第 5 章

计算视觉分析与应用

卷积神经网络是计算机视觉应用中几乎都在使用的一种深度学习模型。

5.1 从全连接到卷积

全连接网络其实和卷积网络是等价的，全连接层就可以转换为卷积层，只不过这个卷积层比较特殊，称为全卷积层，下面举一个简单的例子来说明全连接层如何转换为全卷积层。

由图 5-1 所示，假定要将一个 2×2×1 的特征图(feature map)通过全连接层输出为一个 4 维向量，图中的矩阵 $\boldsymbol{X}$ 即是这个 2×2×1 的特征图，向量 $\boldsymbol{Y}$ 就是输出的 4 维向量，全连接层即是将特征图由矩阵形式展开成向量形式，该向量即为全连接层的输入。

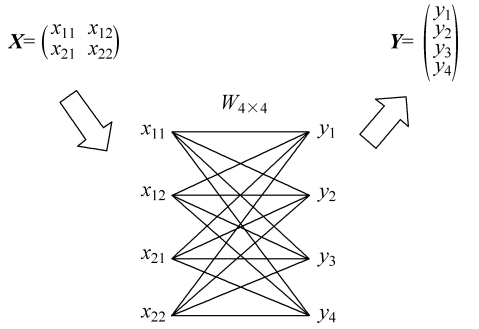


图 5-1 全连接层

如图 5-2 所示，全连接层的运算就是矩阵运算，输出向量 $\boldsymbol{Y}$ 就是由权重矩阵 $\boldsymbol{W}$ 乘展开成向量的 $\boldsymbol{X}'$，可以看到，对于每一个 y_i，都是由权重矩阵的第 i 行与 $\boldsymbol{X}'$对应元素相乘，这个相乘的过程和用权重矩阵的第 i 行所构成的卷积核去卷积 $\boldsymbol{X}$ 会产生一样的结果。

那么将 2×2×1 的特征图通过全连接层得到 4 维向量就相当于以全连接层中的权重矩阵中的 4 行向量所组成的 4 个卷积核去卷积 2×2×1 的特征图，如图 5-3 所示，此时的卷积核的大小就和特征图的大小一样，因此称为全卷积，全卷积最终得到 1×1×4 的

$$\boldsymbol{X}=\begin{pmatrix}x_{11} & x_{12}\\ x_{21} & x_{22}\end{pmatrix} \qquad \boldsymbol{Y}=\begin{pmatrix}y_1\\ y_2\\ y_3\\ y_4\end{pmatrix}$$

$$\boldsymbol{WX}'=\boldsymbol{Y}$$

$$\boldsymbol{W}=\begin{pmatrix}w_{11} & w_{12} & w_{13} & w_{14}\\ w_{21} & w_{22} & w_{23} & w_{24}\\ w_{31} & w_{32} & w_{33} & w_{34}\\ w_{41} & w_{42} & w_{43} & w_{44}\end{pmatrix} \qquad \boldsymbol{X}'=\begin{pmatrix}x_{11}\\ x_{12}\\ x_{21}\\ x_{22}\end{pmatrix}$$

$$y_1=x_{11}w_{11}+x_{12}w_{12}+x_{21}w_{13}+x_{22}w_{14}$$

$$y_1=\begin{pmatrix}w_{11} & w_{12}\\ w_{13} & w_{14}\end{pmatrix}\begin{pmatrix}x_{11} & x_{12}\\ x_{21} & x_{22}\end{pmatrix}$$

图 5-2　全连接层运算

矩阵，这和 4 维向量效果是一样的。

$$\boldsymbol{X}=\begin{pmatrix}x_{11} & x_{12}\\ x_{21} & x_{22}\end{pmatrix} \qquad \boldsymbol{Y}=\begin{pmatrix}y_1\\ y_2\\ y_3\\ y_4\end{pmatrix}$$

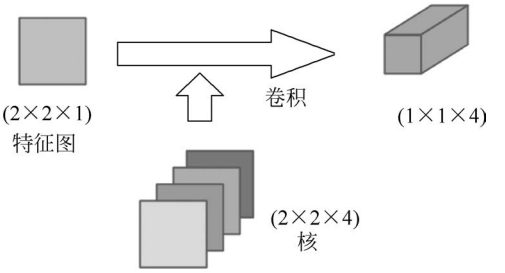


图 5-3　去卷积

5.2　卷积神经网络

卷积神经网络是多层感知机(MLP)的优化，其本质是一个多层感知机，成功的原因在于其所采用的局部连接和权值共享的方式：一方面减少了权值的数量使得网络易于优化；另一方面降低了模型的复杂度，也就是减小了过拟合的风险。

该优点在网络的输入为图像时表现得更为明显，使得图像可以直接作为网络的输入，避免了传统识别算法中复杂的特征提取和数据重建的过程，在二维图像的处理过程中有很大的优势，如网络能够自行抽取图像的特征，包括颜色、纹理、形状及图像的拓扑结构，在处理二维图像的问题上，特别是识别位移、缩放及其他形式扭曲不变性的应用上具有良好的健壮性和运算效率等。

5.2.1　卷积计算过程

卷积(convolution)计算的过程中：

(1) 卷积计算可被认为是一种有效提取图像特征的方法。

(2) 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。对每一个步长，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。

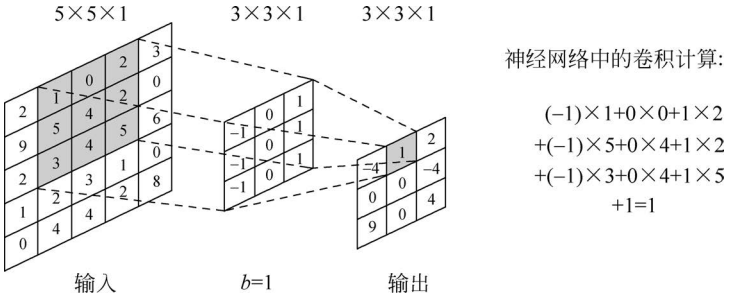


图 5-4　卷积计算结果

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和加操作，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。

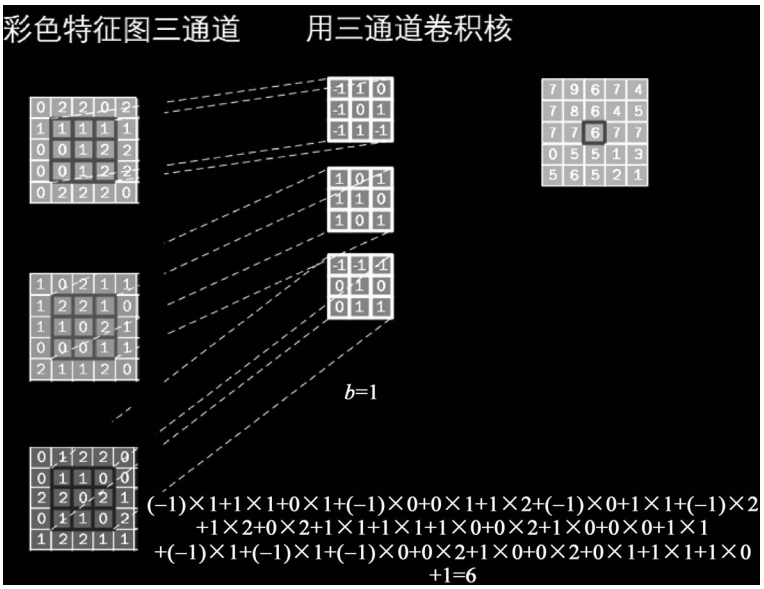


图 5-5　三通道卷积核

5.2.2　感受野

感受野(receptive field)是指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。图 5-6 为感受野示意图。

当卷积核的尺寸不同时，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，常用两层 3×3 卷积核来替换一层 5×5 卷积核的方法，如图 5-7 所示。

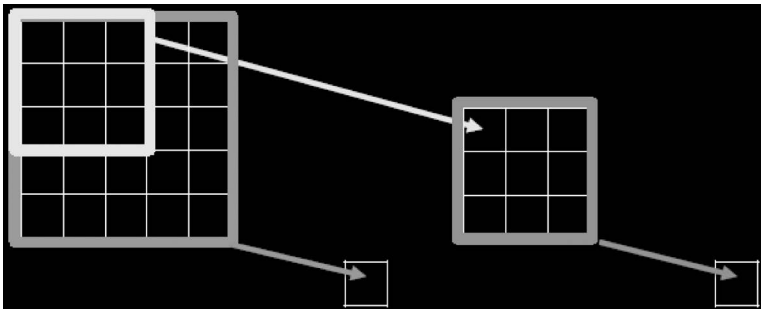


图 5-6　感受野示意图

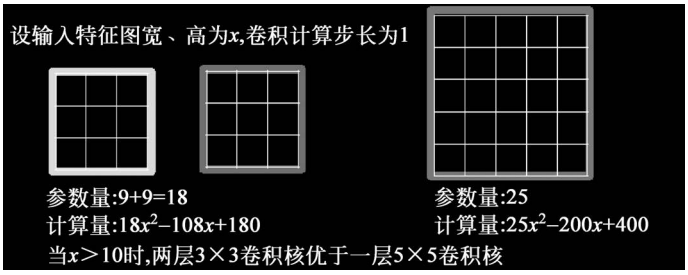


图 5-7　卷积核的替换

5.2.3　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的特征尺寸为 3×3。

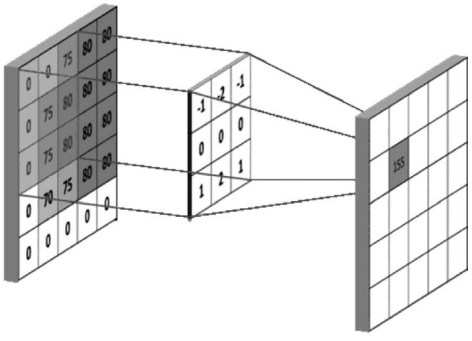


图 5-8　输出特征计算

5.2.4　全零填充

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(padding)。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。

