。它还通过利用瓶颈层和过渡层，成功地保持了深度网络中的低参数数量。

但是，为什么是这两种模型，而不是其他的？好吧，自从它们问世以来，已经有无数的模型，如 ResNeXt[6] 和 WideResNet[7]，它们的灵感来自这两个网络使用的技术。同样地，在了解了 ResNet 和 DenseNet 之后，我们就可以利用它们的设计准则来建立我们自己的模型。通过使用迁移学习，这也将使我们能够利用预先训练好的 ResNet 和 DenseNet 模型来达到我们自己的目的，比如用于物体检测和分割。仅仅这些原因，再加上它们与 Keras 的兼容性，使这两个模型成为探索和补充本书高级深度学习范围的理想选择。

虽然本章的重点是深度神经网络；但我们将在本章开始讨论 Keras 的一个重要功能，即功能 API。这个 API 作为在 tf.keras 中构建网络的另一种方法，使我们能够构建更复杂的网络，而这是顺序模型 API 所不能完成的。我们之所以如此关注这个 API，是因为它将成为构建深度网络的一个非常有用的工具，比如我们在本章重点讨论的两个网络。建议你在进入本章之前，先完成第 1 章”介绍 Keras 的高级深度学习”，因为在本章中，我们将参考该章中探讨的入门级代码和概念，将它们提升到高级水平。本章的目标是:

- Keras 中的功能 API，以及探索运行该 API 的网络实例；
- 深度残差网络（ResNet 1 和 2 版本）在 tf.keras 中的实现；
- 密集连接卷积网络（DenseNet）在 tf.keras 中的实现；
- 探索两种流行的深度学习模型，ResNet 和 DenseNet 让我们先来讨论一下 Functional API。

[1]Alex Krizhevsky 完整的论文 Learning Multiple Layers of Features from Tiny Images（从微小的图像中学习多层次的功能）读者可以在 https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf 下载阅读。

5.1. 1. 功能性 API

在我们在第 1 章《用 Keras 介绍高级深度学习》中首次介绍的顺序模型 API 中，一个层被堆叠在另一个层之上。一般来说，模型将通过其输入和输出层来访问。我们还了解到，如果我们发现自己想在网络中间添加一个辅助输入，甚至想在最后一层之前提取一个辅助输出，那就没有简单的机制了。该模型也有其缺点；例如，它不支持类似图的模型或行为类似 Python 函数的模型。此外，这两种模型之间也很难共享层。这样的局限性被 Functional API 解决了，这也是它成为任何想要使用深度学习模型的人的一个重要工具的原因。

功能性 API 是由以下两个概念指导的：

- 一个层是一个接受张量作为参数的实例。一个层的输出是另一个张量。为了建立一个模型，层的实例是通过输入和输出张量相互链接的对象。这将有一个类似于在序列模型中堆叠多个层的最终结果。然而，使用层实例使得模型更容易拥有辅助或多个输入和输出，因为每个层的输入/输出将很容易访问。
- 一个模型是一个或多个输入标量和输出标量之间的函数。在模型的输入和输出之间，张量是由层输入和输出张量相互连锁的层实例。因此，一个模型是一个或多个输入层和一个或多个输出层的函数。模型实例将数据如何从输入端流向输出端的计算图正式化。

在你完成构建功能型 API 模型后，训练和评估由顺序型模型中使用的相同函数来执行。举例来说，在 Functional API 中，一个二维卷积层，Conv2D，有 32 个滤波器，x 是层的输入张量，y 是层的输出张量，可以写成。

$$y = Conv2D(32)(x) \tag{5.1}$$

我们还能够堆叠多个层来建立我们的模型。例如，我们可以使用 Functional API 重写 MNIST 上的卷积神经网络（CNN）cnn-mnist-1.4.1.py，如以下列表所示。

Listing 5.1: Evaluate how the model does on the test set

```
import numpy as np
   from tensorflow.keras.layers import Dense, Dropout, Input
   from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
   from tensorflow.keras.models import Model
   from tensorflow.keras.datasets import mnist
   from tensorflow.keras.utils import to_categorical
   # load MNIST dataset
   (x_train, y_train), (x_test, y_test) = mnist.load_data()
   # from sparse label to categorical
   num_labels = len(np.unique(y_train))
   y_train = to_categorical(y_train)
   y_test = to_categorical(y_test)
   # reshape and normalize input images
image_size = x_train.shape[1]
x_train = np.reshape(x_train,[-1, image_size, image_size, 1])
x_test = np.reshape(x_test,[-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
# network parameters
input_shape = (image_size, image_size, 1)
batch_size = 128
kernel_size = 3
filters = 64
dropout = 0.3
```

```
# use functional API to build cnn layers
inputs = Input(shape=input_shape)
y = Conv2D(filters=filters,
           kernel_size=kernel_size,
           activation='relu')(inputs)
y = MaxPooling2D()(y)
y = Conv2D(filters=filters,
           kernel_size=kernel_size,
           activation='relu')(y)
y = MaxPooling2D()(y)
y = Conv2D(filters=filters,
           kernel_size=kernel_size,
           activation='relu')(y)
# image to vector before connecting to dense layer
y = Flatten()(y)
# dropout regularization
y = Dropout(dropout)(y)
outputs = Dense(num_labels, activation='softmax')(y)
# build the model by supplying inputs/outputs
model = Model(inputs=inputs, outputs=outputs)
# network model in text
model.summary()
# classifier loss, Adam optimizer, classifier accuracy
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
# train the model with input images and labels
   model.fit(x_train,
             y_train,
             validation_data=(x_test, y_test),
             epochs=20,
             batch_size=batch_size)
   # model accuracy on test dataset
   score = model.evaluate(x_test,
                          y_test,
                          batch_size=batch_size,
                          verbose=0)
   print("\nTest accuracy: %.1f%%" % (100.0 * score[1]))
```

默认情况下，MaxPooling2D 使用 pool_size=2，所以该参数已被删除。

在前面的列表中，每个层都是一个张量的函数。每一层都会产生一个张量作为输出，成为下一层的输入。为了创建这个模型，我们可以调用 Model() 并提供输入和输出张量，或者张量列表。其他一切都保持不变。

同样的列表也可以使用 fit() 和 evaluate() 函数进行训练和评估，与序列模型类似。事实上，Sequential 类是 Model 类的一个子类。我们需要记住，我们在 fit() 函数中插入了 validation_data 参数，以查看训练期间验证准确率的进展。在 20 个 epochs 中，准确率从 99.3 到 99.4。

5.1.1. 创建一个两进一出的模型

我们现在要做一件非常令人兴奋的事情，创建一个有两个输入和一个输出的高级模型。在我们开始之前，重要的是要知道序列模型 API 是为建立 1 输入和 1 输出的模型而设计的。我们假设发明了一个用于 MNIST 数字分类的新模型，它被称为 Y-网络，如图6.2所示。Y 型网络在 CNN 的左、右分支上都使用了两次相同的输入。该网络使用一个连接层将结果结合起来。合并操作 concatenate

类似于将两个相同形状的张量沿连接轴堆叠起来，形成一个张量。例如，将两个形状为（3，3，16）的张量沿最后一个轴线串联起来，将产生一个形状为（3，3，32）的张量。

