第 5 章

卷积神经网络

卷积神经网络是一种前向神经网络，表示在网络中采用卷积的数学运算。卷积神经网络是一种特殊的神经网络，在至少一个层中使用卷积代替一般矩阵乘法运算。

5.1 卷积操作

卷积(Convolution)操作是卷积神经网络的基本操作，与多层感知机中点乘加和操作不同的是，卷积操作相当于一个滑动窗口，从左到右、从上到下地滑动(在此节仅讨论二维的卷积操作)，每滑动一下，就得出一个加权平均值，它更关注一小块或者局部的数据信息。卷积操作包括两个重要的组成成分：输入矩阵(Input)和卷积核(Kernel)，卷积核又称为滤波器(Filter)。它们分别对应感知机中的输入和权重。如图 5.1 所示，给定输入矩阵，通过核矩阵在输入矩阵上做滑动，可以得到所需的输出矩阵，又称为特征图。

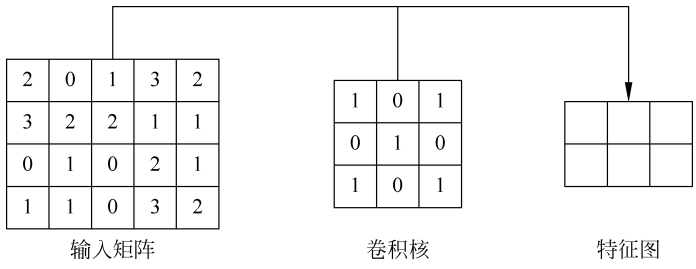


图 5.1 卷积操作的组成成分

那么卷积操作具体是怎么计算的呢？图 5.2(a)、(b)、(c)、(d)、(e)为 5 个不同的卷积步骤，图 5.2(f)为最终的输出结果。先将核矩阵作用到左上角的 3×3 方块上，计

算点乘得到第一个输出值 5，再依次向右移动两次，每次一个单元，得到第一行的输出 5、8、5。同样，计算第二行的卷积，得到如图 5.2(f)所示的最终输出结果。

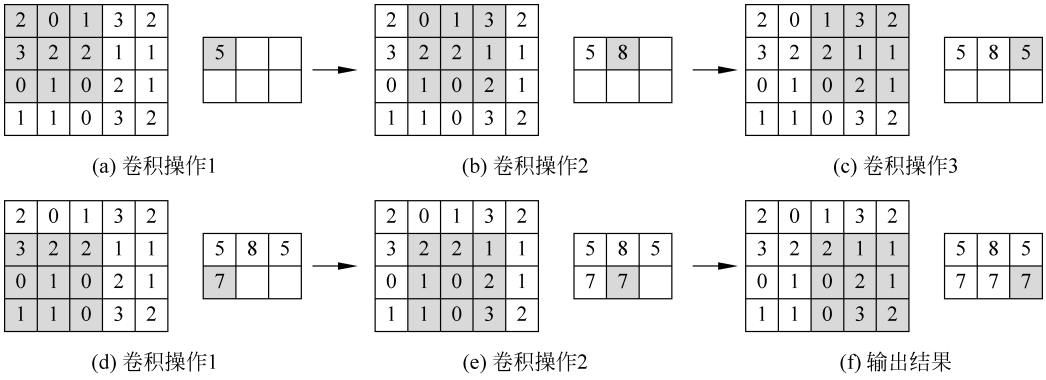


图 5.2　卷积操作的步骤

可以看到卷积核对输入矩阵重复计算卷积，遍历了整个矩阵，其每一个输出，都对应输入矩阵的一小块局部特征。卷积操作的另一个优点在于，输出的 2×3 的矩阵，共享同一个核矩阵，即参数共享(Parameter Sharing)，如果使用全连接操作，则需要一个 25×6≫3×3 的矩阵，图 5.2 中的每个卷积操作是独立的。也就是说，并不需要一定按照从左到右、从上到下的顺序来滑动计算卷积，也可以利用并行计算，同时计算所有方块的卷积值，达到高效运算的目的。

有时想要调整输出矩阵的大小，那么就要提到两个重要的参数，即步长(Stride)和填补(Padding)了。步长的影响如图 5.3 所示，横向移动不再是 1 步，而是设为 2 步，这样就跳过了中间的 3×3 方块，而纵向的步长仍为 1。通过设定大于 1 的步长，可以减小输出矩阵的大小。填补的操作如图 5.4 所示，可以让核矩阵的计算拓展到边缘之外