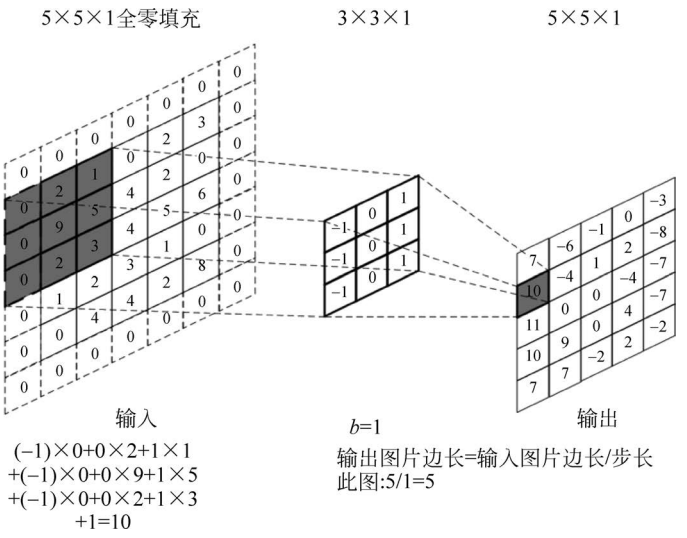


图 5-9　全零填充

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图 5-10 所示。

$$\text{padding}=\begin{cases}\text{'same'}, & \dfrac{\text{入长}}{\text{步长}}\ \text{(面积不变)}\\ \text{'valid'(不全零填充)}, & \dfrac{\text{入长}-\text{核长}+1}{\text{步长}}\ \text{(向上取整)}\end{cases}$$

图 5-10　输出特征尺寸大小

TensorFlow 描述卷积层：

```
tf.keras.layers.Conv2D(
filters = 卷积核个数
kernel_size = 卷积核尺寸, #正方形写核长,整数,或(核高 h,核宽 w)strides = 滑动步长,横纵向
#相同写步长,整数,或(纵向步长 h,横向步长 w),默认为 1
padding = 'same'或'valid', #使用全零填充是'same',不使用是'valid'(默认)
activation = 'ReLU' or 'sigmoid' or 'tanh' or 'softmax'等, #如有 BN 此处不写
input_shape = (高, 宽, 通道数) #输入特征图维度,可省略
)
```

对应的代码形式为：

```
model = tf.keras.models.Sequential([Conv2D(6, 5, padding = 'valid', activation = 'sigmoid'),
MaxPool2D(2, 2),
        Conv2D(6, (5, 5), padding = 'valid', activation = 'sigmoid'),MaxPool2D(2, (2, 2)),
        Conv2D(filters = 6, kernel_size = (5, 5),padding = 'valid', activation = 'sigmoid'),
MaxPool2D(pool_size = (2, 2), strides = 2),
        Flatten(),
        Dense(10, activation = 'softmax')
    ])
```

5.2.5　批标准化

标准化是指使数据符合均值为 0、标准差为 1 的分布；如果对一小批数据(batch)做

标准化处理即为批标准化(Batch Normalization,BN),效果如图 5-11 所示。

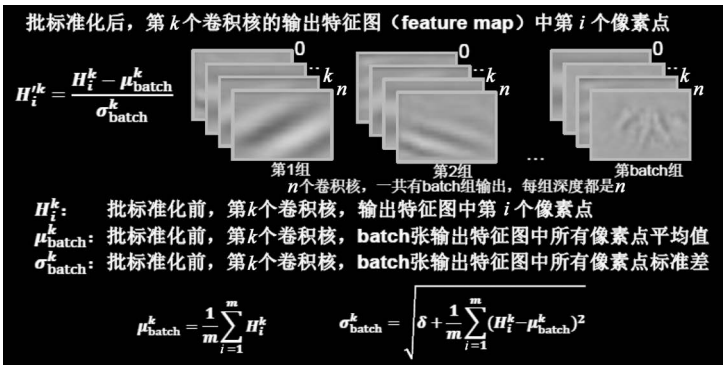


图 5-11　批标准化

BN 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 5-12 所示。

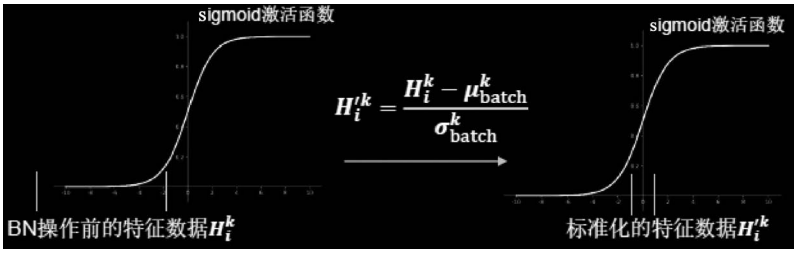


图 5-12　梯度消失

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,如图 5-13 所示。

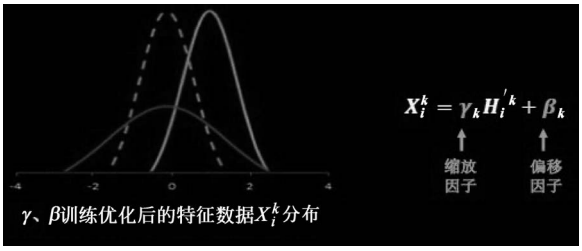


图 5-13　缩放和偏移

对应的代码形式为:

```
model = tf.keras.models.Sequential([Conv2D(filters=6, kernel_size=(5, 5), padding=
'same'),                                                    # 卷积层
    BatchNormalization(),                                   # BN层
    Activation('ReLU'),                                     # 激活层
    MaxPool2D(pool_size=(2, 2), strides=2, padding='same'),  # 池化层
    Dropout(0.2),                                           # dropout层
])
```

提示：BN 层位于卷积层之后，激活层之前。

5.2.6 池化

池化(pooling)用于减少特征数据量，最大池化可提取图片纹理，均值池化可保留背景特征，效果如图 5-14 所示。

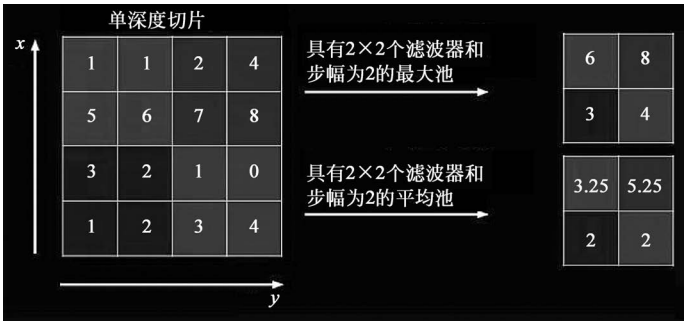


图 5-14 池化效果

Tensorflow 描述池化：

```
tf.keras.layers.MaxPool2D(
pool_size = size1                    # 正方形写核长整数，或(核高 h,核宽 w)
strides = poolstep                   # 步长整数，或(纵向步长 h,横向步长 w)
pool_sizepadding = 'valid'           # 取'valid'(默认) 或'same'(全零填充)值)

tf.keras.layers.AveragePooling2D(
pool_size = size2,                   # 正方形写核长整数，或(核高 h,核宽 w)
strides = poolstep                   # 步长整数，或(纵向步长 h,横向步长 w)
pool_sizepadding = 'valid'           # 取'valid'(默认) 或'same'(全零填充)值)
# 卷积层
model = tf.keras.models.Sequential([Conv2D(filters = 6, kernel_size = (5, 5), padding =
'same'),
    BatchNormalization(),                                          # BN 层
    Activation('relu'),                                            # 激活层
    MaxPool2D(pool_size = (2, 2), strides = 2, padding = 'same'),  # 池化层
    Dropout(0.2),                                                  # dropout 层
])
```

5.2.7 舍弃

在神经网络训练时，将一部分神经元按照一定概率从神经网络中暂时舍弃(dropout)。神经网络使用时，被舍弃的神经元恢复连接，效果如图 5-15 所示。

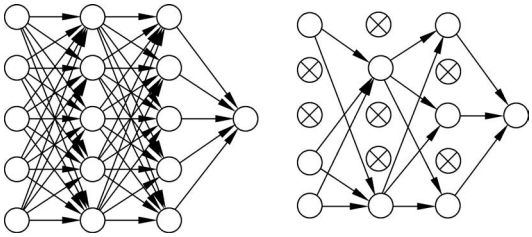


图 5-15 舍弃

对应的代码形式为：

```
model = tf.keras.models.Sequential([Conv2D(filters=6, kernel_size=(5, 5), padding=
'same'),                                                    # 卷积层
    BatchNormalization(),                                   # BN层
    Activation('ReLU'),                                     # 激活层
    MaxPool2D(pool_size=(2, 2), strides=2, padding='same'),  #池化层
    Dropout(0.2),                                           # dropout层
])
```

【例 5-1】 卷积神经网络识别手写数字。

```
'''模型的训练'''
from tensorflow.keras.datasets import mnist
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import losses
from tensorflow.keras import utils

if __name__=='__main__':
    (train_images,train_labels),(test_images,test_labels)=mnist.load_data()
    print(train_images.shape)#(60000, 28, 28)
    print(train_labels.shape)#(60000,)
    #准备训练数据
    train_images=train_images.reshape(train_images.shape[0],28,28,1)#(60000,28,28,1)
    #将图像数据归一化到0～1
    train_images=train_images.astype('float32')/255
    test_images=test_images.reshape(test_images.shape[0],28,28,1)
    test_images=test_images.astype('float32')/255
    #准备标签,标签变为one-hot型
    train_labels=utils.to_categorical(train_labels)#(60000, 10)
    test_labels=utils.to_categorical(test_labels)
    #将训练数据拿出1/5作为验证数据
    x_train=train_images[:48000]
    y_train=train_labels[:48000]
    x_val=train_images[48000:]
    y_val=train_labels[48000:]
    #创建网络模型
    model=models.Sequential()
    model.add(layers.Conv2D(32,(3,3),strides=(1,1),padding='valid',activation='ReLU',
input_shape=(28,28,1)))
    model.add(layers.MaxPooling2D(pool_size=(2,2),strides=None,padding='valid'))
    model.add(layers.Conv2D(64,(3,3),strides=(1,1),padding='valid',activation='ReLU'))
    model.add(layers.MaxPooling2D(pool_size=(2,2),strides=None,padding='valid'))
    model.add(layers.Conv2D(64,(3,3),strides=(1,1),padding='valid',activation='ReLU'))
```

```
    model.add(layers.Flatten())
    model.add(layers.Dense(64,activation='ReLU'))
    model.add(layers.Dense(10,activation='softmax'))

    #编译网络：优化器、损失函数、监控指标
    model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    model.summary()
    #拟合网络
    history=model.fit(x=x_train,y=y_train,batch_size=128,epochs=5,validation_data
=(x_val,y_val))
    print(history.history)
    #检查模型在测试数据上的性能
    test_loss,test_acc=model.evaluate(x=test_images,y=test_labels)
    print(test_acc)
    #保存模型
    model.save('mnist_cnn.h5')
```