连接层之后的其他一切都将与上一章的 CNN MNIST 分类器模型保持一致。扁平化，然后剔除，然后密集。

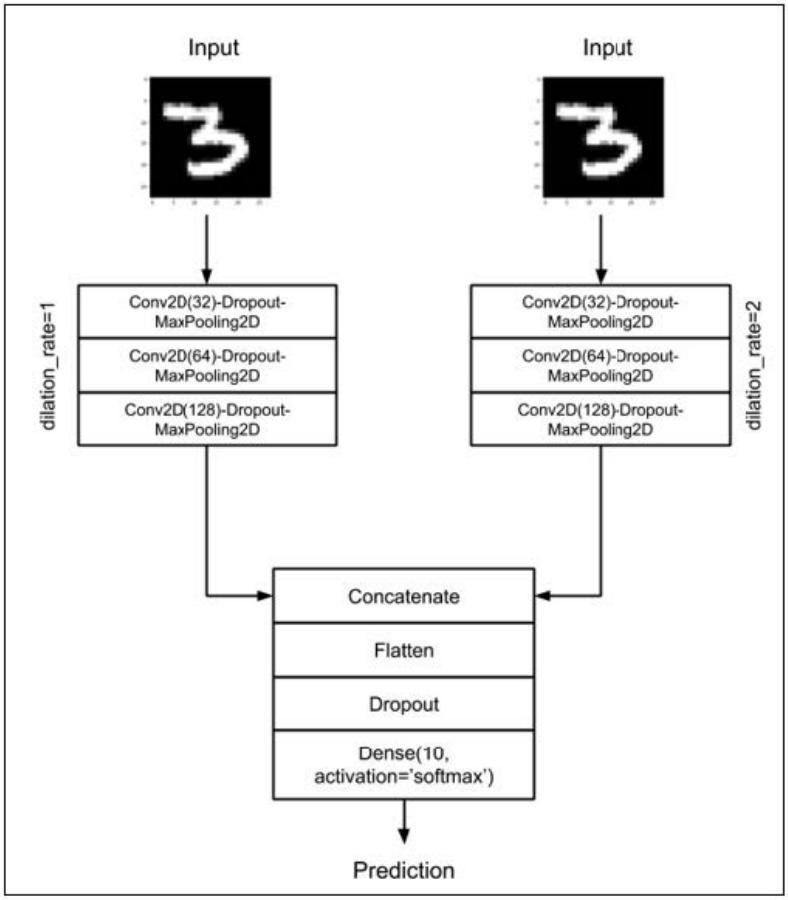

图 5.1: Y 网络接受两次相同的输入，但在卷积网络的两个分支中处理输入。这两个分支的输出通过连接层进行组合。最后一层的预测将类似于前一章的 CNN MNIST 分类器模型。

为了提高程序 2.1.1 中模型的性能，我们可以提出几个变化。首先，Y-Network 的分支将过滤器的数量增加一倍以弥补 MaxPooling2D() 之后特征图大小的减半。例如，如果第一个卷积的输出是（28，28，32），在最大池化之后，新的形状是（14，14，32）。下一次卷积的滤波器大小为 64，输出尺寸为（14, 14, 64）。第二，尽管两个分支的内核大小都是 3，但右边的分支使用的是 2 的扩张率。图 2.1.2 显示了不同的扩张率对内核大小为 3 的影响。我们的想法是，通过使用扩张率增加内核的有效接受场大小，CNN 将使右分支学习不同的特征图。使用大于 1 的扩张率是一种计算效率高的近似方法，可以增加感受野的大小。它是近似的，因为内核实际上不是一个完整的内核。它是有效的，因为我们使用的操作数量与扩张率等于 1 时相同。为了理解接受域的概念，请注意当内核计算一个特征图的每个点时，它的输入是上一层特征图中的一个补丁，它也依赖于它的上一层特征图。如果我们继续追踪这种依赖关系，直到输入图像，内核就会依赖一个叫做感受野的图像

补丁。我们将使用选项 padding='same' 来确保我们在使用扩张的 CNN 时不会出现负的张量尺寸。通过使用 padding='same'，我们将保持输入的尺寸与输出的特征图相同。这是通过在输入中填充零来实现的，以确保输出具有相同的尺寸。

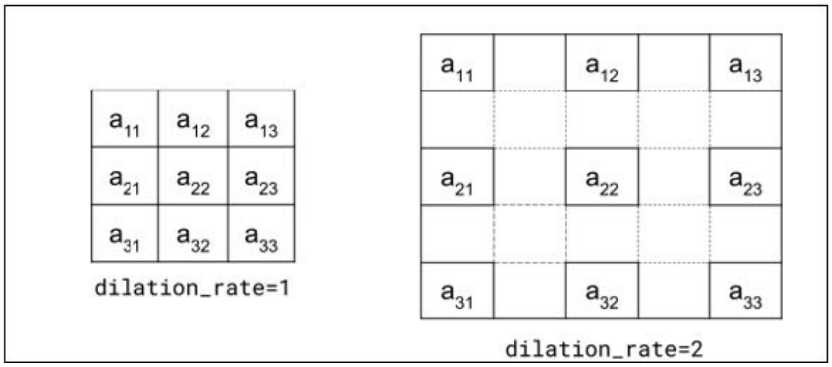

图 **5.2:** By increasing the dilation rate from 1, the effective kernel receptive field size also increases

cnn-y-network-2.1.2.py 的程序 2.1.2 显示了使用 Functional API 对 Y-Network 的实现。两个分支是由两个 for 循环创建的。两个分支都期望有相同的输入形状。这两个 for 循环将创建两个 3 层的 Conv2D-Dropout-MaxPooling2D 堆栈。虽然我们使用了 concatenate 层来合并左右两个分支的输出，但我们也可以利用 tf.keras 的其他合并函数，如 add、dot 和 multiply。合并函数的选择不是纯粹的任意，而是必须基于合理的模型设计决策。在 Y-Network 中，concatenate 不会丢弃特征图的任何部分。相反，我们会让密集层来决定如何处理连接后的特征图。

Listing 5.2: cnn-y-network-2.1.2.py

```
import numpy as np
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.layers import Flatten, concatenate
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.utils import plot_model
# load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# from sparse label to categorical
num_labels = len(np.unique(y_train))
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
# reshape and normalize input images
image_size = x_train.shape[1]
x_train = np.reshape(x_train,[-1, image_size, image_size, 1])
x_test = np.reshape(x_test,[-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
# network parameters
input_shape = (image_size, image_size, 1)
batch_size = 32
kernel_size = 3
dropout = 0.4
n_filters = 32
```

```
    # left branch of Y network
    left_inputs = Input(shape=input_shape)
    x = left_inputs
    filters = n_filters
    # 3 layers of Conv2D-Dropout-MaxPooling2D
    # number of filters doubles after each layer (32-64-128)
    for i in range(3):
        x = Conv2D(filters=filters,
                   kernel_size=kernel_size,
                   padding='same',
                   activation='relu')(x)
        x = Dropout(dropout)(x)
        x = MaxPooling2D()(x)
        filters *= 2
    # right branch of Y network
    right_inputs = Input(shape=input_shape)
    y = right_inputs
    filters = n_filters
    # 3 layers of Conv2D-Dropout-MaxPooling2Do
    # number of filters doubles after each layer (32-64-128)
    for i in range(3):
        y = Conv2D(filters=filters,
                   kernel_size=kernel_size,
                   padding='same',
                   activation='relu',
                   dilation_rate=2)(y)
        y = Dropout(dropout)(y)
        y = MaxPooling2D()(y)
        filters *= 2
    # merge left and right branches outputs
    y = concatenate([x, y])
    # feature maps to vector before connecting to Dense
    y = Flatten()(y)
    y = Dropout(dropout)(y)
    outputs = Dense(num_labels, activation='softmax')(y)
    # build the model in functional API
    model = Model([left_inputs, right_inputs], outputs)
    # verify the model using graph
    plot_model(model, to_file='cnn-y-network.png', show_shapes=True)
    # verify the model using layer text description
    model.summary()
    # classifier loss, Adam optimizer, classifier accuracy
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
    # train the model with input images and labels
    model.fit([x_train, x_train],
              y_train,
              validation_data=([x_test, x_test], y_test),
              epochs=20,
              batch_size=batch_size)
    # model accuracy on test dataset
    score = model.evaluate([x_test, x_test],
                           y_test,
                           batch_size=batch_size,
                           verbose=0)
    print("\nTest accuracy: %.1f%%" % (100.0 * score[1]))
```

退一步讲，我们可以注意到，Y-Network 期望有两个输入，用于训练和验证。这些输入是相同的，所以提供了 x_train, x_train□□20□$epochs$□□□□□$Y-Network$□□□□□99.4□□99.5□□□□□□3□CNN□□□□□□□□□$Network$□□□□