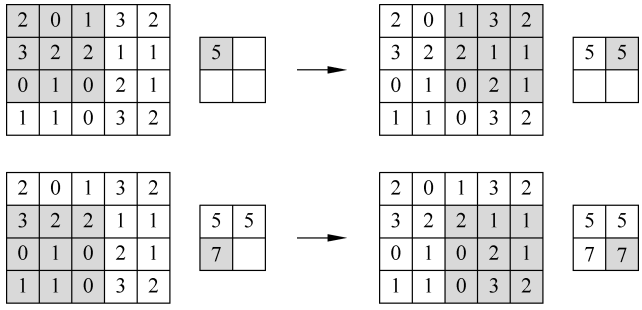


图 5.3　步长示意图

的区域。在原矩阵的上、下、左、右分别填补了 0、1、1、2 栏全为 0 的数值(假像素),则实现了这一目的。填补一方面增加了输出矩阵的大小,另一方面允许核函数以边缘像素为中心进行计算。在卷积计算中,可以通过步长和填补操作,来控制输出矩阵的大小,例如得到相等的或者长、宽各自减半的特征图。

0	2	0	1	3	2	0	0
0	3	2	2	1	1	0	0
0	0	1	0	2	1	0	0
0	1	1	0	3	2	0	0
0	0	0	0	0	0	0	0

图 5.4　填补示意图

在上述的例子中,只介绍了一个输入矩阵和一个核矩阵的卷积操作。事实上可以有多个相同的矩阵叠加在一起,例如图像通常有 3 个通道(Channel),分别代表红、黄、蓝三原色。图 5.5 给出了一个以图像为例的多通道的卷积操作,以图像为例,先把红、黄、蓝三个通道平铺开来,分别对三个通道用各自的核矩阵进行卷积操作,再把这三个输出矩阵相加,得到最终的特征图。注意到每个通道都有各自不同的核矩阵,如果输入通道数为 c_1,输出通道数为 c_2,则共需要 $c_1 \times c_2$ 个核矩阵。

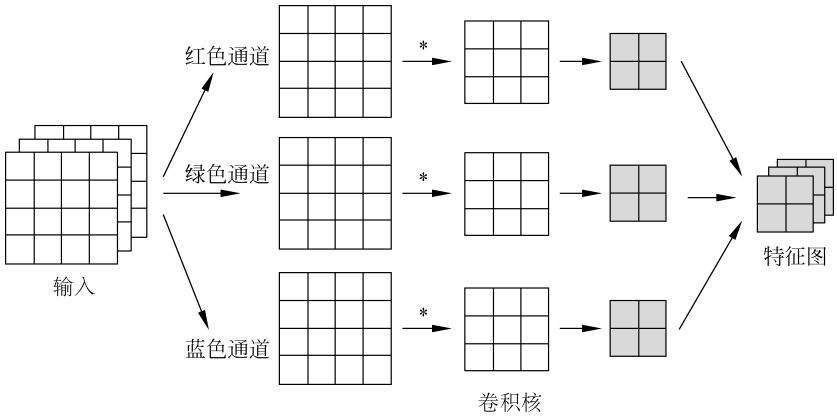


图 5.5　多通道卷积操作

5.2　池化

在 5.1 节,介绍了可以通过增加步长来减小输出矩阵大小等内容。池化(Pooling)则是另一种常见的降维操作。如图 5.6 所示,对 4×4 的特征图进行降维,对其中每个 2×2 的区域进行池化。常见的池化有两种:最大池化和平均池化。顾名思义,最大池

化是选择局部区域的最大值，而平均池化是计算局部区域的平均值。

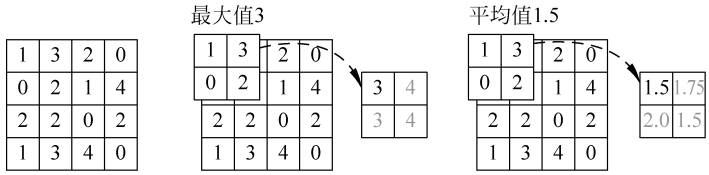


图 5.6　池化示意图

最大池化可以获取局部信息，更好地保留纹理上的特征。如果不用观察物体在图片中的具体位置，只关心其是否出现，则使用最大池化效果比较好。平均池化往往能保留整体数据的特征，更适用于突出背景信息时使用。通过池化，将一部分不重要的信息丢掉，保留更重要、更有利于特定任务的信息，从而达到降维、减少计算复杂度的目的。

同卷积操作类似，池化操作也可以通过重叠、定义步长等参数，来适应不同的应用场景。但与卷积操作不同的是，池化操作在单个矩阵上进行，卷积则是核矩阵在输入矩阵上的操作。可以把池化理解成一种特殊的核矩阵。