运行程序，输出如下：

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
mnist.npz
11490434/11490434 [==============================] - 55s 5us/step
(60000, 28, 28)
(60000,)
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
max_pooling2d (MaxPooling2D  (None, 13, 13, 32)        0
)
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)          0
2D)
conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928
flatten (Flatten)            (None, 576)               0
dense (Dense)                (None, 64)                36928
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
Epoch 1/5
375/375 [==============================] - 27s 70ms/step - loss: 0.2694 -
accuracy: 0.9153 - val_loss: 0.0806 - val_accuracy: 0.9753
...
```

```
{ ' loss ' : [ 0. 2693770229816437, 0. 06094488874077797, 0. 04082140699028969,
0.03071589395403862, 0. 022480076178908348 ], ' accuracy ' : [ 0. 9152708053588867,
0.9813541769981384, 0.9871875047683716, 0.9898333549499512, 0.9929583072662354], 'val_
loss ' : [ 0. 08064586669206619, 0. 04999165236949921, 0. 040647201240062714,
0.04069573059678078, 0. 04543827474117279 ], ' val _ accuracy ' : [ 0. 9752500057220459,
0.9852499961853027, 0.9882500171661377, 0.9879999756813049, 0.9869166612625122]}
313/313 [ ============================== ] - 2s 5ms/step - loss: 0.0374 -
accuracy: 0.9881
0.988099992275238
```

识别的数字如图 5-16 所示。

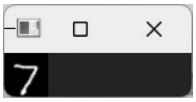


图 5-16　数字识别

```
'''网络模型的调用'''
from tensorflow.keras.datasets import mnist
from tensorflow.keras import models
import cv2
import numpy as np

if __name__ == '__main__':
    (train_images,train_labels),(test_images,test_labels) = mnist.load_data()
    #直接载入的图像范围是 0～255
    print(train_images[0].shape) #(28, 28)
    #OpenCV 的图像数据是(rows,cols,channels)
    test_img = test_images[0].reshape(28,28,1) #(28, 28, 1)
    print(test_img.shape)
    cv2.imshow('test',test_img)
    cv2.waitKey(0)
    #载入模型
    network = models.load_model('mnist_cnn.h5')
    network.summary()
    #模型中载入的图像数据是批量的,必须包含 batch,即使 batch 为 1
    test_img = test_img.reshape((1,) + test_img.shape) #(1,28,28,1)
    #归一化为 0～1
    test_img = test_img.astype('float32')/255
    #进行预测
    output = network.predict(test_img) #(batch,10)
    #取出轴 1 的最大值
    output = output.argmax(axis = 1)
    print(output)
```

运行程序,输出如下:

```
(28, 28)
(28, 28, 1)
```

5.3 现代经典网络

卷积和池化的随机组合赋予了 CNN 很大的灵活性，因此也诞生了很多耳熟能详的经典网络，LeNet、AlexNet、VGGNet、NiN、Google Inception Net、ResNet、DenseNet 这几种网络在深度和复杂度方面依次递增。下面将分别介绍这几种网络原理、架构以及实现。

5.3.1 LeNet 网络

LeNet 网络诞生于 1994 年，是最早的深层卷积神经网络之一，并且推动了深度学习的发展。它是第一个成功大规模应用在手写数字识别问题的卷积神经网络，在 MNIST 数据集中的正确率可以高达 99.2%。

图 5-17 为 LeNet-5 网络工作的原理图。

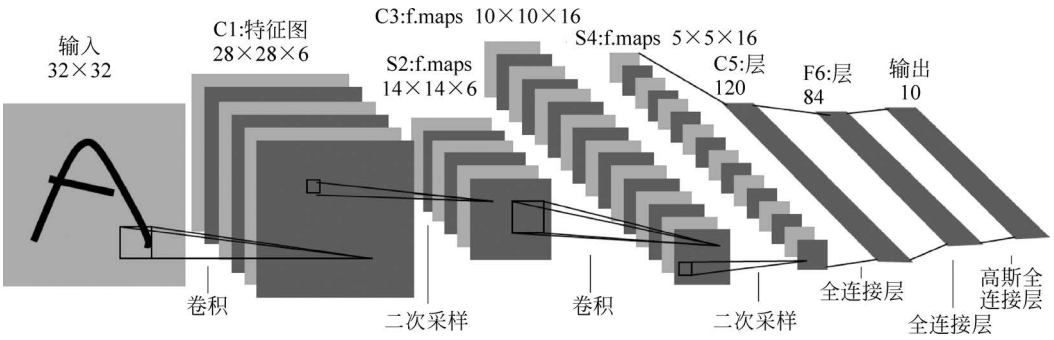


图 5-17 LeNet-5 网络工作的原理图

LeNet-5 网络是针对灰度图进行训练的，输入图像大小为 32×32×1，不包含输入层的情况下共有 7 层，每层都包含可训练参数(连接权重)，具体如下。

(1) C1 层是一个卷积层(通过卷积运算，可以使原信号特征增强，并且降低噪声)。第一层使用 5×5 大小的滤波器 6 个，步长 s=1，padding=0，输出得到的特征图大小为 28×28×6，一共有 156 个可训练参数(每个滤波器 5×5=25 个 unit 参数和 1 个 bias 参数，一共 6 个滤波器，共(5×5+1)×6=156 个参数)，共 156×(28×28)=122 304 个连接。

(2) S2 层是一个下采样层(平均池化层)，利用图像局部相关性的原理，对图像进行子抽样，可以：

- 减少数据处理量，同时保留有用信息；
- 降低网络训练参数及模型的过拟合程度。

第二层使用 2×2 大小的滤波器，步长 s=2，padding=0，输出得到的特征图大小为 14 146。池化层只有一组超参数 f 和 s，没有需要学习的参数。

(3) C3 层是一个卷积层。第三层使用 5×5 大小的滤波器 16 个，步长 s=1，padding=0，输出得到的特征图大小为 10×10×16。C3 有 416 个可训练参数。

（4）S4 层是一个下采样层（平均池化层）。第四层使用 2×2 大小的滤波器，步长 s=2，padding=0，输出得到的特征图大小为 5×5×16。

（5）F5 层是一个全连接层，有 120 个单元，是由上一层输出经过 120 个大小为 5×5 的卷积核得到的，没有 padding，步长 s=1，上一层的 16 个特征图都连接到该层的每一个单元，所以这里相当于一个全连接层。

（6）F6 层是一个全连接层，有 84 个单元，与上一层构成全连接的关系，再经由 sigmoid 激活函数传到输出层。

（7）输出层也是一个全连接层，共有 10 个单元，对应 0～9 共 10 个数字。本层单元计算的是径向基函数 $y_i=\sum_j (x-w_{i,j})^2$，RBF 的计算与第 i 个数字的比特图编码有关，对于第 i 个单元，y_i 的值越接近 0，则表示越接近第 i 个数字的比特编码，即识别当前输入的结果为第 i 个数字。

LeNet-5 网络基于 PyTorch 的网络实现：

```
import torch
import torch.nn as nn
import torch.optim as optim
import time
#net
class Flatten(torch.nn.Module):                          #展平操作
     def forward(self, x):
          return x.view(x.shape[0], -1)

class Reshape(torch.nn.Module):                          #图像重构
     def forward(self, x):
          return x.view(-1,1,32,32)                      #(B x C x H x W),通道数在第二维度

net = torch.nn.Sequential(
     Reshape(),
     nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
                                                         #b*1*32*32 =>b*6*28*28
     nn.Sigmoid(),
     nn.AvgPool2d(kernel_size=2,stride=2),               #b*6*28*28 =>b*6*14*14
     nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5),
                                                         #b*6*14*14 =>b*16*10*10
     nn.Sigmoid(),
     nn.AvgPool2d(kernel_size=2,stride=2),               #b*16*10*10 =>b*16*5*5
     Flatten(),                                          #b*16*5*5 =>b*400
     nn.Linear(in_features=16*5*5, out_features=120),
     nn.Sigmoid(),
     nn.Linear(120, 84),
     nn.Sigmoid(),
     nn.Linear(84, 10)
)

X = torch.randn(size=(1,1,32,32), dtype = torch.float32)
for layer in net:
```

```
        X = layer(X)
        print(layer.__class__.__name__,'output shape: \t',X.shape)
```

运行程序，输出如下：