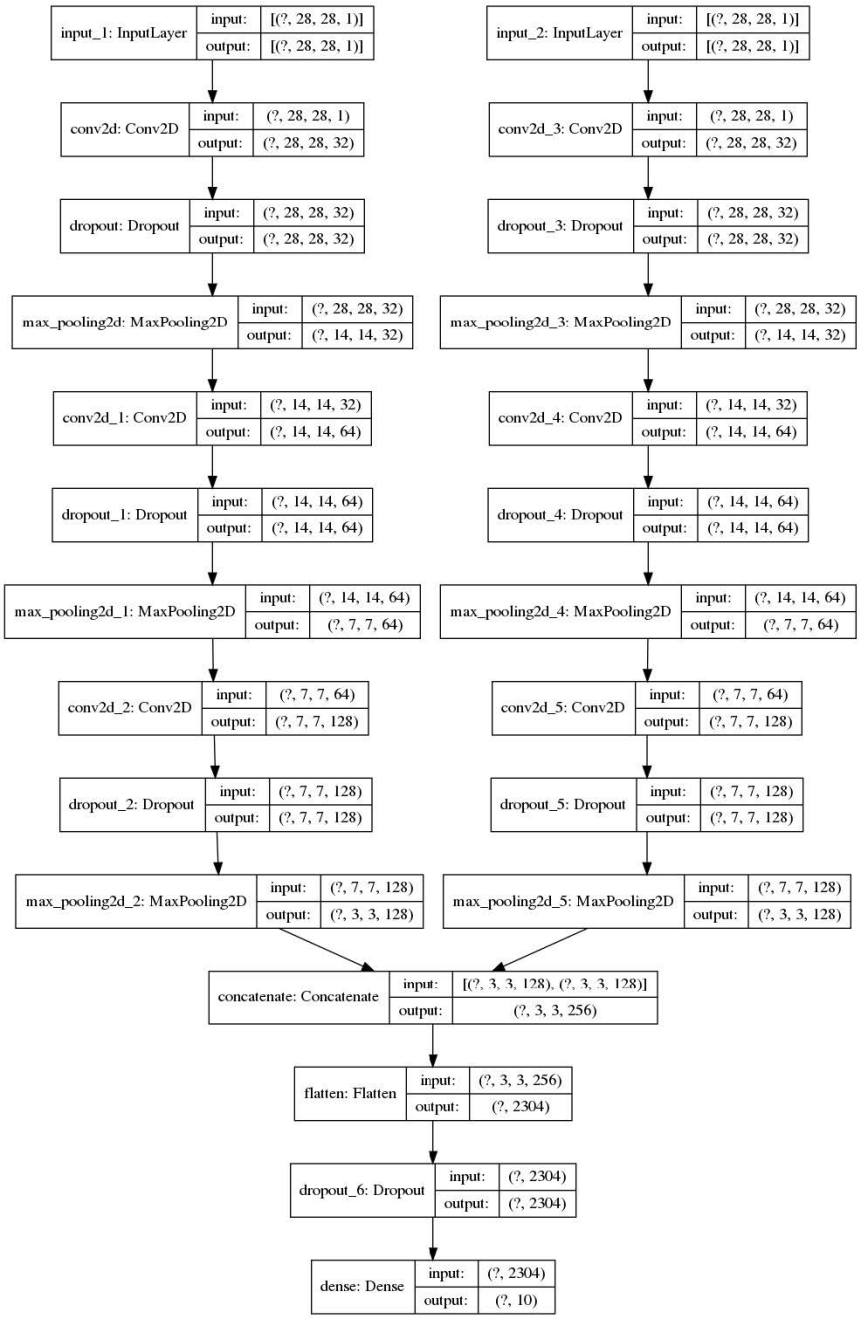

图 5.3: 在程序 2.1.2 中实现的 CNN Y-网络

至此，我们结束了对 Functional API 的研究。我们应该记住，本章的重点是构建深度神经网络，特别是 ResNet 和 DenseNet。因此，我们只涉及构建它们所需的 Functional API 材料，因为

涵盖整个 API 将超出本书的范围。说到这里，让我们继续讨论 ResNet。[2]

5.2. 深度剩余网络 (ResNet)

深度网络的一个关键优势是，它们有很大的能力从输入和特征图中学习不同层次的表示。在分类、分割、检测和其他一些计算机视觉问题中，学习不同的特征图通常会带来更好的性能。然而，你会发现，训练深度网络并不容易，因为在反向传播过程中，梯度可能会随着浅层的深度而消失（或爆炸）。图5.4说明了梯度消失的问题。网络参数通过反向传播从输出层更新到之前的所有层。由于反向传播是基于链式规则的，所以当它到达浅层时，梯度会有减少的趋势。这是由于小数的乘法，特别是对于小的损失函数和参数值。乘法运算的数量将与网络的深度成正比。还值得注意的是，如果梯度退化，参数将不会被适当地更新。

因此，网络将无法提高其性能。

[2]关于 Functional API 的其他信息，请读者参考 https://keras.io/。

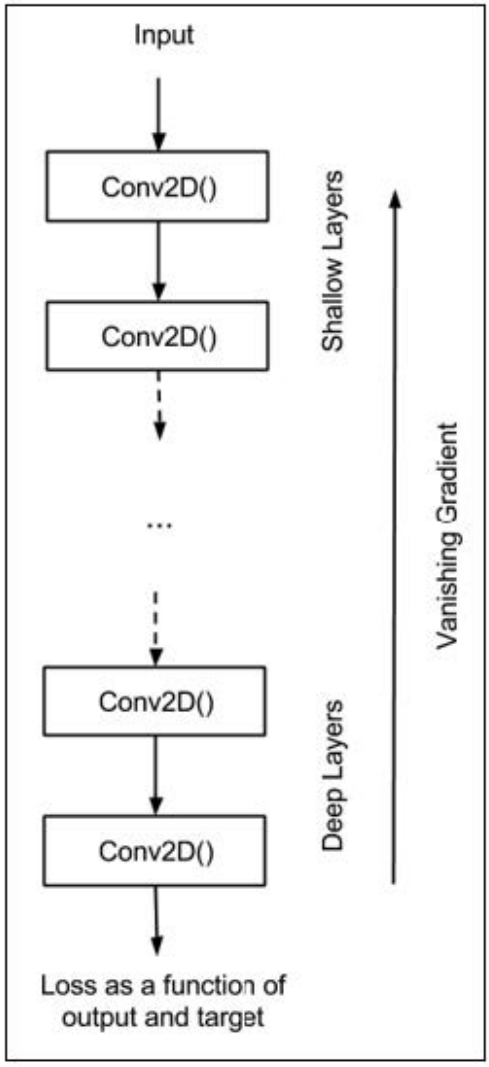

图 5.4: 深度网络中的一个常见问题是，在反向传播过程中，梯度在到达浅层时消失了。

为了缓解深度网络中梯度的退化问题，ResNet 引入了深度残差学习框架的概念。让我们来分析一个块：我们深度网络的一个小段。图5.5显示了一个典型的 CNN 块和一个 ResNet 残差块之间的比较。ResNet 的想法是，为了防止梯度退化，我们会让信息通过捷径连接流向浅层。

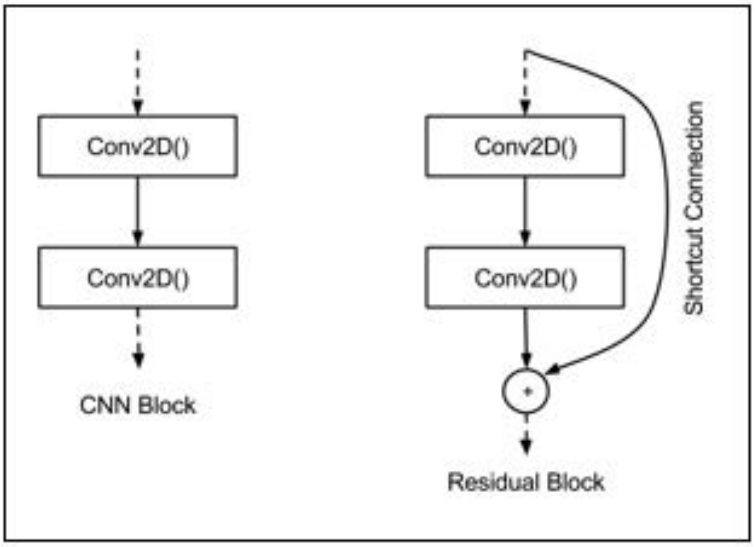

图 5.5: 典型的 CNN 中的块和 ResNet 中的块之间的比较。为了防止反向传播过程中梯度的退化，引入了一个快捷连接。

接下来，我们要在讨论这两个块的差异时，看一下更多的细节。图 2.2.3 显示了另一个常用的深度网络 VGG[3] 的 CNN 块和 ResNet 的更多细节。我们将层的特征图表示为 x，第 l 层的特征图为 x 和 l。CNN 层的操作是 Conv2D-Batch Normalization(BN)-ReLU。假设我们用 $H() = Conv2D - BatchNormalization(BN) - ReLU$ 的形式表示这组操作；那么:

$$x_{l-1} = H(x_{l-2}) \tag{5.2}$$

$$x_l = H(x_{l-1}) \tag{5.3}$$

换句话说，第 $l-2$ 层的特征图通过 $H() = Conv2D - BatchNormalization(BN) - ReLU$ 转换为 x_{l-1}。同样的一组操作被应用于将 x_{l-1} 转化为 x_l。换句话说，如果我们有一个 18 层的 VGG，那么在输入图像被转换到第 18 层特征图之前有 18 个 $H()$ 操作。一般来说，我们可以观察到，第 l 层的输出特征图只直接受到前面的特征图的影响。同时，对于 ResNet。

$$x_{l-1} = H(x_{l-2}) \tag{5.4}$$

$$x_l = ReLU(F(l-1) + x_{l-2}) \tag{5.5}$$

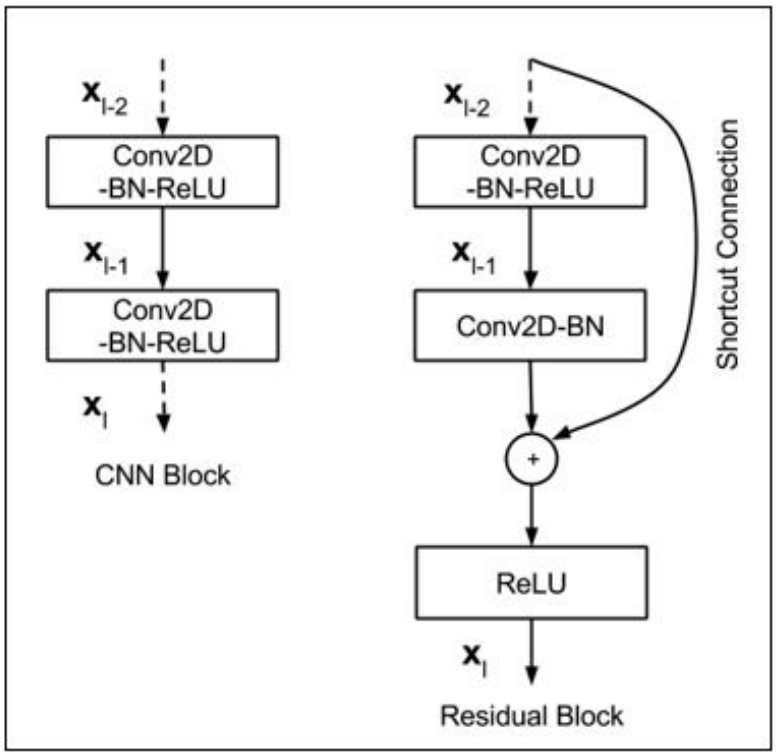

图 5.6: 普通 CNN 块和剩余块的详细层操作

$F(x_{l-1})$ 是由 Conv2D-BN 构成的，它也被称为残差映射。+ 号是捷径连接和 $F(x_{l-1})$ 的输出之间的张量元素相加。捷径连接并不增加额外的参数，也不增加额外的计算复杂性。加法操作可以在 tf.keras 中通过 add() 合并函数实现。然而，$F(x_{l-1})$ 和 x_{l-2} 都应该有相同的维度。如果维度不同，例如，当改变特征图的大小时，我们应该对 xl-2 进行线性投影，以匹配 F(xl-1) 的大小。在最初的论文中，当特征图大小减半时，线性投影是由一个 1×1 核和 strides=2 的 Conv2D 完成的。早在第一章《介绍 Keras 的高级深度学习》中，我们就讨论过 stride>1 相当于在卷积过程中跳过像素。例如，如果 strides=2，我们可以在卷积过程中滑动内核时跳过每一个其他像素。前面的方程 2.2.3 和方程 2.2.4 都是 ResNet 剩余块操作的模型。它们意味着，如果较深的层可以被训练成具有较少的错误，那么较浅的层就没有理由具有较高的错误。知道了 ResNet 的基本构件，我们就能设计一个用于图像分类的深层残差网络。然而，这一次，我们要处理的是一个更具挑战性的数据集。在我们的例子中，我们要考虑 CIFAR10，这是原始论文验证的数据集之一。在这个例子中，tf.keras 提供了一个 API 来方便地访问 CIFAR10 数据集，如图所示。

Listing 5.3: cnn-y-network-2.1.2.py

```
from tensorflow.keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

与 MNIST 一样，CIFAR10 数据集有 10 个类别。该数据集是一个小型（32×32）RGB 真实世界图像的集合，包括飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船和卡车，分别对应 10 个类别。图5.7显示了 CIFAR10 的样本图像。

在该数据集中，有 50,000 张标记的训练图像和 10,000 张标记的测试图像用于验证。

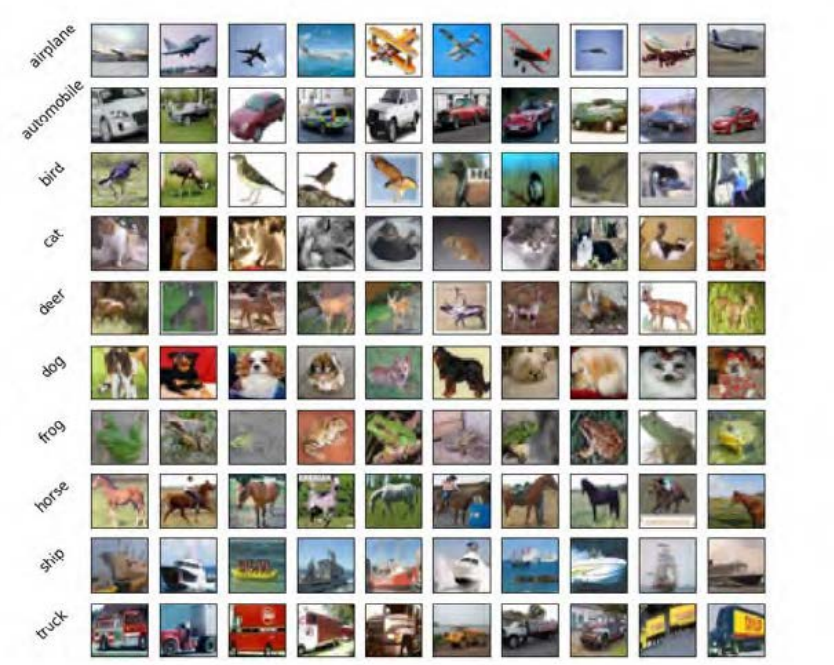

图 5.7: CIFAR10 数据集的样本图像。完整的数据集有 50,000 张标记的训练图像和 10,000 张标记的测试图像用于验证。

对于 CIFAR10 数据，可以使用不同的网络结构构建 ResNet，如表 2.2.1 所示。表 2.2.1 意味着我们有三组残差块。每组有 $2n$ 个层，对应 n 个残差块。32×32 中的额外层是输入图像的第一层。

表 5.1: ResNet 网络架构配置

Layers 层	输出尺寸	过滤尺寸	操作
Convolution	32 $\times$32	16	3 $\times$3$Conv2D$
Residual Block (1)	32 $\times$32		$\begin{pmatrix} 3\times3 & Conv2D \\ 3\times3 & Conv2D \end{pmatrix}\times n$
Transition Layer (1)	32 $\times$32		{1 $\times$1$Conv2D, strides = 2$}
	16 $\times$16		Residual Block (2)
16 $\times$16	32	singleton null object	None
None	singleton null object	None	None
singleton null object	None	None	singleton null object

核大小为 3，除了两个不同大小的特征图之间的过渡，实现了线性映射。例如，一个核大小为 1、步长 =2 的 Conv2D。为了与 DenseNet 保持一致，当我们连接两个不同大小的残差块时，我们将

使用过渡层这个术语。ResNet 使用 kernel_initializer='he_normal'，以便在反向传播进行时帮助收敛 [1]。最后一层是由 AveragePooling2D-Flatten-Dense 组成。在这一点上值得注意的是，ResNet 并没有使用 dropout。另外，添加合并操作和 1×1 卷积似乎也有自我规范化的作用。图5.8显示了表 2.2.1 中描述的 CIFAR10 数据集的 ResNet 模型架构。

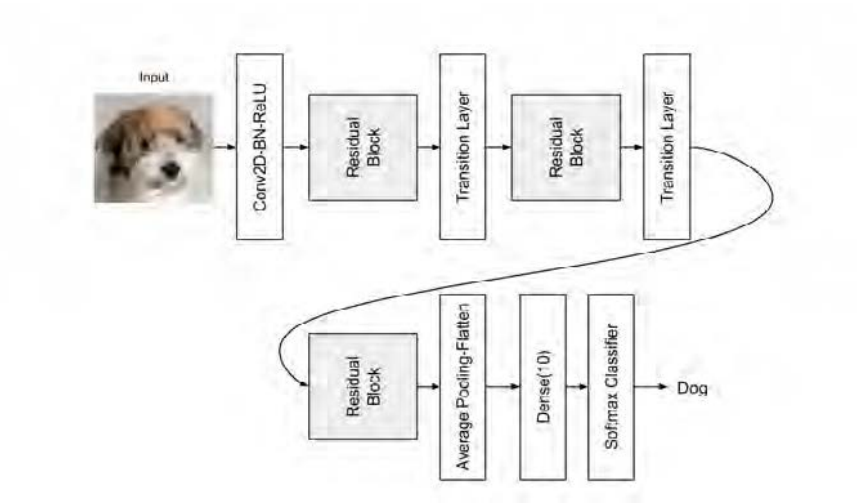

图 5.8: 用于 CIFAR10 数据集分类的 ResNet 的模型结构

下面的代码片段显示了 tf.keras 中的部分 ResNet 实现。该代码已被贡献到 Keras 的 GitHub 仓库。从表 2.2.2（即将显示）我们还可以看到，通过修改 n 的值，我们能够增加网络的深度。

例如，对于 $n = 18$，我们已经有了 ResNet110，一个有 110 层的深度网络。为了建立 ResNet20，我们使用 $n = 3$。

Listing 5.4: 部分 ResNet 实现代码