在了解卷积和池化的操作之后，就可以以图 5.7 中的 LeNet 为例，了解基本卷积神经网络的构成。给定一个 1×32×32(此处为灰度图，因而只有一个通道)的图片，先利用 6 个大小为 5×5 的卷积核，得到大小为 6×28×28 的特征图，这是网络的第一层。第二层为池化操作，对特征图进行降维，得到 6×14×14 的特征图。经过后面两层卷积核池化操作后，得到 16×5×5 特征图，经过最后一层卷积操作，每个输出为 1×1 的点，最终拼凑得到长度为 120 的特征向量，最后经过两层的全连接层，得到最终的输出向量，即类别的表达。这就是经典的 LeNet 模型，利用卷积神经网络提取特征图，最终利用全连接层将特征图转换为所需要的向量表达和输出形式。

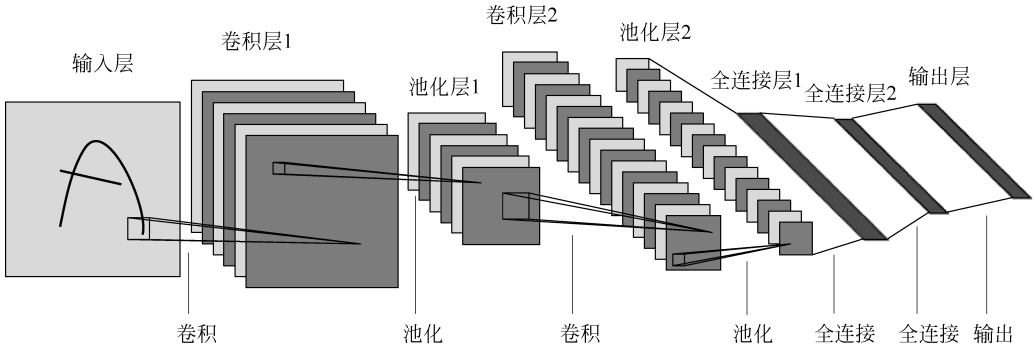


图 5.7　LeNet

5.3　残差网络

卷积神经网络通过层数的递增，逐渐抽取了更深层更普遍的特征信息，换句话说，特征的层次可以靠加深网络的层次来丰富。但实际应用中，当网络层数增加时，将会遇到梯度消失或爆炸的问题，导致网络很难训练。本节将介绍一种能有效解决神经网络深度增加的难题，即残差网络（Residual Network，ResNet）。

首先介绍残差网络的基本单元，残差块（Residual Block）。图 5.8 为一个基本的残差块。与普通连接网络不同的是，残差块中存在一条特殊的边，称为捷径（Shortcut），它使得上一层的输入 $\boldsymbol{x}_l$ 可以直接连接到输出 $\boldsymbol{x}_{l+1}$ 上，即 $\boldsymbol{x}_{l+1}=\boldsymbol{x}_l+\mathcal{F}(\boldsymbol{x}_l)$，其中 $\mathcal{F}(\boldsymbol{x}_l)=\boldsymbol{W}_2\mathrm{ReLU}(\boldsymbol{W}_1\boldsymbol{x}_l)$，为非线性变换，又称为残差。如果要学习一个映射函数 $\mathcal{H}(\boldsymbol{x})=\boldsymbol{x}$，那么学习 $\mathcal{F}(\boldsymbol{x})=0$ 要比 $\mathcal{F}(\boldsymbol{x})=\boldsymbol{x}$ 容易得多。也就是说，拟合残差会更加容易一些，这也是为什么这样的结构被称为残差块。

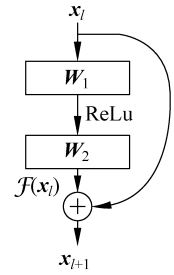


图 5.8　残差块示意图

之前提到残差网络可以解决梯度消失或爆炸的问题，可以简单推导残差网络上的反向传播来观察这一效果。假设网络一共有 L 层，则从任意第 l 层递归可以得到输出，公式如下：

$$\boldsymbol{x}_L=\boldsymbol{x}_l+\sum_{i=l}^{L-1}\mathcal{F}(\boldsymbol{x}_i) \tag{5.1}$$

假设损失函数为 E，通过链式法则可以得到对输入 $\boldsymbol{x}_l$ 的梯度为：

$$\frac{\partial E}{\partial \boldsymbol{x}_l}=\frac{\partial E}{\partial \boldsymbol{x}_L}\cdot\frac{\partial \boldsymbol{x}_L}{\partial \boldsymbol{x}_l}=\frac{\partial E}{\partial \boldsymbol{x}_L}\left(1+\frac{\partial}{\partial \boldsymbol{x}_l}\sum_{i=l}^{L-1}\mathcal{F}(\boldsymbol{x}_i)\right) \tag{5.2}$$