```
Reshape output shape:       torch.Size([1, 1, 32, 32])
Conv2d output shape:        torch.Size([1, 6, 28, 28])
Sigmoid output shape:       torch.Size([1, 6, 28, 28])
AvgPool2d output shape:     torch.Size([1, 6, 14, 14])
Conv2d output shape:        torch.Size([1, 16, 10, 10])
Sigmoid output shape:       torch.Size([1, 16, 10, 10])
AvgPool2d output shape:     torch.Size([1, 16, 5, 5])
Flatten output shape:       torch.Size([1, 400])
Linear output shape:        torch.Size([1, 120])
Sigmoid output shape:       torch.Size([1, 120])
Linear output shape:        torch.Size([1, 84])
Sigmoid output shape:       torch.Size([1, 84])
Linear output shape:        torch.Size([1, 10])
```

5.3.2 AlexNet 网络

AlexNet 网络由 5 个卷积层和 3 个池化层以及 3 个全连接层构成。AlexNet 网络跟 LeNet 网络结构类似，但使用了更多的卷积层和更大的参数空间来拟合大规模数据集 ImageNet。它是浅层神经网络和深度神经网络的分界线，其结构如图 5-18 所示。

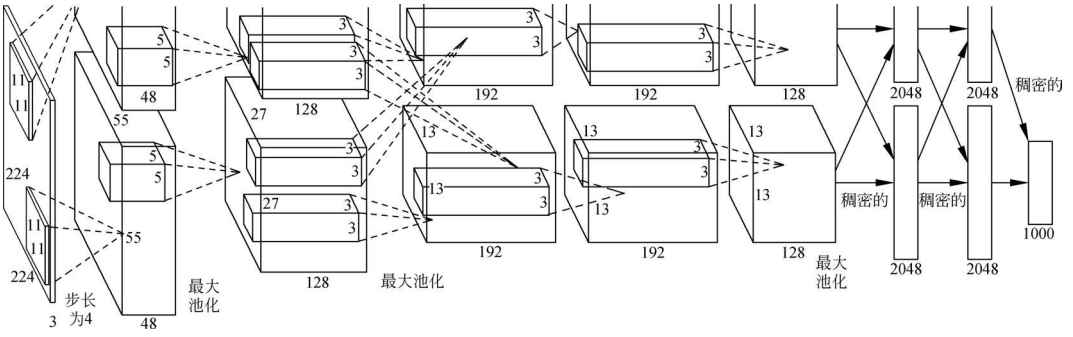


图 5-18　AlexNet 网络结构

图 5-17 中的输入是 224×224，所以使用 227×227 作为输入，则(227－11)/4＝55。网络包含 8 个带权重的层，前 5 层是卷积层，剩下的 3 层是全连接层。最后一层全连接层的输出是 1000 维 softmax 的输入，softmax 会产生 1000 类标签的分布。

(1) 卷积层 C1，该层的处理流程是：卷积→ReLU→池化→局部响应归一化。

- 卷积，输入为 227×227，使用 96 个 11×11×3 的卷积核，得到的 FeatureMap(特征图)为 55×55×96。
- ReLU，将卷积层输出的 FeatureMap 输入到 ReLU 函数中。
- 池化，使用 3×3、步长为 2 的池化单元(重叠池化，步长小于池化单元的宽度)，输出为 27×27×96((55－3)/2＋1＝27)。

- 局部响应归一化，使用 $k=2, n=5, \alpha=10-4, \beta=0.75$ 进行局部归一化，输出为 27×27×96，输出分为 2 组，每组的大小为 27×27×48。

(2) 卷积层 C2，该层的处理流程是：卷积→ReLU→池化→局部响应归一化。

- 卷积，输入是 2 组 27×27×48。使用 2 组，每组 128 个大小为 5×5×48 的卷积核，并做了边缘填充，padding=2，卷积的步长为 1，则输出的 FeatureMap 为 2 组，每组的大小为(27+2×2−5)/1+1=27。
- ReLU，将卷积层输出的 FeatureMap 输入到 ReLU 函数中。
- 池化运算的尺寸为 3×3，步长为 2，池化后图像的尺寸为(27−3)/2+1=13，输出为 13×13×256。
- 局部响应归一化，使用 $k=2, n=5, \alpha=10-4, \beta=0.75$ 进行局部归一化，输出仍然为 13×13×256，输出分为 2 组，每组的大小为 13×13×128。

(3) 卷积层 C3，该层的处理流程是：卷积→ReLU。

- 卷积，输入是 13×13×256，使用 2 组共 384 个大小为 3×3×256 的卷积核，做了边缘填充，padding=1，卷积的步长为 1。
- ReLU，将卷积层输出的 FeatureMap 输入到 ReLU 函数中。

(4) 卷积层 C4，该层的处理流程是：卷积→ReLU。

- 卷积，输入是 13×13×384，分为 2 组，每组为 13×13×192。使用 2 组，每组 192 个大小为 3×3×192 的卷积核，做了边缘填充，padding=1，卷积的步长为 1，则输出的 FeatureMap 分为 2 组，每组的大小为 13×13×192。
- ReLU，将卷积层输出的 FeatureMap 输入到 ReLU 函数中。

(5) 卷积层 C5，该层处理流程为：卷积→ReLU→池化。

- 卷积，输入为 13×13×384，分为 2 组，每组为 13×13×192。使用 2 组，每组为 128 个大小为 3×3×192 的卷积核，做了边缘填充，padding=1，卷积的步长为 1，则输出的 FeatureMap 为 13×13×256。
- ReLU，将卷积层输出的 FeatureMap 输入到 ReLU 函数中。
- 池化，池化运算的尺寸为 3×3，步长为 2，池化后图像的尺寸为(13−3)/2+1=6，即池化后的输出为 6×6×256。

(6) 全连接层 FC6，该层的流程为：(卷积)全连接→ReLU→dropout。

- (卷积)全连接：输入为 6×6×256，该层有 4096 个卷积核，每个卷积核的大小为 6×6×256。由于卷积核的尺寸刚好与待处理特征图(输入)的尺寸相同，即卷积核中的每个系数只与特征图(输入)尺寸的一个像素值相乘，并一一对应，因此，该层被称为全连接层。由于卷积核与特征图的尺寸相同，卷积运算后只有一个值，因此，卷积后的像素层尺寸为 4096×1×1，即有 4096 个神经元。
- ReLU，这 4096 个运算结果通过 ReLU 激活函数生成 4096 个值。
- dropout，抑制过拟合，随机地断开某些神经元的连接或者不激活某些神经元。

(7) 全连接层 FC7，该层流程为：全连接→ReLU→dropout。

- 全连接，输入为 4096 的向量。
- ReLU，这 4096 个运算结果通过 ReLU 激活函数生成 4096 个值。

• dropout，抑制过拟合，随机地断开某些神经元的连接或者不激活某些神经元。

(8) 输出层。

第七层输出的 4096 个数据与第八层的 1000 个神经元进行全连接，经过训练后输出 1000 个 float 型的值，这就是预测结果。

AlexNet 网络基于 PyTorch 的网络实现：

```
import time
import torch
from torch import nn, optim
import torchvision
import numpy as np
import sys
import os
import torch.nn.functional as F

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 96, 11, 4), # in_channels, out_channels, kernel_size,
                                     # stride, padding
              nn.ReLU(),
              nn.MaxPool2d(3, 2), # kernel_size, stride
              # 减小卷积窗口，使用填充为 2 来使得输入与输出的高和宽一致，且增大输出通
              # 道数
              nn.Conv2d(96, 256, 5, 1, 2),
              nn.ReLU(),
              nn.MaxPool2d(3, 2),
              # 连续 3 个卷积层，且使用更小的卷积窗口。除了最后的卷积层外，进一步增大
              # 了输出通道数
              # 前两个卷积层后不使用池化层来减小输入的高和宽
              nn.Conv2d(256, 384, 3, 1, 1),
              nn.ReLU(),
              nn.Conv2d(384, 384, 3, 1, 1),
              nn.ReLU(),
              nn.Conv2d(384, 256, 3, 1, 1),
              nn.ReLU(),
              nn.MaxPool2d(3, 2)
        )
         # 此处全连接层的输出个数比 LeNet 中的大数倍。使用 dropout 层来缓解过拟合
        self.fc = nn.Sequential(
              nn.Linear(256 * 5 * 5, 4096),
              nn.ReLU(),
              nn.Dropout(0.5),
              nn.Linear(4096, 4096),
              nn.ReLU(),
              nn.Dropout(0.5),
              nn.Linear(4096, 1000),
        )
```

```
    def forward(self, img):
        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
        return output
net = AlexNet()
print(net)
```

运行程序，输出如下：

```
AlexNet(
  (conv): Sequential(
    (0): Conv2d(1, 96, kernel_size=(11, 11), stride=(4, 4))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(96, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(256, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU()
    (10): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU()
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=
False)
  )
  (fc): Sequential(
    (0): Linear(in_features=6400, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

5.3.3　VGGNet 网络

VGGNet 是牛津大学计算机视觉组(Visual Geometry Group)和 Google DeepMind 公司的研究员一起研发的深度卷积神经网络。

VGGNet 探索了卷积神经网络的深度与其性能之间的关系，通过反复地堆叠 3×3 的小型卷积核和 2×2 的最大池化层，构建了 16～19 层深度的卷积神经网络，整个网络结构简洁，都使用同样大小的卷积核尺寸(3×3)和最大池化尺寸(2×2)。VGGNet 的扩展性很强，迁移到其他图片数据上的泛化性很好，因此，目前为止，也常被用来抽取图像的特征，被广泛用于其他很多地方。

VGGNet 网络中全部使用了 3×3 的卷积核和 2×2 的池化核，通过不断加深网络结构来提升性能。图 5-19 所示为 VGGNet 各级别的网络结构和每一级别的参数量，从 11

层的网络一直到 19 层的网络都有详尽的性能测试。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input(224×224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
softmax					

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

图 5-19　VGGNet 各级别的网络结构和每一级别的参数量

A 网络(11 层)有 8 个卷积层和 3 个全连接层,E 网络(19 层)有 16 个卷积层和 3 个全连接层,卷积层宽度(通道数)从 64 到 512,每经过一次池化操作,扩大一倍。

1. VGGNet 网络结构

VGGNet 网络结构主要表现在:

(1) 输入:训练时输入为 224×224 大小的 RGB 图像;

(2) 预处理:在训练集中的每个像素减去 RGB 的均值;

(3) 卷积核:3×3 大小的卷积核,有的地方使用 1×1 的卷积,这种 1×1 的卷积可以被看作对输入通道的线性变换;

(4) 步长:步长为 1;

(5) 填充:填充 1 像素;

(6) 池化层:共有 5 层,在一部分卷积层之后,连接的最大池化的窗口是 2×2,步长为 2;

(7) 全连接层:前两个全连接层均有 4096 个通道,第三个全连接层有 1000 个通道,用来分类,所有网络的全连接层配置相同;

(8) 激活函数:ReLU;

(9) 不使用 LRN,这种标准化并不能带来很大的提升,反而会导致更多的内存消耗和计算时间。

2. 与 AlexNet 的对比

VGGNet 与 AlexNet 对比主要的变化有:

(1) LRN 层作用不大,还耗时,抛弃;

(2) 网络越深,效果越好;

(3) 卷积核使用更小的卷积核,如 3×3。

VGGNet 虽然比 AlexNet 网络层数多,且每轮训练时间会比 AlexNet 更长,但是因为更深的网络和更小的卷积核带来的隐式正则化结果,需要的收敛的迭代次数减少了许多。