```
n= 3
   # model version
   # orig paper: version = 1 (ResNet v1),
   # improved ResNet: version = 2 (ResNet v2)
   version = 1
   # computed depth from supplied model parameter n
   if version == 1:
       depth = n * 6 + 2
   elif version == 2:
depth = n * 9 + 2
   if version == 2:
       model = resnet_v2(input_shape=input_shape, depth=depth)
   else:
       model = resnet_v1(input_shape=input_shape, depth=depth)
```

resnet_v1() 方法是 ResNet 的一个模型构建器。它使用一个实用函数 resnet_layer() 来帮助建立 Conv2D-BN-ReLU 的堆栈。它被称为版本 1，正如我们将在下一节看到的，一个改进的 ResNet 被提出来了，这被称为 ResNet 第二版，或称 v2 版。与 ResNet 相比，ResNet v2 有一个改进的剩余块设计，从而获得了更好的性能。下面列出了 resnet-cifar10-2.2.1.py 的部分代码，这是 ResNet v1 的 tf.keras 模型实现。

Listing 5.5: resnet-cifar10-2.2.1.py 的部分代码

```
Listing 2.2.1: resnet-cifar10-2.2.1.py
def resnet_v1(input_shape, depth, num_classes=10):
    """ResNet Version 1 Model builder [a]
    Stacks of 2 x (3 x 3) Conv2D-BN-ReLU
    Last ReLU is after the shortcut connection.
    At the beginning of each stage, the feature map size is halved
    (downsampled) by a convolutional layer with strides=2, while
    the number of filters is doubled. Within each stage,
    the layers have the same number filters and the
    same number of filters.
    Features maps sizes:
    stage 0: 32x32, 16
    stage 1: 16x16, 32
    stage 2:  8x8,  64
    The Number of parameters is approx the same as Table 6 of [a]:
    ResNet20 0.27M
    ResNet32 0.46M
    ResNet44 0.66M
    ResNet56 0.85M
    ResNet110 1.7M
    Arguments:
        input_shape (tensor): shape of input image tensor
        depth (int): number of core convolutional layers
        num_classes (int): number of classes (CIFAR10 has 10)
    Returns:
        model (Model): Keras model instance
    """
    if (depth - 2) % 6 != 0:
        raise ValueError('depth should be 6n+2 (eg 20, 32, in [a])')
    # Start model definition.
    num_filters = 16
    num_res_blocks = int((depth - 2) / 6)
    inputs = Input(shape=input_shape)
    x = resnet_layer(inputs=inputs)
    # instantiate the stack of residual units
    for stack in range(3):
        for res_block in range(num_res_blocks):
            strides = 1
            # first layer but not first stack
            if stack > 0 and res_block == 0:
                strides = 2  # downsample
            y = resnet_layer(inputs=x,
                             num_filters=num_filters,
                             strides=strides)
            y = resnet_layer(inputs=y,
                             num_filters=num_filters,
                             activation=None)
            # first layer but not first stack
            if stack > 0 and res_block == 0:
                # linear projection residual shortcut
                # connection to match changed dims
                x = resnet_layer(inputs=x,
                                 num_filters=num_filters,
                                 kernel_size=1,
                                 strides=strides,
                                 activation=None,
```

```
                                 batch_normalization=False)
            x = add([x, y])
            x = Activation('relu')(x)
        num_filters *= 2
    # add classifier on top.
    # v1 does not use BN after last shortcut connection-ReLU
    x = AveragePooling2D(pool_size=8)(x)
    y = Flatten()(x)
    outputs = Dense(num_classes,
                    activation='softmax',
                    kernel_initializer='he_normal')(y)
    # instantiate model.
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

表 2.2.2 显示了 ResNet 在不同 n 值上的性能。

Table 2.2.2: ResNet architecture validated with CIFAR10 for different values of n

与 ResNet 的原始实现有一些小的区别。特别是，我们不使用 SGD，而是使用 Adam。这是因为 ResNet 用 Adam 更容易收敛。我们还将使用一个学习率（lr）调度器，lr_schedule()，以便在 80、120、160 和 180 个 epochs 安排 lr 从默认的 1e-3 下降。lr_schedule() 函数将在训练期间的每个 epoch 后作为回调变量的一部分被调用。另一个回调是在每次验证精度有进展时保存检查点。在训练深度网络时，保存模型或权重检查点是一个好的做法。这是因为训练深度网络需要大量的时间。

当你想使用你的网络时，你所需要做的就是简单地重新加载检查点，训练过的模型就会恢复。这可以通过调用 tf.keras load_model() 来实现。lr_reducer() 函数也包括在内。如果指标在预定的减少之前已经达到平稳，如果验证损失在耐心 =5 个 epochs 之后没有改善，这个回调函数将通过参数中提供的某个系数减少学习率。回调变量在调用 model.fit() 方法时被提供。与原始论文类似，tf.keras 的实现使用了数据增强，ImageDataGenerator()，以便提供额外的训练数据作为正则化方案的一部分。随着训练数据数量的增加，泛化效果将得到改善。例如，一个简单的数据增强是翻转一张狗的照片，如图 2.2.6 所示（horizontal_flip = True）。如果这是一张狗的图片，那么翻转后的图片仍然是一张狗的图片。你还可以进行其他的变换，比如缩放、旋转、美白等等，而标签仍然保持不变。