这个独立的“1”使得输出层梯度可以直接传回到 x_l 上，从而避免了梯度消失的问题。尽管梯度表达式没有显式地给出防止梯度爆炸的原因。但是在实际应用中，残差网络的使用确实有助于解决梯度爆炸问题，让我们在训练更深网络的同时，又能保证良好的性能。

由图 5.9[①] 中模型可以看出，残差网络就是由一层层残差块构成的，中间的每一个

① 图片来源：https://arxiv.org/pdf/1512.03385.pdf。

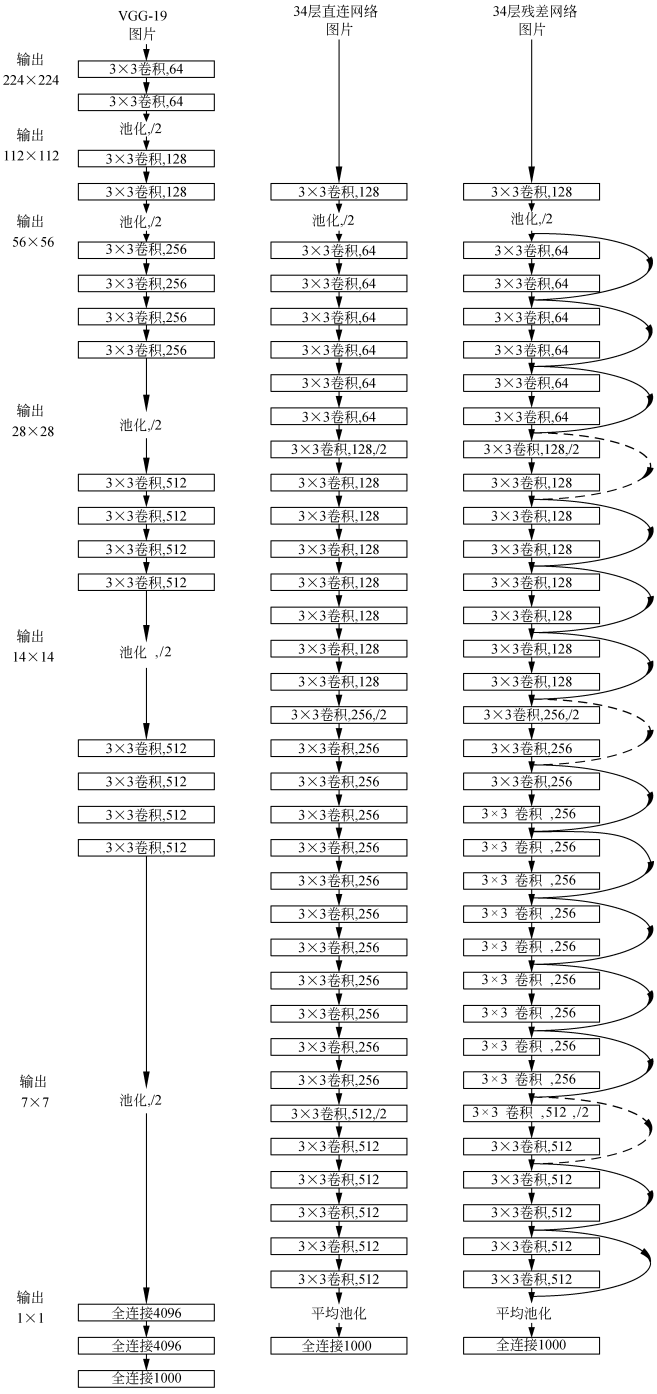

VGG-19
图片
34层直连网络
图片
34层残差网络
图片
输出
224×224
3×3卷积,64
3×3卷积,64
池化,/2
输出
112×112
3×3卷积,128
3×3卷积,128
池化,/2
输出
56×56
3×3卷积,256
3×3卷积,256
3×3卷积,256
3×3卷积,256
池化,/2
输出
28×28
3×3卷积,512
3×3卷积,512
3×3卷积,512
3×3卷积,512
池化 ,/2
输出
14×14
3×3卷积,512
3×3卷积,512
3×3卷积,512
3×3卷积,512
池化,/2
输出
7×7
全连接4096
全连接4096
全连接1000
输出
1×1
3×3卷积,128
池化,/2
3×3卷积,64
3×3卷积,128,/2
3×3卷积,256,/2
3×3卷积,512,/2
平均池化
全连接1000
3×3卷积,128
池化,/2
3×3卷积,64
3×3卷积,128,/2
3×3卷积,256,/2
3×3 卷积 ,512 ,/2
平均池化
全连接1000