3. VGGNet 实现

VGGNet 网络基于 PyTorch 的网络实现如下。

(1) 导入模块。

```
import torch.nn as nn
import torch

__all__ = [
     'VGG', 'vgg11', 'vgg11_bn', 'vgg13', 'vgg13_bn', 'vgg16', 'vgg16_bn',
     'vgg19_bn', 'vgg19',
]
model_urls = {
     'vgg11': 'https://download.pytorch.org/models/vgg11-bbd30ac9.pth',
     'vgg13': 'https://download.pytorch.org/models/vgg13-c768596a.pth',
     'vgg16': 'https://download.pytorch.org/models/vgg16-397923af.pth',
     'vgg19': 'https://download.pytorch.org/models/vgg19-dcbb9e9d.pth',
     'vgg11_bn': 'https://download.pytorch.org/models/vgg11_bn-6002323d.pth',
     'vgg13_bn': 'https://download.pytorch.org/models/vgg13_bn-abd245e5.pth',
     'vgg16_bn': 'https://download.pytorch.org/models/vgg16_bn-6c64b313.pth',
     'vgg19_bn': 'https://download.pytorch.org/models/vgg19_bn-c79401a0.pth',
}
```

(2) 定义分类网络结构。

```
class VGG(nn.Module):
    #定义初始化函数
    def __init__(self, features, num_classes=1000, init_weights=True):
        super(VGG, self).__init__()
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
              nn.Dropout(0.4),
              nn.Linear(512 * 7 * 7, 4096),
              nn.ReLU6(True),
              nn.Dropout(0.4),
              nn.Linear(4096, 2048),
              nn.ReLU6(True),
```

```
                nn.Dropout(0.4),
                nn.Linear(2048, num_classes),

            )

            if init_weights:
                self._initialize_weights()

        #定义前向传播函数
        def forward(self, x):
            x = self.features(x)
            x = self.avgpool(x)
            x = x.view(x.size(0), -1)
            x = self.classifier(x)
            return x

        #定义初始化权重函数
        def _initialize_weights(self):
            for m in self.modules():
                if isinstance(m, nn.Conv2d):
                    nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity=
'ReLU')
                    if m.bias is not None:
                        nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.BatchNorm2d):
                    nn.init.constant_(m.weight, 1)
                    nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight, 0, 0.01)
                    nn.init.constant_(m.bias, 0)
```

（3）定义提取特征网络结构函数。

```
def make_layers(cfg: list, batch_norm=False):
    layers = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
cfg = {
    'A0': [64, 'M', 128, 'M', 256, 256, 'M'],
    'A1': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M'],
    'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
```

```
    'B': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
   'D': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512,
'M'],
    'E': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512,
512, 512, 512, 'M'],
}
```

（4）定义实例化给定的配置模型函数。

```
def vgg7(**kwargs):
    model = VGG(make_layers(cfg['A0']), **kwargs)
    return model

def vgg7_bn(**kwargs):
    model = VGG(make_layers(cfg['A0'], batch_norm=True), **kwargs)
    return model

def vgg9(**kwargs):
    model = VGG(make_layers(cfg['A1']), **kwargs)
    return model

def vgg9_bn(**kwargs):
    model = VGG(make_layers(cfg['A1'], batch_norm=True), **kwargs)
    return model

def vgg11(**kwargs):
    model = VGG(make_layers(cfg['A']), **kwargs)
    return model

def vgg11_bn(**kwargs):
    model = VGG(make_layers(cfg['A'], batch_norm=True), **kwargs)
    return model

def vgg13(**kwargs):
    model = VGG(make_layers(cfg['B']), **kwargs)
    return model

def vgg13_bn(**kwargs):
    model = VGG(make_layers(cfg['B'], batch_norm=True), **kwargs)
    return model

def vgg16(**kwargs):
    model = VGG(make_layers(cfg['D']), **kwargs)
    return model

def vgg16_bn(**kwargs):
    model = VGG(make_layers(cfg['D'], batch_norm=True), **kwargs)
    return model

def vgg19(**kwargs):
    model = VGG(make_layers(cfg['E']), **kwargs)
```

```
    return model

def vgg19_bn(**kwargs):
    model = VGG(make_layers(cfg['E'], batch_norm=True), **kwargs)
    return model

if __name__ == '__main__':
    # 'VGG', 'vgg11', 'vgg11_bn', 'vgg13', 'vgg13_bn', 'vgg16', 'vgg16_bn', 'vgg19_bn', 'vgg19'
    # Example
    net13 = vgg13_bn()
    print(net13)
```

运行程序,输出如下:

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
True)
    (2): ReLU(inplace=True)
...
(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Dropout(p=0.4, inplace=False)
    (1): Linear(in_features=25088, out_features=4096, bias=True)
    (2): ReLU6(inplace=True)
    (3): Dropout(p=0.4, inplace=False)
    (4): Linear(in_features=4096, out_features=2048, bias=True)
    (5): ReLU6(inplace=True)
    (6): Dropout(p=0.4, inplace=False)
    (7): Linear(in_features=2048, out_features=1000, bias=True)
  )
)
```

5.3.4 NiN

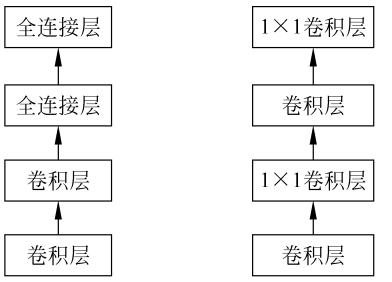
NiN(Network in Network)改进了传统的CNN,采用了少量参数就取得了超过AlexNet的性能,AlexNet网络参数大小是230M,NiN只需要29M,此模型后来被Inception与ResNet等所借鉴。关于NiN有如下两个很重要的观点。

(1) 1×1卷积层中可以把通道当作特征,高和宽上的每个元素相当于样本。因此,NiN使用1×1卷积层来替代全连接层,从而使空间信息能够自然传递到后面的层中(可以实现多个特征图的线性组合,实现跨通道的信息整合的功效,如图5-20所示)。

(2) NiN块是NiN中的基础块。它由一个卷积层加两个充当全连接层的1×1卷积层串联而成。其中第一个卷积层的超参数可以自行设置,而第二个和第三个卷积层的超参数一般是固定的。

完整的NiN结构如图5-21所示。

下面使用PyTorch实现NiN,并使用CIFAR 10数据集进行训练和测试。



(a) AlexNet和VGG结构　　(b) NiN结构局部

图 5-20　跨通道的信息整合

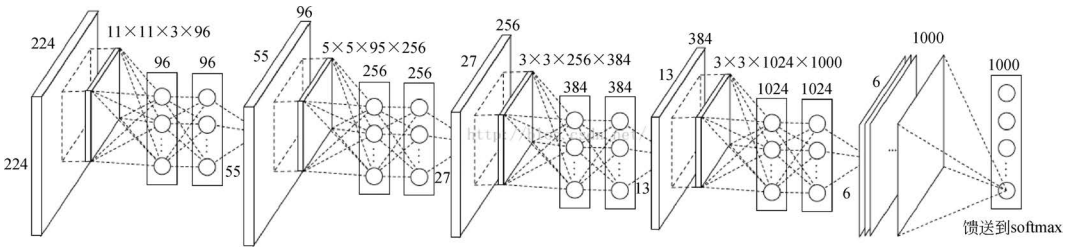


图 5-21　完整的 NiN 结构

（1）导入库。

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

（2）数据预处理(在线下载数据)。

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_
workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_
workers=2)
```

（3）定义 NiN。

```
class NiN(nn.Module):
    def __init__(self):
        super(NiN, self).__init__()
        self.conv1 = nn.Conv2d(3, 192, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(192, 160, kernel_size=1)
        self.conv3 = nn.Conv2d(160, 96, kernel_size=1)
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.dropout1 = nn.Dropout2d(p=0.5)
        self.conv4 = nn.Conv2d(96, 192, kernel_size=5, padding=2)
        self.conv5 = nn.Conv2d(192, 192, kernel_size=1)
        self.conv6 = nn.Conv2d(192, 192, kernel_size=1)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.dropout2 = nn.Dropout2d(p=0.5)
        self.conv7 = nn.Conv2d(192, 192, kernel_size=3, padding=1)
        self.conv8 = nn.Conv2d(192, 192, kernel_size=1)
        self.conv9 = nn.Conv2d(192, 10, kernel_size=1)
        self.pool3 = nn.AvgPool2d(kernel_size=8, stride=1)

    def forward(self, x):
        x = self.conv1(x)
        x = nn.functional.relu(x)
        x = self.conv2(x)
        x = nn.functional.relu(x)
        x = self.conv3(x)
        x = nn.functional.relu(x)
        x = self.pool1(x)
        x = self.dropout1(x)
        x = self.conv4(x)
        x = nn.functional.relu(x)
        x = self.conv5(x)
        x = nn.functional.relu(x)
        x = self.conv6(x)
        x = nn.functional.relu(x)
        x = self.pool2(x)
        x = self.dropout2(x)
        x = self.conv7(x)
        x = nn.functional.relu(x)
        x = self.conv8(x)
        x = nn.functional.relu(x)
        x = self.conv9(x)
        x = self.pool3(x)
        x = x.view(-1, 10)
        return x
net = NiN()
```

（4）定义损失函数和优化器。

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
```

（5）训练网络。

```
for epoch in range(100):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 100 == 99:
            print('[ % d, % 5d] loss: % .3f' %
                  (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0
```

（6）测试网络。

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: % d % % ' % (
    100 * correct / total))
```

首先，假设有一个输入张量 X，其形状为 $C_{\text{in}}\times H\times W$，其中 C_{in} 表示输入通道数，H 和 W 分别表示输入的高度和宽度。对输入进行卷积操作，得到一个输出张量 Y，其形状为 $C_{\text{out}}\times H\times W$，其中 C_{out} 表示输出通道数。

传统的卷积操作是使用一个大小为 $C_{\text{in}}\times C_{\text{out}}\times k\times k$ 的卷积核对输入进行卷积操作，其中 k 表示卷积核的大小。但是，NiN 引入了 1×1 卷积，可以使用一个大小为 $1\times 1\times C_{\text{in}}\times C_{\text{out}}$ 的卷积核来代替传统的卷积操作。