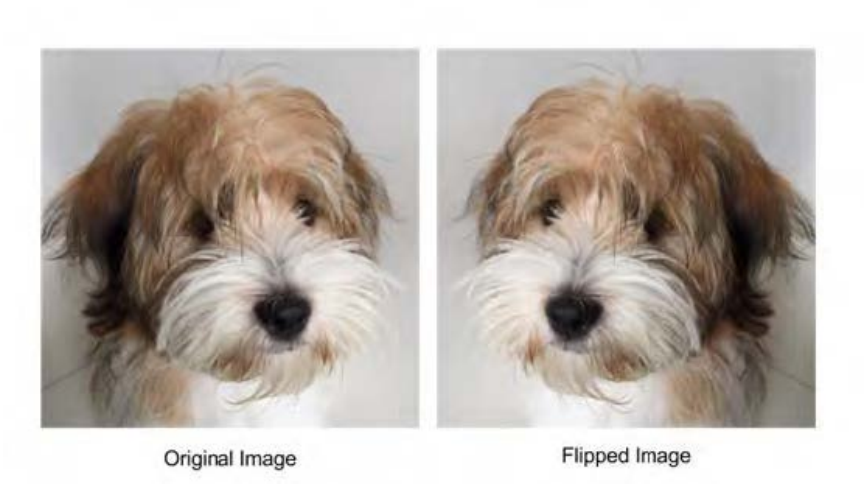

图 5.9: 一个简单的数据增强是翻转原始图像

要完全复制原始论文的实现往往是很困难的。在这本书中，我们使用了不同的优化器和数据增量。这可能会导致本书中实现的 tf.keras ResNet 的性能与原论文中的模型有细微差别。在关于 ResNet 的第二篇论文 [4] 发布后，本节介绍的原始模型被称为 ResNet v1，改进后的 ResNet 通常被称为 ResNet v2，我们将在下一节讨论。

5.3. ResNet v2

ResNet v2 的改进主要体现在图 2.3.1 所示的剩余块中的层的排列。

ResNet v2 的突出变化是：

- 使用 1×1-3×3-1×1 的 BN-ReLU-Conv2D 堆栈。
- 批量归一化和 ReLU 激活排在二维卷积之前

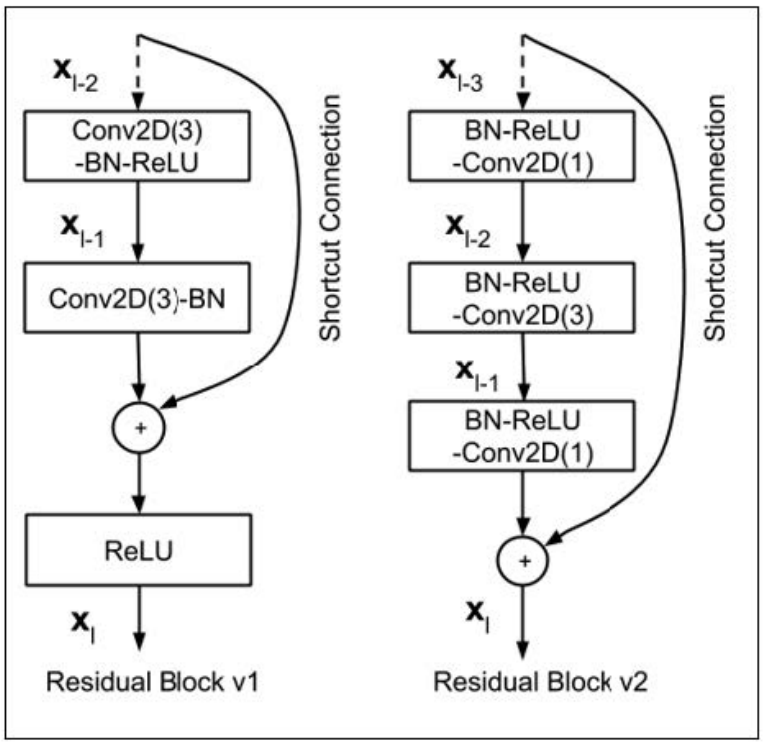

图 **5.10:** ResNet v1 和 ResNet v2 之间的残留块的比较

ResNet v2 也是在与 resnet-cifar10-2.2.1.py 相同的代码中实现的，在程序 2.2.1 中可以看到这一点。

Listing 5.6: ResNet v2 也是在与 resnet-cifar10-2.2.1.py 相同的代码中实现的

```
Listing 2.2.1: resnet-cifar10-2.2.1.py
    def resnet_v2(input_shape, depth, num_classes=10):
        """ResNet Version 2 Model builder [b]
        Stacks of (1 x 1)-(3 x 3)-(1 x 1) BN-ReLU-Conv2D or
        also known as bottleneck layer.
        First shortcut connection per layer is 1 x 1 Conv2D.
        Second and onwards shortcut connection is identity.
        At the beginning of each stage,
        the feature map size is halved (downsampled)
        by a convolutional layer with strides=2,
        while the number of filter maps is
        doubled. Within each stage, the layers have
        the same number filters and the same filter map sizes.
        Features maps sizes:
        conv1  : 32x32,  16
        stage 0: 32x32,  64
        stage 1: 16x16, 128
        stage 2:  8x8,  256
        Arguments:
            input_shape (tensor): shape of input image tensor
            depth (int): number of core convolutional layers
            num_classes (int): number of classes (CIFAR10 has 10)
```

```
        Returns:
            model (Model): Keras model instance
        """
        if (depth - 2) % 9 != 0:
            raise ValueError('depth should be 9n+2 (eg 110 in [b])')
        # start model definition.
        num_filters_in = 16
        num_res_blocks = int((depth - 2) / 9)
        inputs = Input(shape=input_shape)
        # v2 performs Conv2D with BN-ReLU
        # on input before splitting into 2 paths
        x = resnet_layer(inputs=inputs,
                         num_filters=num_filters_in,
                         conv_first=True)
        # instantiate the stack of residual units
        for stage in range(3):
            for res_block in range(num_res_blocks):
                activation = 'relu'
                batch_normalization = True
                strides = 1
                if stage == 0:
                    num_filters_out = num_filters_in * 4
                    # first layer and first stage
                    if res_block == 0:
                        activation = None
                        batch_normalization = False
                else:
                    num_filters_out = num_filters_in * 2
                    # first layer but not first stage
                    if res_block == 0:
                    # downsample
                     strides = 2
         # bottleneck residual unit
         y = resnet_layer(inputs=x,
                          num_filters=num_filters_in,
                          kernel_size=1,
                          strides=strides,
                          activation=activation,
                          batch_normalization=batch_normalization,
                          conv_first=False)
         y = resnet_layer(inputs=y,
                          num_filters=num_filters_in,
                          conv_first=False)
         y = resnet_layer(inputs=y,
                          num_filters=num_filters_out,
                          kernel_size=1,
                          conv_first=False)
         if res_block == 0:
             # linear projection residual shortcut connection
             # to match changed dims
             x = resnet_layer(inputs=x,
                              num_filters=num_filters_out,
                              kernel_size=1,
                              strides=strides,
                              activation=None,
                              batch_normalization=False)
         x = add([x, y])
```

```
    num_filters_in = num_filters_out
# add classifier on top.
# v2 has BN-ReLU before Pooling
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = AveragePooling2D(pool_size=8)(x)
y = Flatten()(x)
outputs = Dense(num_classes,
                activation='softmax',
                kernel_initializer='he_normal')(y)
# instantiate model.
model = Model(inputs=inputs, outputs=outputs)
return model
```

ResNet v2 的模型构建器如下代码所示。例如，为了构建 ResNet110 v2，我们将使用 $n = 12$ 和版本 $n = 2$。

Listing 5.7: ResNet v2 的模型构建器如下代码所示

```
n = 12
# model version
# orig paper: version = 1 (ResNet v1),
# improved ResNet: version = 2 (ResNet v2) version = 2
   # computed depth from supplied model parameter n
   if version == 1:
       depth = n * 6 + 2
   elif version == 2:
depth = n * 9 + 2
   if version == 2:
       model = resnet_v2(input_shape=input_shape, depth=depth)
   else:
       model = resnet_v1(input_shape=input_shape, depth=depth)
```

ResNet v2 的准确度如下表 2.3.1 所示。Table 2.3.1: The ResNet v2 architectures validated on the CIFAR10 dataset

在 Keras 应用包中，某些 ResNet v1 和 v2 模型（例如：50、101、152）已经被实现。这些是预先训练好的权重不明确的替代实现，可以很容易地重复用于迁移学习。本书中使用的模型在层数方面提供了灵活性。我们已经完成了对最常用的深度神经网络之一 ResNet v1 和 v2 的讨论。在下一节中，将介绍另一个流行的深度神经网络架构–DenseNet。