图 5.9　残差网络

残差块，通过调整填补数使得输入和输出维度相等。捷径的存在，可以使我们根据需要添加或减少网络层数，保证模型训练的可行性。在卷积神经网络的发展中，残差网络有着相当大的影响力。

5.4　应用：图片分类

图片分类对于人来说是很简单的事情，但是对计算机来说，却是不容易的。在传统图像分类方法中，人们手工设计一些特征符，提取图像上一些局部的外表、形状、纹理等，再利用标准分类器，如支持向量机等，进行分类，其中还包含大量图片处理的方法技巧。卷积神经网络的诞生，大大推进了图片分类的发展，通过深层次的神经网络，可以直接从原始图像层面提取深层次的语义，让计算机有能力理解图片中的信息，从而将不同类别区分开来。以图 5.10 为例，不同卷积核可以对图像进行不同类型的操作，例如提取边缘轮廓、图像锐化等，与传统图像识别利用人工提取特征不同，卷积神

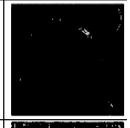
操作	卷积核	结果
自映射	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
边缘检测	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
锐化	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	

图 5.10　不同卷积核在图片上的作用

经网络可以根据具体任务需求，自发地学习特征提取的方式，不仅实现了更好的图片分类效果，而且适用于更多的任务数据场景。

图片分类最早最经典的应用，要数 MNIST 手写图像识别了。如图 5.11 所示，数据样本为 0～9 这 10 个手写数字，每个图像为 28×28 像素的灰度图。如果使用全连接网络进行分类，需要把每个图展开成长度为 784 的向量，这样一方面会丢失图片在空间上的信息，另一方面会造成训练参数过多，很容易过拟合。而卷积神经网络则很好地解决了这两个问题，首先卷积核的操作不会改变图像的空间像素分布，其次由于一个卷积核在一张图像上共享，可以更好地解决过拟合问题。

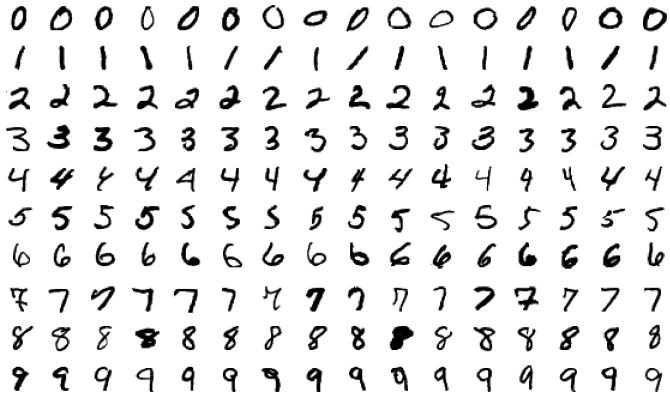


图 5.11　MNIST 手写图像识别

卷积神经网络先通过低层的卷积核，提取数字的轮廓信息，对图片本身进行降维，再逐步将这些信息抽象成计算机所能理解的特征，最终通过全连接层实现对数字的分类。如图 5.12 所示，如果将神经网络分类错误的图像筛选出来，将会发现其中有很多人类也容易混淆的数字，说明卷积神经网络确实学习到了图像中的数字语义信息。

下面再来看一组彩色图片分类的应用——CIFAR-10 数据的分类。这个数据集包含 6 万张 32×32 的彩色图像，代表飞机、汽车、鸟等 10 个类别的自然物体，图 5.13 展示了这 10 个类别和部分样例。CIFAI-10 中的语义信息很明显比数字中的更为复杂，同时输入的彩色数据具有 3 个通道而不是灰度图的单个通道。

图 5.14 展示了卷积神经网络中不同层中卷积核信息，从左到右依次由浅入深。可以观察到，浅层的卷积核用于学习边的特征，随着层次加深，逐渐学习到了局部轮廓，甚至整体语义的信息，而这些卷积核的初始状态，均为随机噪声。可以看到，卷积神经网络具有强大的图像特征学习能力，正是基于这种能力，计算机视觉在 2012 年得到了飞速发展。