接下来推导 1×1 卷积的计算过程。假设使用一个大小为 $1\times 1\times C_{\text{in}}\times C_{\text{out}}$ 的卷积核 K，对输入张量 X 进行卷积操作，得到输出张量 Y，则 1×1 卷积的计算公式为

$$Y_{i,j,l}=\sum_{c=1}^{C_{\text{in}}}X_{i,j,c}K_{c,l}$$

其中，i 和 j 分别表示输出张量 $\boldsymbol{Y}$ 的高度和宽度，l 表示输出张量 $\boldsymbol{Y}$ 的通道数。

上式可用矩阵乘法的形式表示：

$$\boldsymbol{Y}_{i,j}=\boldsymbol{X}_{i,j}\boldsymbol{W}$$

其中，$\boldsymbol{X}_{i,j}$ 表示输入张量 $\boldsymbol{X}$ 在 (i,j) 位置上的特征向量，$\boldsymbol{W}$ 表示大小为 $C_{\text{in}}\times C_{\text{out}}$ 的卷积

核 K，$\boldsymbol{Y}_{i,j}$ 表示输出张量 $\boldsymbol{Y}$ 在(i,j)位置上的特征向量。这样就可以使用 NiN 中的 1×1 卷积对输入进行卷积操作，得到输出张量。

（7）使用 PyTorch 实现该网络。

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class NiN(nn.Module):
    def __init__(self):
        super(NiN, self).__init__()
        self.conv1 = nn.Conv2d(3, 192, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(192, 160, kernel_size=1)
        self.conv3 = nn.Conv2d(160, 96, kernel_size=1)
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.dropout1 = nn.Dropout(p=0.5)
        self.conv4 = nn.Conv2d(96, 192, kernel_size=5, padding=2)
        self.conv5 = nn.Conv2d(192, 192, kernel_size=1)
        self.conv6 = nn.Conv2d(192, 192, kernel_size=1)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.dropout2 = nn.Dropout(p=0.5)
        self.conv7 = nn.Conv2d(192, 192, kernel_size=3, padding=1)
        self.conv8 = nn.Conv2d(192, 192, kernel_size=1)
        self.conv9 = nn.Conv2d(192, 10, kernel_size=1)
        self.pool3 = nn.AdaptiveAvgPool2d(output_size=1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.pool1(x)
        x = self.dropout1(x)
        x = F.relu(self.conv4(x))
        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = self.pool2(x)
        x = self.dropout2(x)
        x = F.relu(self.conv7(x))
        x = F.relu(self.conv8(x))
        x = self.conv9(x)
        x = self.pool3(x)
        x = x.view(x.size(0), -1)
        return x
#使用一个随机生成的输入，计算该网络的输出
net = NiN()
x = torch.randn(1, 3, 32, 32)
y = net(x)
print(y)
```

运行程序，输出如下：

```
[1,   100] loss: 2.304
[1,   200] loss: 2.305
[1,   300] loss: 2.305
...
[7,   300] loss: 2.305
[8,   100] loss: 2.305
[8,   200] loss: 2.304
tensor([[ -0.0495,  0.0198, -0.0152,  0.0597, -0.0159, -0.0469,  0.0025, -0.0185,
          -0.0051, -0.0109]], grad_fn=<ViewBackward>)
```

可以看到，该网络的输出是一个大小为 1×10 的张量，表示该输入图片在每个类别上的得分。

5.3.5　Google Inception Net 网络

Google Inception Net 采用了特殊的 Inception Module 构建网络，网络模型比 VGG 复杂，网络层数更深，但参数量比 VGG 少，性能也更好，在 ILSVRC 2014 的比赛中以较大优势获得了第一名，同年提出的 VGGNet 获得了第二名。从 2014 年该网络被第一次提出到 2016 年，Inception 共经历了 4 次改进和升级，并分别衍生了 Inception V1～V4 版本。本小节主要对 Inception V1 进行介绍。

Inception V1 降低参数量的目的有两点：第一，参数越多模型越庞大，需要供模型学习的数据量就越大，而目前高质量的数据非常昂贵；第二，参数越多，耗费的计算资源也会越大。

Inception V1 参数少但效果好的原因除了模型层数更深、表达能力更强外，还有两点：

(1) 去除了最后的全连接层，用全局平均池化层(即将图片尺寸变为 1×1)来取代它。全连接层几乎占据了 AlexNet 或 VGGNet 中 90%的参数量，而且会引起过拟合，去除全连接层后模型训练更快并且减轻了过拟合。

(2) Inception V1 中精心设计的 Inception Module 提高了参数的利用效率，其结构如图 5-22 所示。这一部分也借鉴了 NiN 的思想，形象的解释就是 Inception Module 本身如同大网络中的一个小网络，其结构可以反复堆叠在一起形成大网络。

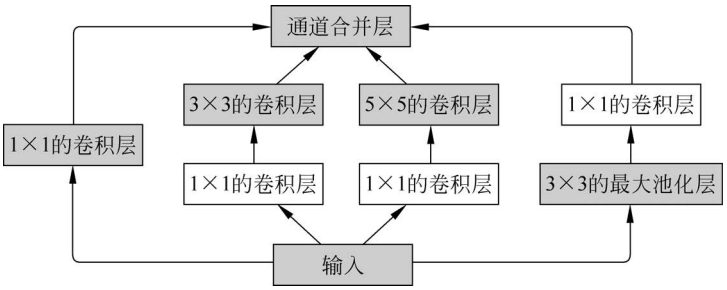


图 5-22　Inception V1 结构

Inception Module 的基本结构有 4 个分支。第一个分支对输入进行 1×1 的卷积，这

其实也是 NiN 中提出的一个重要结构。1×1 的卷积结构可以跨通道组织信息，提高网络的表达能力，同时可以对输出通道升维和降维。Inception Module 的 4 个分支都用到了 1×1 卷积进行低成本的跨通道的特征变换。第二个分支先使用 1×1 的卷积，然后连接 3×3 的卷积，相当于进行了两次特征变换。第三个分支与第二个分支类似，先是 1×1 的卷积，然后连接 5×5 的卷积。最后一个分支则是 3×3 的最大池化后直接使用 1×1 的卷积。Inception Module 的 4 个分支在最后通过一个聚合操作合并(在输出通道数这个维度上聚合)。

使用 PyTorch 来实现 Google Inception Net：

```
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms, datasets
import torch.optim as optim
from tqdm import tqdm

epochs = 5                          #迭代次数
lr = 0.1                            #学习率
batch_size = 32

data_transform = {
        "train": transforms.Compose([transforms.RandomResizedCrop(224),
                                transforms.RandomHorizontalFlip(),      #随机左右翻转
                                transforms.RandomVerticalFlip(),        #随机上下翻转
                                transforms.RandomRotation(degrees=5), #随机旋转
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.5, 0.5, 0.5), (0.5,
0.5, 0.5))]), "val": transforms.Compose([transforms.Resize((224, 224)),
             transforms.ToTensor(),transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))])}

train_dataset = datasets.CIFAR10('cifar', True,transform=data_transform["train"],
download=True)
validate_dataset = datasets.CIFAR10('cifar', True,transform=data_transform["val"],
download=False)

train_loader = torch.utils.data.DataLoader(train_dataset,
                           batch_size=batch_size, shuffle=True, num_workers=2)
validate_loader = torch.utils.data.DataLoader(validate_dataset,
                           batch_size=batch_size, shuffle=False,num_workers=2)
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
model = torchvision.models.resnet18()
model.fc.out_features = 10              #修改输出类别数
model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)

print('开始训练')
#训练模型
```

```
for epoch in range(epochs):
     model.train()                                          #训练模式
     epoch_loss = 0
     epoch_accuracy = 0
     for data, label in tqdm(train_loader, leave = False):
          data = data.to(device)
          label = label.to(device)

          output = model(data)
          loss = criterion(output, label)
          optimizer.zero_grad()          #清空以往梯度(因为每次循环都是一次完整的训练)
          loss.backward()                                   #反向传播
         optimizer.step()                                   #更新参数
         acc = (output.argmax(dim = 1) == label).float().mean()
         epoch_accuracy += acc / len(train_loader)      #当前训练平均准确率
         epoch_loss += loss / len(train_loader)         #累计 loss

    print(f'EPOCH: {epoch: 2}, train loss: {epoch_loss: .4f}, train acc: {epoch_accuracy:
.4f}')

model.eval()
acc = 0.0                                                   #计算准确数量/epoch
with torch.no_grad():
    for data,label in tqdm(validate_loader, leave = False):
         data = data.to(device)
         label = label.to(device)
         outputs = model(data)                              #评估模型只有最后一个输出层
         predict_y = torch.max(outputs, dim = 1)[1]
         acc += torch.eq(predict_y, label).sum().item()

val_accurate = acc / len(validate_dataset)
print(val_accurate)
```

运行程序，输出如下：

```
开始训练
EPOCH: 0, train loss: 4.5451, train acc: 0.1437
EPOCH: 1, train loss: 3.5573, train acc: 0.1574
EPOCH: 2, train loss: 3.5473, train acc: 0.1608
…
```

5.3.6　ResNet 网络

ResNet 在 2015 年被提出，在 ImageNet 比赛分类(classification)任务上获得第一名，因为它“简单与实用”并存，所以之后很多方法都是在 ResNet50 或者 ResNet101 的基础上完成的，检测、分割、识别等领域都纷纷使用 ResNet。

随着网络的加深，出现了训练集准确率下降的现象，可以确定这不是由于过拟合造成的(过拟合的情况训练集应该准确率很高)，所以针对这个问题提出了一种全新的网络，叫深度残差网络，它允许网络尽可能地加深，其中引入了全新的结构，如图 5-23 所示。

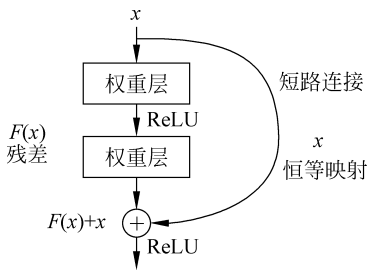


图 5-23　全新的结构

其中 ResNet 提出了两种映射(mapping)：一种是恒等映射(identity mapping)，指的就是图 5-23 中“弯弯的曲线”；另一种是残差映射(residual mapping)，指的就是除了“弯弯的曲线”的部分，所以最后的输出是 $y=F(x)+x$。

恒等映射顾名思义就是指本身，也就是公式中的 x，而残差映射指的是“差”，也就是 $y-x$，所以残差指的就是 $F(x)$部分。

使用 PyTorch 来实现 ResNet 网络：