5.4. 密集连接的卷积网络

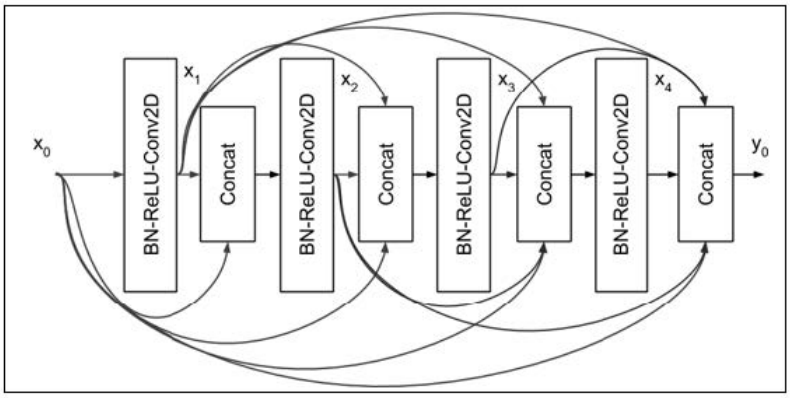

图 5.11: DenseNet 中的 4 层 Dense 块。每一层的输入是由之前所有的特征图组成的。

DenseNet 用一种不同的方法来攻击梯度消失的问题。不使用捷径连接，而是所有之前的特征图都将成为下一层的输入。上图显示了一个 Dense 块中的 Dense 互连的例子。为了简单起见，在这个图中，我们只显示四层。请注意，第 l 层的输入是之前所有特征图的串联。如果我们让 BN-ReLU-Conv2D 用运算 H(x) 来表示，那么第 l 层的输出就是。

$$x_1 = H(x_0, x_1, x_2, \cdots x_{l-1}) \tag{5.6}$$

Conv2D 使用一个大小为 3 的核。每层生成的特征图数量称为增长率，即 k。通常情况下，$k = 12$，但在 Huang 等人（2017）的论文《密集连接卷积网络》中也使用了 $k = 24$[5]。因此，如果特征图 x_0 的数量是 k_0，那么图 2.4.1 中 4 层密集块结束时的特征图总数将是 $4 \times k + k_0$。DenseNet 建议在 Dense 块之前加入 BN-ReLU-Conv2D，同时加入两倍于增长率的特征图数量，$k_0 = 2xk$。在输出层，DenseNet 建议我们在 Dense() 之前用 softmax 层进行一次平均池化。如果不使用数据增强，在 Dense block Conv2D 之后必须有一个 dropout 层。

随着网络的深入，会出现两个新的问题。首先，由于每一层都贡献了 k 个特征图，第 l 层的输入数是 $(l - 1\square xk + k_0$。特征图在深层内会迅速增长，使计算速度减慢。例如，对于一个 101 层的网络，这将是 $1200 + 24 = 1224$，对于 $k = 12$。

其次，与 ResNet 类似，随着网络的深入，特征图的大小将被缩小，以增加内核的接受场大小。如果 DenseNet 在合并操作中使用串联法，它必须调和大小的差异。为了防止特征图的数量增加到计算效率低下的程度，DenseNet 引入了瓶颈层，如图5.12所示。其想法是，在每次串联之后，现在应用一个 $1x1$ 的卷积，过滤器大小等于 $4k$。这种降维技术可以防止 Conv2D(3) 所要处理的特征图的数量迅速增加。

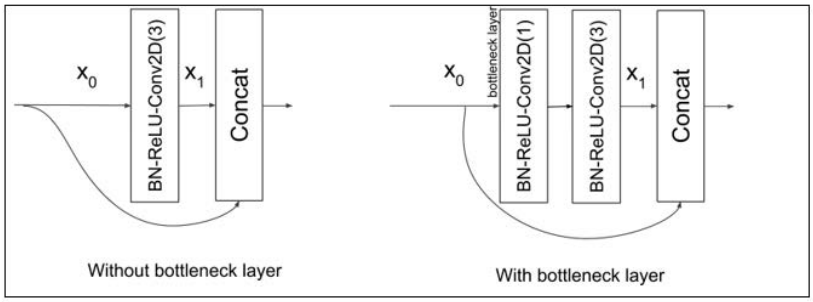

图 5.12: DenseNet 的 Dense 块中的一个层，有和没有瓶颈层 BN-ReLU-Conv2D（1）。为了清楚起见，我们会把内核大小作为 Conv2D 的一个参数。

然后，瓶颈层将 DenseNet 层修改为 BN-ReLU-Conv2D(1)-BN-ReLU-Conv2D(3)，而不仅仅是 BN-ReLU-Conv2D(3)。为了清楚起见，我们将内核大小作为 Conv2D 的一个参数。有了瓶颈层，每个 Conv2D(3) 都只处理 4k 个特征图，而不是 l 层的 $(l-1)\times k+k_0$。例如，对于 101 层的网络，最后一个 Conv2D(3) 的输入仍然是 $k=12$ 的 48 个特征图，而不是之前计算的 1224 个。为了解决特征图大小不匹配的问题，DenseNet 将一个深度网络划分为多个 Dense 块，这些块通过过渡层连接起来，如图 2.4.3 所示。在每个 Dense 块内，特征图的大小（即宽度和高度）将保持不变。

过渡层的作用是在两个 Dense 块之间从一个特征图大小过渡到一个较小的特征图大小。尺寸的减少通常为一半。这是由平均池化层完成的。例如，一个默认 pool_size=2 的 AveragePooling2D 将尺寸从（64，64，256）减少到（32，32，256）。过渡层的输入是前面 Dense 块中最后一个连接层的输出。

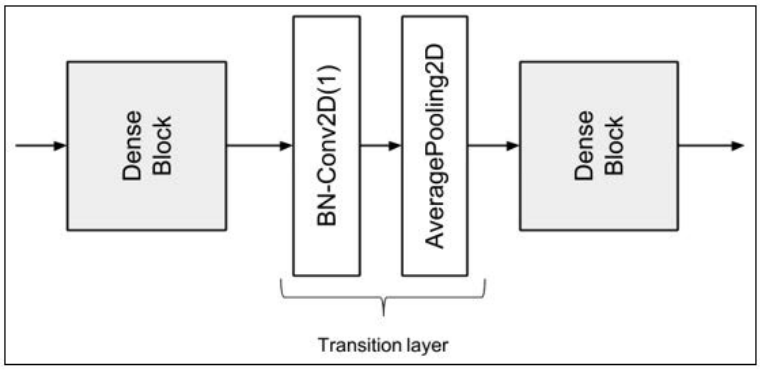

图 5.13: 通过从 1 增加扩张率，有效内核接收场的大小也会增加

然而，在特征图被传递到平均池之前，它们的数量将被减少一定的压缩系数，$0<\Theta<1$，使用 Conv2D（1）。DenseNet 在他们的实验中使用 $\Theta=0.5$。例如，如果前一个 Dense 块最后一次串联的输出是（64，64，512），那么在 Conv2D（1）之后，特征图的新维度将是（64，64，256）。当压缩和降维放在一起时，过渡层是由 BN-Conv2D(1)-AveragePooling2D 层组成。在实践中，批量归一化在卷积层之前。现在我们已经涵盖了 DenseNet 的重要概念。接下来，我们将在 tf.keras 中为 CIFAR10 数据集构建并验证一个 DenseNet-BC。

5.4.1. 为 CIFAR10 构建一个 100 层的 DenseNet-BC

We're now going to build a DenseNet-BC (Bottleneck-Compression) with 100 layers for the CIFAR10 dataset, using the design principles that we discussed above.

Table 2.4.1 shows the model configuration, while Figure 2.4.4 shows the model architecture. The listing shows us the partial Keras implementation of DenseNet-BC with 100 layers. We need to take note that we use RMSprop since it converges better than SGD or Adam when using DenseNet.