图 5.12　MNIST 图像分类错误实例

图 5.13　CIFAR-10 数据集

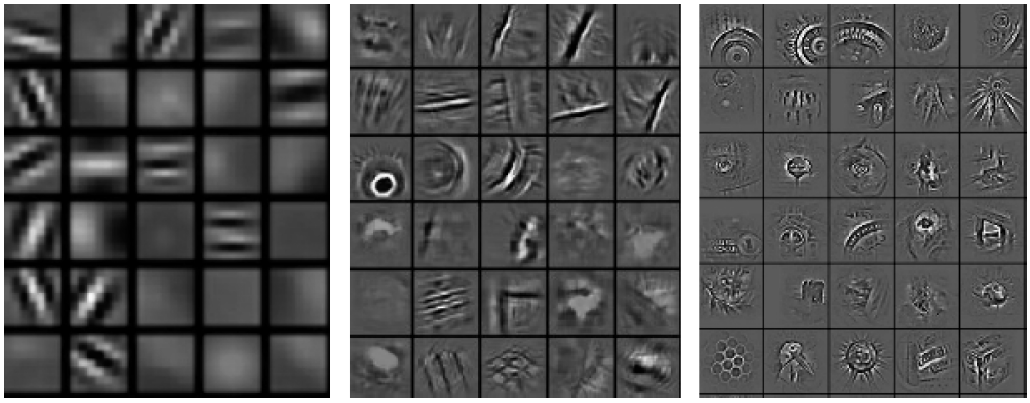


图 5.14　卷积神经网络不同层次的卷积核

随着卷积神经网络的发展，图片分类的应用也被拓宽到更广的领域，如图 5.15 所示照片中复杂物体的分类，图 5.16 的人脸识别以及植被鉴别等。总之，图片分类的应用，离不开卷积神经网络的贡献。

图 5.15　照片中复杂物体

图 5.16　图片分类的具体应用

5.5　用 MindSpore 实现基于卷积神经网络图片分类

说明：开发迭代 MindSpore 的接口及流程会不断演进，书中代码仅为示意代码，完整可运行代码请大家以线上代码仓中对应章节代码为准。

网址为：https://mindspore.cn/resource。读者可扫描右侧二维码获取相关资源。

5.4 节讲述了卷积神经网络在图片分类场景的作用，本节将使用 MindSpore 一步步来实现一个基于 ResNet50 网络的图片分类应用。

5.5.1　加载 MindSpore 模块

在进行网络训练之前需要导入 MindSpore 模块和辅助的第三方库，代码如下：

代码 5.1　导入 MindSpore 及第三方库

```
import numpy as np
from mindspore.nn import Conv2d, BatchNorm2d, ReLU, Dense, MaxPool2d, Cell, Flatten
from mindspore.ops.operations import TensorAdd, SimpleMean
from mindspore.common.tensor import Tensor
from mindspore.train.model import Model
from mindspore.nn import SoftmaxCrossEntropyWithLogits
from mindspore.nn import Momentum
from mindspore import context
```

5.5.2　定义 ResNet 网络结构

ResNet50 的连接结构主要有以下几个步骤。

(1) 底层输入连接层，包括 conv\batchnorm\relu\maxpool 操作。

(2) 连接 4 组残差模块，即下面的 4 个 MakeLayer，每个 MakeLayer 有不同的输入、输出通道和步长。

(3) 对网络进行最大池化和全连接层操作。

每个步骤的详细操作如下。

1. 定义基础操作

1) 定义变量初始化操作

由于构建网络的各个操作都需要初始化变量,因此需要定义变量初始化操作,此处利用 shape 构建初始化都为 0.01 的 Tensor,代码如下:

代码 5.2　定义变量初始化操作

```
def weight_variable(shape):
  ones = np.ones(shape).astype(np.float32)
  return Tensor(ones * 0.01)
```

2) 定义 conv 操作

构建网络之前需要定义一组卷积网络,即 conv。

定义 conv,卷积核大小分别为 1×1,3×3,7×7,步长为 1,代码如下。

代码 5.3　定义 conv

```
def conv1x1(in_channels, out_channels, stride = 1, padding = 0):
  """1x1 convolution"""
  weight_shape = (out_channels, in_channels, 1, 1)
  weight = weight_variable(weight_shape)
  return Conv2d(in_channels,
                out_channels,
                kernel_size = 1,
                stride = stride,
                padding = padding,
                weight_init = weight,
                has_bias = False,
                pad_mode = "same")
def conv3x3(in_channels, out_channels, stride = 1, padding = 1):
  """3x3 convolution """
  weight_shape = (out_channels, in_channels, 3, 3)
  weight = weight_variable(weight_shape)
  return Conv2d(in_channels,
                out_channels,
                kernel_size = 3,
                stride = stride,
```

```
                padding = padding,
                weight_init = weight,
                has_bias = False,
                pad_mode = "same")

def conv7x7(in_channels, out_channels, stride = 1, padding = 0):
  """1x1 convolution"""
  weight_shape = (out_channels, in_channels, 7, 7)
  weight = weight_variable(weight_shape)
  return Conv2d(in_channels, out_channels,
                kernel_size = 7,
                stride = stride,
                padding = padding,
                weight_init = weight,
                has_bias = False,
                pad_mode = "same")
```