```
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms, datasets
import torch.optim as optim
from tqdm import tqdm

epochs = 5                          #迭代次数
lr = 0.1                            #学习率
batch_size = 32

data_transform = {
        "train": transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.RandomHorizontalFlip(),     #随机左右翻转
                                     transforms.RandomVerticalFlip(),       #随机上下翻转
                                     transforms.RandomRotation(degrees=5), #随机旋转
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))]),"val": transforms.Compose([transforms.Resize((224, 224)),
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))])}
train_dataset = datasets.CIFAR10('cifar', True,transform=data_transform["train"],
download=True)
validate_dataset = datasets.CIFAR10('cifar', True,transform=data_transform["val"],
download=False)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, num_workers=2)
validate_loader = torch.utils.data.DataLoader(validate_dataset,
          batch_size=batch_size,shuffle=False, num_workers=2)

device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
model = torchvision.models.resnet18()
model.fc.out_features = 10                      #修改输出类别数
model.to(device)
```

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr = lr)
print('开始训练')
#训练模型
for epoch in range(epochs):
     model.train()                                    #训练模式
     epoch_loss = 0
     epoch_accuracy = 0
     for data, label in tqdm(train_loader, leave = False):
          data = data.to(device)
          label = label.to(device)
          output = model(data)
          loss = criterion(output, label)
          optimizer.zero_grad()          #清空以往梯度(因为每次循环都是一次完整的训练)
          loss.backward()                             #反向传播
          optimizer.step()                            #更新参数
         acc = (output.argmax(dim = 1) == label).float().mean()
         epoch_accuracy += acc / len(train_loader)    #当前训练平均准确率
         epoch_loss += loss / len(train_loader)       #累计 loss

    print(f'EPOCH: {epoch: 2}, train loss: {epoch_loss: .4f}, train acc: {epoch_accuracy:
.4f}')

model.eval()
acc = 0.0                                             #计算准确数量/epoch
with torch.no_grad():
    for data,label in tqdm(validate_loader, leave = False):
        data = data.to(device)
        label = label.to(device)
        outputs = model(data)                         #评估模型只有最后一个输出图层
        predict_y = torch.max(outputs, dim = 1)[1]
        acc += torch.eq(predict_y, label).sum().item()

val_accurate = acc / len(validate_dataset)
print(val_accurate)
```

运行程序,输出如下:

```
开始训练
EPOCH: 0, train loss: 2.0876, train acc: 0.2231
…
```

5.3.7　DenseNet 网络

DenseNet 网络的基本思路与 ResNet 一致,但它建立的是前面所有层与后面层的密集连接(即相加变连接),它的名称也由此而来。DenseNet 的另一大特色是通过特征在通道上的连接来实现特征重用。这些特点让 DenseNet 的参数量和计算成本都变得更少了(相对 ResNet),效果也更好了。ResNet 解决了深层网络梯度消失问题,它是从深度方向研究的。宽度方向是 GoogleNet 的 Inception。而 DenseNet 是从特征入手,通过对特

征的极致利用能达到更好的效果和减少参数。

1. DenseBlock

DenseBlock(密集块)包含很多层,每层的特征图大小相同(才可以在通道上进行连接),层与层之间采用密集连接方式,结构如图 5-24 所示。

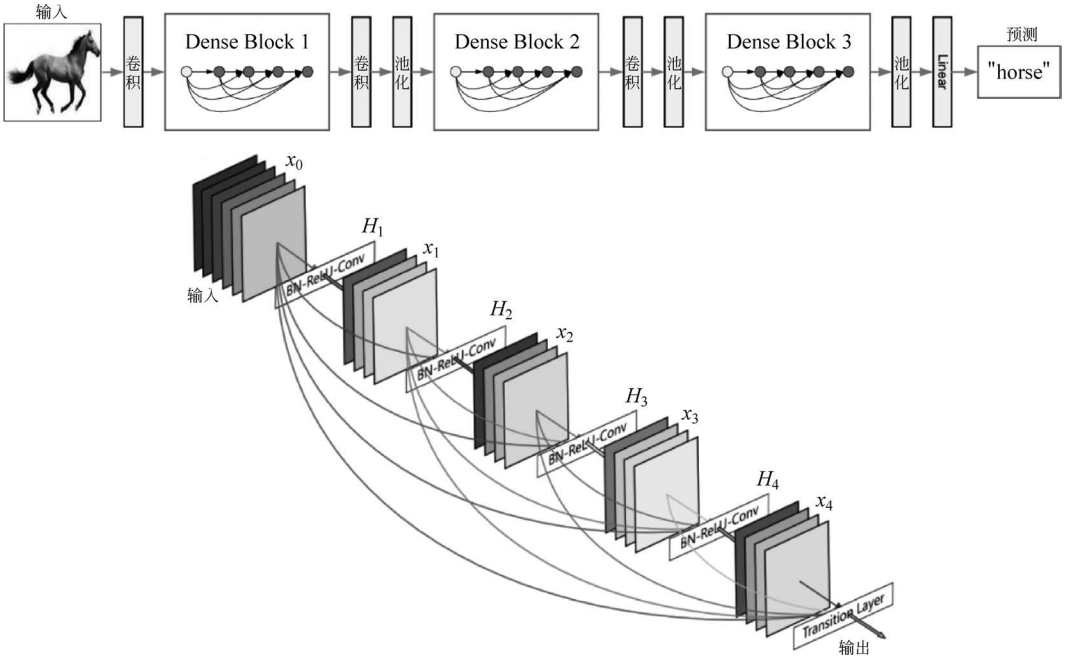


图 5-24 密集块结构

图 5-24 是一个包含 5 层的密集块。可以看出密集块互相连接所有的层,具体来说就是每层的输入都来自于它前面所有层的特征图,每层的输出均会直接连接到它后面所有层的输入。所以对于一个 L 层的密集块,共包含 $L(L+1)/2$ 个连接(等差数列求和公式),如果是 ResNet 则为 $2(L-1)+1$。从这里可以看出:相比 ResNet,密集块采用密集连接。而且密集块是直接连接来自不同层的特征图,这可以实现特征重用(即对不同"级别"的特征——不同表征进行总体性的再探索)以提升效率,这一特点是 DenseNet 与 ResNet 最主要的区别。

2. 转换层

转换层主要用于连接两个相邻的密集块,整合上一个密集块获得的特征,缩小上一个密集块的宽高,达到下采样效果,特征图的宽高减半。转换层包括一个 1×1 的卷积(用于调整通道数)和 2×2 的平均池化(用于降低特征图大小),结构为 BN+ReLU+1×1 的卷积+2×2 的平均池化。因此,转换层可以起到压缩模型的作用。

DenseNet 的网络结构主要由密集块和转换层组成,一个 DenseNet 中有 3 个或 4 个密集块。而一个密集块中也会有多个瓶颈层。最后的密集块之后是一个全局平均池化层,然后送入一个 softmax 分类器,得到每个类别的分数。

3. DenseNet-121 的 PyTorch 实现

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary
from torchstat import stat
from collections import OrderedDict

#构建密集块中的内部结构
#通过语法结构,把这个当成一个层即可
#bottleneck + DenseBlock == > DenseNet - B

class _DenseLayer(nn.Sequential):
    # num_input_features 作为输入特征层的通道数,growth_rate 为增长率,bn_size 输出的倍
#数一般都是 4,drop_rate 判断都是在 dropout 层进行处理
    def __init__(self, num_input_features, growth_rate, bn_size, drop_rate):
        super(_DenseLayer, self).__init__()
        self.add_module('norm1',nn.BatchNorm2d(num_input_features))
        self.add_module('ReLU1',nn.ReLU(inplace = True))

        self.add_module('conv1',nn.Conv2d(num_input_features,bn_size * growth_rate,
kernel_size = 1,stride = 1,bias = False))

        self.add_module('norm2',nn.BatchNorm2d(bn_size * growth_rate))
        self.add_module('ReLU2',nn.ReLU(inplace = True))
        self.add_module('conv2',nn.Conv2d(bn_size * growth_rate,growth_rate,kernel_size
 = 3,stride = 1, padding = 1,bias = False))
        self.drop_rate = drop_rate

    def forward(self, x):
        new_features = super(_DenseLayer, self).forward(x)
        if self.drop_rate > 0:
            new_features = F.dropout(new_features, p = self.drop_rate, training = self
.training)
        return torch.cat([x, new_features],1)

#定义密集块模块
class _DenseBlock(nn.Sequential):
    def __init__(self, num_layers, num_input_features, bn_size, growth_rate, drop_rate):
        super(_DenseBlock, self).__init__()
        for i in range(num_layers):
            layer = _DenseLayer(num_input_features + i * growth_rate, growth_rate, bn_
size, drop_rate)
            self.add_module("denselayer %d" %(i + 1), layer)

#定义转换层
#负责将密集块连接起来,一般都有 0.5 维道(通道数)的压缩
class _Transition(nn.Sequential):
    def __init__(self, num_input_features, num_output_features):
        super(_Transition, self).__init__()
        self.add_module('norm', nn.BatchNorm2d(num_input_features))
        self.add_module('ReLU',nn.ReLU(inplace = True))
```

```
        self.add_module('conv',nn.Conv2d(num_input_features,num_output_features,kernel_
size = 1,stride = 1,bias = False))
        self.add_module('pool', nn.AvgPool2d(2,stride = 2))

#实现 DenseNet 网络
class DenseNet(nn.Module):
    def __init__(self, growth_rate = 32,block_config = (6,12,24,26), num_init_features =
64,bn_size = 4, comparession_rate = 0.5, drop_rate = 0,num_classes = 1000):
        super(DenseNet,self).__init__()
        #前面:卷积层 + 最大池化
        self.features = nn.Sequential(OrderedDict([
              ('conv0',nn.Conv2d(3, num_init_features, kernel_size = 7, stride = 2,
padding = 3, bias = False)),
              ('norm0', nn.BatchNorm2d(num_init_features)),
              ('ReLU0', nn.ReLU(inplace = True)),
              ('pool0', nn.MaxPool2d(3,stride = 2,padding = 1))