我们现在要为 CIFAR10 数据集建立一个有 100 层的 DenseNet-BC（瓶颈-压缩），使用我们上面讨论的设计原则。

表 2.4.1 显示了模型的配置，而图5.14显示了模型的架构。该列表向我们展示了具有 100 层的 DenseNet-BC 的部分 Keras 实现。我们需要注意的是，我们使用了 RMSprop，因为它在使用 DenseNet 时比 SGD 或 Adam 收敛得更好。

Table 2.4.1: DenseNet-BC with 100 layers for CIFAR10 classification

从配置到架构的转变。

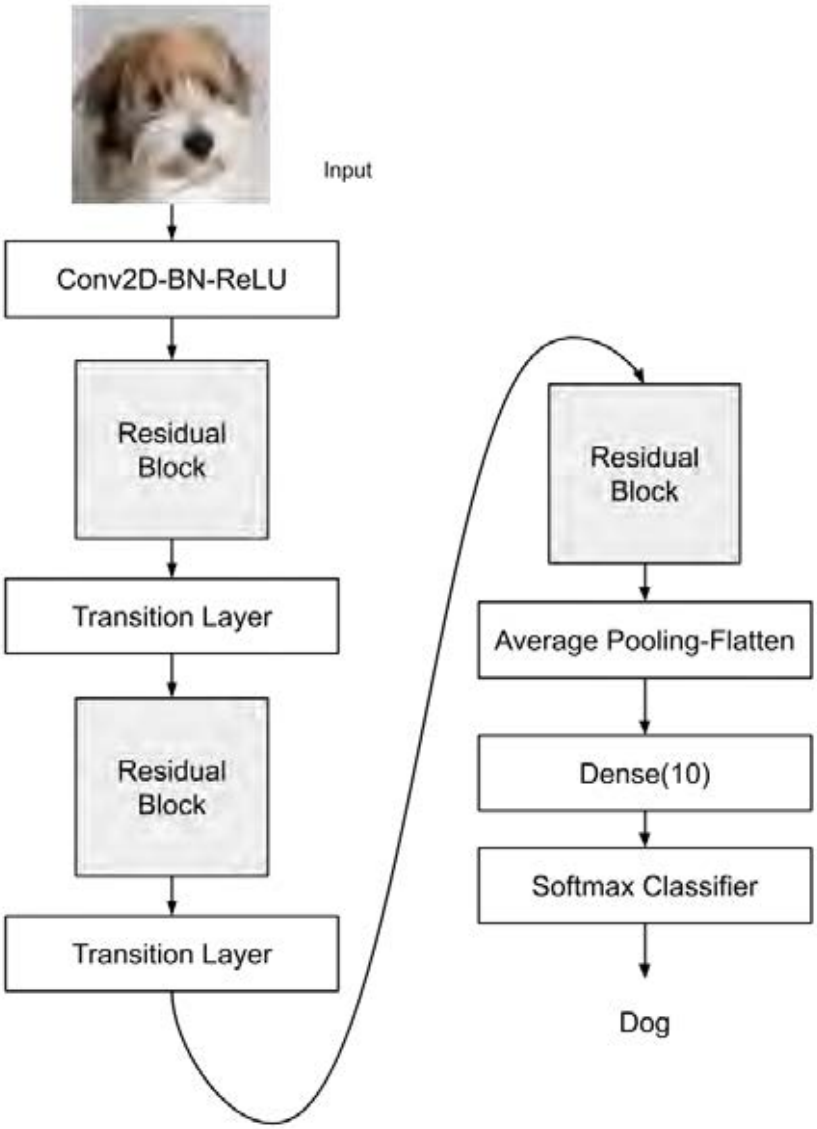

图 5.14: 用于 CIFAR10 分类的具有 100 层的 DenseNet-BC 的模型结构

下面的程序 2.4.1 是 DenseNet-BC 的部分 Keras 实现，有 100 层，如表 2.4.1 所示。

Listing 5.8: Evaluate how the model does on the test set

```
Listing 2.4.1: densenet-cifar10-2.4.1.py
    # start model definition
    # densenet CNNs (composite function) are made of BN-ReLU-Conv2D
    inputs = Input(shape=input_shape)
    x = BatchNormalization()(inputs)
    x = Activation('relu')(x)
    x = Conv2D(num_filters_bef_dense_block,
               kernel_size=3,
               padding='same',
               kernel_initializer='he_normal')(x)
```

```
    x = concatenate([inputs, x])
    # stack of dense blocks bridged by transition layers
    for i in range(num_dense_blocks):
        # a dense block is a stack of bottleneck layers
        for j in range(num_bottleneck_layers):
            y = BatchNormalization()(x)
            y = Activation('relu')(y)
            y = Conv2D(4 * growth_rate,
                       kernel_size=1,
                       padding='same',
                       kernel_initializer='he_normal')(y)
            if not data_augmentation:
                y = Dropout(0.2)(y)
            y = BatchNormalization()(y)
            y = Activation('relu')(y)
            y = Conv2D(growth_rate,
                       kernel_size=3,
                       padding='same',
                       kernel_initializer='he_normal')(y)
            if not data_augmentation:
                y = Dropout(0.2)(y)
            x = concatenate([x, y])
        # no transition layer after the last dense block
        if i == num_dense_blocks - 1:
continue
     # transition layer compresses num of feature maps and # reduces
    the size by 2
        num_filters_bef_dense_block += num_bottleneck_layers * growth_rate
        num_filters_bef_dense_block = int(num_filters_bef_dense_block *
    compression_factor)
        y = BatchNormalization()(x)
        y = Conv2D(num_filters_bef_dense_block,
                   kernel_size=1,
                   padding='same',
                   kernel_initializer='he_normal')(y)
        if not data_augmentation:
            y = Dropout(0.2)(y)
        x = AveragePooling2D()(y)
    # add classifier on top
    # after average pooling, size of feature map is 1 x 1
    x = AveragePooling2D(pool_size=8)(x)
    y = Flatten()(x)
    outputs = Dense(num_classes,
                    kernel_initializer='he_normal',
                    activation='softmax')(y)
    # instantiate and compile model
    # orig paper uses SGD but RMSprop works better for DenseNet
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(loss='categorical_crossentropy',
                  optimizer=RMSprop(1e-3),
                  metrics=['accuracy'])
    model.summary()
```

对 DenseNet 的 tf.keras 实现进行 200 个 epochs 的训练，实现了 93.74% 的准确率，而论文中报告的是 95.49%。使用了数据增强。我们将 ResNet v1/v2 中相同的回调函数用于 DenseNet。对于更深的层，必须使用 Python 代码上的表格改变 growth_rate 和 depth 变量。然而，按照论

文中的做法，在 190 或 250 的深度上训练网络将需要大量的时间。为了给我们一个训练时间的概念，每个 epoch 在 1060Ti GPU 上运行了大约一个小时。与 ResNet 类似，Keras 应用包也有针对 DenseNet 121 和更高的预训练模型。DenseNet 完成了我们对深度神经网络的讨论。与 ResNet 一起，这两个网络在许多下游任务中都是不可或缺的，或作为特征提取器网络。

5.5. 总结

在本章中，我们已经介绍了 Functional API 作为一种高级方法构建复杂的深度神经网络模型的 Tf.keras。我们还展示了如何使用 Functional API 来构建多输入单输出的 Y-Network。这个网络与单分支的 CNN 网络相比，能达到更好的精度。在本书的其余部分，我们会发现 Functional API 在构建更复杂、更高级的模型时是不可或缺的。例如，在下一章，Functional API 将使我们能够建立一个模块化的编码器、解码器和自动编码器。我们还花了大量的时间来探索两个重要的深度网络，ResNet 和 DenseNet。这两个网络不仅被用于分类，也被用于其他领域，如分割、检测、跟踪、生成和视觉语义理解。在第 11 章物体检测和第 12 章语义分割中，我们将使用 ResNet 进行物体检测和分割。我们需要记住，更重要的是，我们要更深入地理解 ResNet 和 DenseNet 中的模型设计决策，而不是仅仅遵循最初的实现。通过这种方式，我们就能将 ResNet 和 DenseNet 的关键概念用于我们的目的。

6. References 1. Kaiming He et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Proceedings of the IEEE international conference on computer vision, 2015

2. Kaiming He et al. Deep Residual Learning for Image Recognition. Proceedings of the IEEE conference on computer vision and pattern recognition, 2016a

3. Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. ICLR, 2015

4. Kaiming He et al. Identity Mappings in Deep Residual Networks. European Conference on Computer Vision. Springer International Publishing, 2016b

5. Gao Huang et al. Densely Connected Convolutional Networks. Proceedings of the IEEE conference on computer vision and pattern recognition, 2017

6. Saining Xie et al. Aggregated Residual Transformations for Deep Neural Networks. Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on. IEEE, 2017

7. Zagoruyko, Sergey, and Nikos Komodakis. ”Wide residual networks.” arXiv preprint arXiv:1605.07146 (2016).