3）定义 BatchNorm 操作

定义 BatchNorm 操作，进行归一化操作，核心代码如下。

代码 5.4　定义 BatchNorm 操作

```
def bn_with_initialize(out_channels):
  shape = (out_channels)
  mean = weight_variable(shape)
  var = weight_variable(shape)
  beta = weight_variable(shape)
  gamma = weight_variable(shape)
  bn = BatchNorm2d(out_channels,
                   momentum = 0.1,
                   eps = 1e - 5,
                   gamma_init = gamma,
                   beta_init = beta,
                   moving_mean_init = mean,
                   moving_var_init = var)
  return bn
```

4）定义 dense 操作

最后定义 dense 操作，将前面各层的特征整合到一起，核心代码如下。

代码 5.5　定义 dense 操作

```
def fc_with_initialize(input_channels, out_channels):
  weight_shape = (out_channels, input_channels)
```

```
    bias_shape = (out_channels)
    weight = weight_variable(weight_shape)
    bias = weight_variable(bias_shape)
    return Dense(input_channels, out_channels, weight, bias)
```

2. 定义 ResidualBlock 模块

每个 ResidualBlock 操作由 Conv > BatchNorm > ReLU 组成，用于传递给 MakeLayer 模块，核心代码如代码 5.6、代码 5.7 所示。

代码 5.6　定义 ResidualBlock 模块

```
class ResidualBlock(Cell):
  expansion = 4
  def __init__(self,
               in_channels,
               out_channels,
               stride=1,
               down_sample=False):
    super(ResidualBlock, self).__init__()

    out_chls = out_channels // self.expansion
    self.conv1 = conv1x1(in_channels, out_chls, stride=stride, padding=0)
    self.bn1 = bn_with_initialize(out_chls)

    self.conv2 = conv3x3(out_chls, out_chls, stride=1, padding=0)
    self.bn2 = bn_with_initialize(out_chls)

    self.conv3 = conv1x1(out_chls, out_channels, stride=1, padding=0)
    self.bn3 = bn_with_initialize(out_channels)

    self.relu = ReLU()
    self.add = TensorAdd()

  def construct(self, x):
    identity = x
    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)
    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)
```

```
    out = self.conv3(out)
    out = self.bn3(out)
    out = self.add(out, identity)
    out = self.relu(out)
    return out
```

代码 5.7　定义 ResidualBlock 模块

```
class ResidualBlockWithDown(Cell):
  expansion = 4
  def __init__(self,
               in_channels,
               out_channels,
               stride=1,
               down_sample=False):
    super(ResidualBlockWithDown, self).__init__()

    out_chls = out_channels // self.expansion
    self.conv1 = conv1x1(in_channels, out_chls, stride=stride, padding=0)
    self.bn1 = bn_with_initialize(out_chls)

    self.conv2 = conv3x3(out_chls, out_chls, stride=1, padding=0)
    self.bn2 = bn_with_initialize(out_chls)

    self.conv3 = conv1x1(out_chls, out_channels, stride=1, padding=0)
    self.bn3 = bn_with_initialize(out_channels)

    self.relu = ReLU()
    self.downSample = down_sample

     self.conv_down_sample = conv1x1(in_channels, out_channels, stride=stride,
padding=0)
    self.bn_down_sample = bn_with_initialize(out_channels)
    self.add = TensorAdd()

  def construct(self, x):
    identity = x
    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)
    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)
    out = self.conv3(out)
    out = self.bn3(out)
```

```
        identity = self.conv_down_sample(identity)
        identity = self.bn_down_sample(identity)
        out = self.add(out, identity)
        out = self.relu(out)
        return out
```