        ]))
        #Denseblock
        num_features = num_init_features
        for i, num_layers in enumerate(block_config):
            block = _DenseBlock(num_layers, num_features, bn_size, growth_rate, drop_
rate)

            self.features.add_module("denseblock%d" %(i + 1), block)
            num_features += num_layers * growth_rate #确定一个密集块输出的通道数

            if i!= len(block_config) - 1: #判断是不是最后一个密集块
                transition = _Transition(num_features, int(num_features *
comparession_rate))
                self.features.add_module("transition%d" %(i + 1), transition)
                num_features = int(num_features * comparession_rate) #为下一个密
                                                                    #集块的输出做准备

        #最终:BN + ReLU
        self.features.add_module('norm5',nn.BatchNorm2d(num_features))
        self.features.add_module('ReLU5',nn.ReLU(inplace = True))
        #分类层
        self.classifier = nn.Linear(num_features,num_classes)
        #参数初始化
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.bias, 0)
                nn.init.constant_(m.weight, 1)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)
    def forward(self,x):
        features = self.features(x)
        out = F.avg_pool2d(features,7, stride = 1).view(features.size(0), - 1)
```

```
        out = self.classifier(out)
        return out

def densenet121(pretrained = False, **kwargs):
    """DenseNet121"""
    model = DenseNet(num_init_features = 64, growth_rate = 32, block_config = (6, 12, 24,
16), **kwargs)
    return model

if __name__ == '__main__':
    #输出模型的结构
    dense = densenet121()
    #模型每层的输出
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    #print(device)
    dense = dense.to(device)
    print('每层模型的输入输出: ',dense)
```

运行程序,输出如下:

```
每层模型的输入输出: DenseNet(
  (features): Sequential(
    (conv0): Conv2d(3, 64, kernel_size = (7, 7), stride = (2, 2), padding = (3, 3), bias =
False)
    (norm0): BatchNorm2d(64, eps = 1e - 05, momentum = 0.1, affine = True, track_running_
stats = True)
    (relu0): ReLU(inplace = True)
    (pool0): MaxPool2d(kernel_size = 3, stride = 2, padding = 1, dilation = 1, ceil_mode =
False)
    (denseblock1): _DenseBlock(
…
```

5.4　卷积神经网络 CIFAR10 数据集分类

CIFAR10 数据集中有 5 万张训练集和 1 万张测试集。针对该分类问题,其中包括的 10 个类别如图 5-25 所示。

['airplane','automobile','bird','cat','deer','dog','frog','horse','ship','truck']

图 5-25　CIFAR10 图片类别

CIFAR10 数据集中的图像都是 3 通道,尺寸为 32×32。

在 PyTorch 中,数据集是通过 Torchvision 下的 datasets 来加以实现的,实例代码如下:

1. CIFAR10 数据集

```
from torchvision import datasets,transforms
from torch.utils.data import DataLoader
```

```
#root: 数据集的根目录
#train: 是否为训练集
#transform: 可以将 PIL 和 NumPy 格式的数据从[0,255]范围转换到[0,1]. 原始数据的大小是
#(H × W × C),转换后大小会变为(C × H × W)
#download: 是否下载
train_ dataset  =  datasets. CIFAR10 ( root = " data/CIFAR10", train = True, transform =
transforms.ToTensor(),download = True)
test_ dataset  =  datasets. CIFAR10 ( root = " data/CIFAR10", train = False, transform =
transforms.ToTensor(),download = True)
#训练集长度
print(len(train_dataset))
#测试集长度
print(len(test_dataset))
#数据集类别
print(train_dataset.classes)
#训练集最后一张图片的类别
print(train_dataset.targets[49999])
#训练集最后一张图片的形状
print(train_dataset.data[49999].shape)
#训练集最后一张图片的数据
print(train_dataset.data[49999])
#测试集最后一张图片的类别
print(test_dataset.targets[9999])
#测试集最后一张图片的形状
print(test_dataset.data[9999].shape)
#测试集最后一张图片的数据
print(test_dataset.data[9999])

#上述的 ToTensor 在 dataloader 中调用
train_dataloader = DataLoader(dataset = train_dataset,batch_size = 5,shuffle = True)
for i,(img,tag) in enumerate(train_dataloader):
      print(tag)
      print(img.shape)
      print(img)
      Break
```

2. 神经网络设计

因为使用的数据集为 CIFAR10 数据集，最终做的是一个分类问题，所以在神经网络中包含了卷积神经网络和全连接神经网络。使用全连接神经网络对最终的分类概率进行求解。

(1) 残差模块。

在进行网络设计之前先设计出一个残差模块。注意，残差模块的输入通道和输出通道必须是相等的。

```
class Res_Net(nn.Module):
    def __init__(self,c_in,c_out,c):
        super(Res_Net, self).__init__()
        self.layer = nn.Sequential(
              #输入通道,输出通道,卷积核尺寸,步长,padding,参数 b
```

```
            nn.Conv2d(c_in,c,3,1,padding = 1,bias = True),
            nn.ReLU(),
            nn.Conv2d(c,c_out,3,1,padding = 1,bias = True),
            nn.ReLU()
        )

    def forward(self,x):
        return self.layer(x) + x
```

（2）网络设计。

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        #卷积
        self.conv_layer = nn.Sequential(
            #将 3 通道的图片转换为 64 通道
            nn.Conv2d(3,64,3,1,padding = 1),
            nn.ReLU(),
            #最大池化
            nn.MaxPool2d(2,2),
            nn.ReLU(),
            #残差模块
            Res_Net(64,64,64),

            nn.Conv2d(64,128,3,1),
            nn.ReLU(),
            Res_Net(128,128,128),
            Res_Net(128,128,128),

            nn.Conv2d(128,256,3,1),
            nn.ReLU(),
            Res_Net(256,256,256),
            Res_Net(256,256,256),
            Res_Net(256,256,256),

            nn.Conv2d(256,512,3,1),
            nn.ReLU(),
            Res_Net(512,512,512)
        )
        #全连接
        self.linear_layer = nn.Sequential(
            #输入为卷积的输出
            nn.Linear(512 * 10 * 10,1024),
            nn.ReLU(),
            #抑制全连接神经网络,减小运算量
            nn.Dropout(0.5),
            nn.Linear(1024,512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512,10),
```

```
            #softmax 激活函数,10 个类别的真实概率
            nn.Softmax(dim = 1)
        )
    def forward(self,x):
        conv_out = self.conv_layer(x)
        linear_in = conv_out.reshape( - 1,512 * 10 * 10)
        linear_out = self.linear_layer(linear_in)
        return linear_out
```

(3) 测试网络。

```
if __name__ == '__main__':
    net = Net()
    print(net)
    x = torch.randn(3,3,32,32)
    result = net.forward(x)
    print(result.shape)
    print(result)
```

3. 模型训练

```
import torch
from torchvision import datasets,transforms
from torch.utils.data import DataLoader
from torch import nn
from torch.utils.tensorboard import SummaryWriter
from Net import Net

#如果有 CUDA,则用 CUDA 训练,没有则使用 CPU 训练
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class Train():
    def __init__(self):
        #训练集
        self.train_dataset = datasets.CIFAR10(root = "data/CIFAR10", train = True,
transform = transforms.ToTensor(),download = True)
        #测试集
        self.test_dataset = datasets.CIFAR10(root = "data/CIFAR10", train = False,
transform = transforms.ToTensor(),download = True)
        #训练集加载器
        self.train_dataloader = DataLoader(dataset = self.train_dataset,batch_size =
500,shuffle = True)
        #测试集加载器
        self.test_dataloader = DataLoader(dataset = self.test_dataset,batch_size = 100,
shuffle = True)
        #创建网络
        self.net = Net()
        #网络位置
        self.net.to(DEVICE)
        #优化器
        self.opt = torch.optim.Adam(self.net.parameters())
```

```
        #损失函数：均方差
        self.loss_func = nn.MSELoss()
        #SummaryWriter类可以在指定文件夹生成一个事件文件,这个事件文件可以对
#TensorBoard解析
        self.summaryWriter = SummaryWriter("logs")

    def __call__(self, *args, **kwargs):
        #训练轮次
        for epoch in range(1000):
            #每一轮次训练总损失
            sum_loss = 0
            #加载数据
            for i,(img,tag) in enumerate(self.train_dataloader):
                self.net.train()                                    #训练模式
                img,tag = img.to(DEVICE),tag.to(DEVICE)             #将数据放在CUDA上
                out = self.net.forward(img)                         #计算结果
                one_hot_tag = nn.functional.one_hot(tag,10).float()#制作one-hot
                                                                    #标签
                loss = self.loss_func(out,one_hot_tag)              #计算损失
                sum_loss = sum_loss + loss
                self.opt.zero_grad()                                #清空梯度
                loss.backward()                                     #反向求导
                self.opt.step()                                     #更新参数
            avg_loss = sum_loss/len(self.train_dataloader)
            print("训练轮次：{}".format(epoch))
            print("训练损失：{}".format(avg_loss))
            #每一轮次测试总损失
            sum_test_loss = 0
            #每一轮次测试总分数
            sum_score = 0
            with torch.no_grad():                                   #不进行梯度下降操作,节约空间
                #加载数据
                for i,(img,tag) in enumerate(self.test_dataloader):
                    self.net.eval()                                 #测试模式
                    img,tag = img.to(DEVICE),tag.to(DEVICE)         #将数据放在CUDA上
                    test_out = self.net.forward(img)                #计算结果
                    one_hot_tag = nn.functional.one_hot(tag,10).float()#制作one-
                                                                    #hot标签
                    test_loss = self.loss_func(test_out,one_hot_tag)    #计算损失
                    sum_test_loss = sum_test_loss + test_loss
                    out_label = torch.argmax(test_out,dim=1)
                    tag_label = torch.argmax(one_hot_tag,dim=1)
                    score = torch.mean(torch.eq(out_label,tag_label).float())#计算
                                                                    #得分
                    sum_score = sum_score + score
                avg_test_loss = sum_test_loss/len(self.test_dataloader)
                avg_score = sum_score/len(self.test_dataloader)
                print(" ")
                print("测试损失：{}".format(avg_test_loss))
                print("测试得分：{}".format(avg_score))
                #保存训练参数
                torch.save(self.net.state_dict(),f"weights/{epoch}.pt")
```

```
                    #训练损失可视化：图片名 y 值和 x 值
                    self.summaryWriter.add_scalars("loss",{"train_loss": avg_loss,"test_
loss": avg_test_loss},epoch)

if __name__ == '__main__':
    train = Train()
    train()
```