3. 定义 MakeLayer 模块

定义一组 MakeLayer 模块，每组模块的 block 不同，可以进行输入、输出通道和步长的设置，核心代码如下。

代码 5.8　定义 MakeLayer 模块

```
class MakeLayer0(Cell):
  def __init__(self, block, layer_num, in_channels, out_channels, stride):
    super(MakeLayer0, self).__init__()
    self.a = ResidualBlockWithDown(in_channels, out_channels, stride = stride, down_
sample = True)
    self.b = block(out_channels, out_channels, stride = 1)
    self.c = block(out_channels, out_channels, stride = 1)

  def construct(self, x):
    x = self.a(x)
    x = self.b(x)
    x = self.c(x)
    return x

class MakeLayer1(Cell):
  def __init__(self, block, layer_num, in_channels, out_channels, stride):
    super(MakeLayer1, self).__init__()
    self.a = ResidualBlockWithDown(in_channels, out_channels, stride = stride, down_
sample = True)
    self.b = block(out_channels, out_channels, stride = 1)
    self.c = block(out_channels, out_channels, stride = 1)
    self.d = block(out_channels, out_channels, stride = 1)

  def construct(self, x):
    x = self.a(x)
    x = self.b(x)
    x = self.c(x)
    x = self.d(x)
    return x

class MakeLayer2(Cell):
```

```
  def __init__(self, block, layer_num, in_channels, out_channels, stride):
    super(MakeLayer2, self).__init__()
    self.a = ResidualBlockWithDown(in_channels, out_channels, stride = stride, down_
sample = True)
    self.b = block(out_channels, out_channels, stride = 1)
    self.c = block(out_channels, out_channels, stride = 1)
    self.d = block(out_channels, out_channels, stride = 1)
    self.e = block(out_channels, out_channels, stride = 1)
    self.f = block(out_channels, out_channels, stride = 1)

  def construct(self, x):
    x = self.a(x)
    x = self.b(x)
    x = self.c(x)
    x = self.d(x)
    x = self.e(x)
    x = self.f(x)
    return x

class MakeLayer3(Cell):
  def __init__(self, block, layer_num, in_channels, out_channels, stride):
    super(MakeLayer3, self).__init__()
    self.a = ResidualBlockWithDown(in_channels, out_channels, stride = stride, down_
sample = True)
    self.b = block(out_channels, out_channels, stride = 1)
    self.c = block(out_channels, out_channels, stride = 1)

  def construct(self, x):
    x = self.a(x)
    x = self.b(x)
    x = self.c(x)
    return x
```

4. 定义整体网络

以上模块创建完之后，就可以定义整体 ResNet50 网络结构了，核心代码如下。

代码 5.9　定义整体 ResNet50 网络结构

```
class ResNet(Cell):
  def __init__(self, block, layer_num, num_classes = 10):
    super(ResNet, self).__init__()
```

```
    self.conv1 = conv7x7(3, 64, stride=2, padding=3)

        self.bn1 = bn_with_initialize(64)
        self.relu = ReLU()
        self.maxpool = MaxPool2d(kernel_size=3, stride=2, pad_mode="same")

        self.layer1 = MakeLayer0(
          block, layer_num[0], in_channels=64, out_channels=256, stride=1)
        self.layer2 = MakeLayer1(
          block, layer_num[1], in_channels=256, out_channels=512, stride=2)
        self.layer3 = MakeLayer2(
          block, layer_num[2], in_channels=512, out_channels=1024, stride=2)
        self.layer4 = MakeLayer3(
          block, layer_num[3], in_channels=1024, out_channels=2048, stride=2)

        self.pool = SimpleMean()
        self.fc = fc_with_initialize(512 * block.expansion, num_classes)
        self.flatten = Flatten()

      def construct(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.pool(x)
        x = self.flatten(x)
        x = self.fc(x)
        return x

    def resnet50(num_classes):
      return ResNet(ResidualBlock, resnet_shape, num_classes)
```

5.5.3 设置超参数

设置 batch，epoch，classes 等和损失函数及优化器相关超参数。损失函数定义的是 SoftmaxCrossEntropyWithLogits，采用 Softmax 进行交叉熵计算。选取 Momentum 优化器，学习率设置为 0.1，动量设置为 0.9，核心代码如代码 5.10 所示。

代码 5.10　超参数定义

```
context.switch_to_graph_mode()

epoch_size = 1
batch_size = 32
step_size = 1
num_classes = 10
lr = 0.1
momentum = 0.9
resnet_shape = [3, 4, 6, 3]
```

5.5.4　导入数据集

使用 MindSpore 数据格式 API 创建 ImageNet 数据集，其中下文调用的 train_dataset()函数具体实现和 MindSpore 数据格式 API 介绍详见第 14 章。

5.5.5　训练模型

1. 利用 train_dataset()读取数据

```
ds = train_dataset()
```

2. 利用 resnet()创建 ResNet50 网络结构

```
net = resnet50(num_classes)
net.set_train()
```

3. 设置损失函数和优化器

```
loss = SoftmaxCrossEntropyWithLogits(is_grad = False, sparse = True, sens = (1.0/batch_size))
opt = Momentum(lr, momentum, net.trainable_params())
```

4. 创建模型，调用 model.train()方法开始训练

```
model = Model(net, loss, opt)
model.train(epoch_size, ds)
```