

# Chapter 5

## Convolutional Neural Network



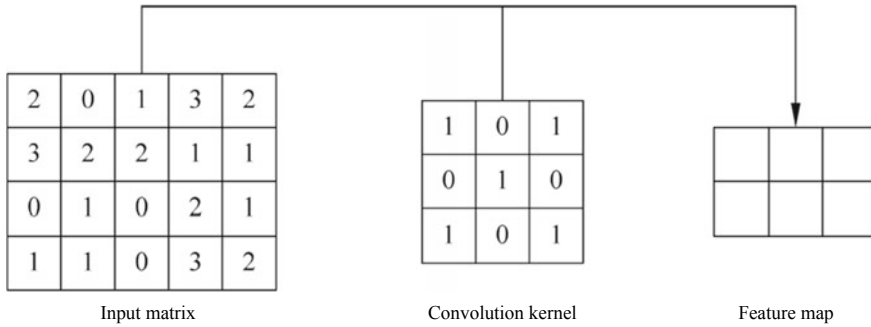
In this chapter, we describe the CNN. This network is a special neural network that uses convolution instead of general matrix multiplication at one or more layers. In essence, it is a feedforward neural network that uses convolutional mathematical operations.

### 5.1 Convolution

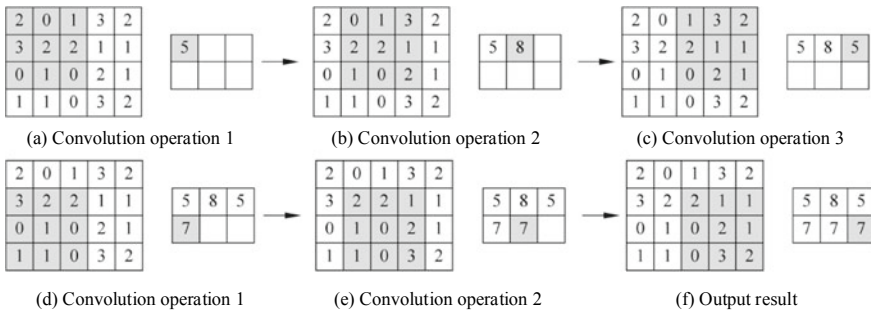
The convolution operation is fundamental in the CNN. Unlike the dot product accumulation operation in the MLP, the convolution operation is like a sliding window that slides from left to right and from top to bottom. (In this section, we focus exclusively on two-dimensional convolution operations.) Each time the window slides from one point to another, a weighted mean value focused on a small piece of data or local data is obtained. The convolution operation consists of two important components: input matrix and convolution kernel (also known as the filter), which correspond to the input and the weight in the perceptron, respectively. As shown in Fig. 5.1, we can obtain the desired output matrix (also known as a feature map) given the input matrix by sliding the kernel matrix on the input matrix.

The calculation is performed as follows during the convolution operation: First, the kernel matrix is applied to the  $3 \times 3$  blocks in the upper left corner, as shown in Fig. 5.2a. The first output value 5 is obtained from the dot product. Then the kernel matrix is moved to the right twice. For each move, the output in one block is obtained, as shown in Fig. 5.2b, c. The output values 5, 8, and 5 in the first row are obtained. Similarly, the convolution of the second row is calculated to obtain the final output result, as shown in Fig. 5.2f.

We can see that the convolution kernel repeatedly calculates convolution for the input matrix and traverses the entire matrix. Furthermore, each output corresponds to a local feature of a small part of the input matrix. One advantage of the convolution operation is that the output  $2 \times 3$  matrix shares the same kernel matrix; that is, it



**Fig. 5.1** Components of the convolution operation



**Fig. 5.2** Steps of the convolution operation

shares the same parameter setting. If the full-connection operation is used, a  $25 \times 6 \gg 3 \times 3$  matrix is required, and each convolution operation in Fig. 5.2 is independent. This means that we do not need to slide the window from one point to another in order to perform convolution calculation. Instead, the convolutional values of all the blocks can be calculated concurrently for efficient operation.

Sometimes the output matrix needs to be resized, and this can be accomplished by using two important parameters: stride and padding. As shown in Fig. 5.3, the stride for lateral movement is 2 instead of 1, (this means that the  $3 \times 3$  blocks in the middle are skipped), whereas the stride for longitudinal movement is 1. By setting a stride greater than 1, we can reduce the size of the output matrix. The other important parameter is padding. As shown in Fig. 5.4, padding allows the calculation of the kernel matrix to be extended beyond the confines of the matrix. This is achieved by padding one row of 0 s (false pixels), one column of 0 s, and two columns of 0 s on the lower, left, and right sides of the original matrix. The padding increases the size of the output matrix and allows the kernel function to be calculated around the edge pixels. In convolution calculation, the size of the output matrix can be controlled based on the stride and padding parameters. This can be useful if we want to obtain

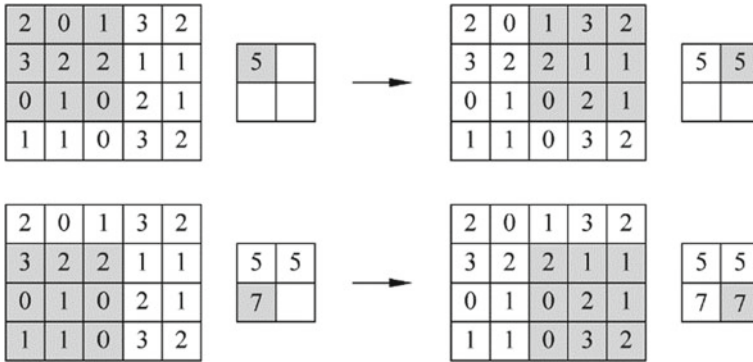


Fig. 5.3 Stride

Fig. 5.4 Padding

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

a feature map with the same length and width, or half-length and half-width, for example.

The convolution operation above involves only one input matrix and one kernel matrix. However, we can superimpose multiple identical matrices together. As an example, take an image, which typically includes three channels that represent the three primary colors: red, green, and blue. In a multi-channel convolution operation of an image (as shown in Fig. 5.5), the red, green, and blue channels are tiled first. These channels are convoluted by using their respective kernel matrices, and then three output matrices are added to obtain a final feature map. Note that each channel has its own kernel matrix. If the number of input channels is  $c_1$  and the number of output channels is  $c_2$ , a total of  $c_1 \times c_2$  kernel matrices are needed.

## 5.2 Pooling

As described in Sect. 5.1, we can reduce the size of the output matrix by increasing the stride parameter. Another common method for such reduction is pooling. For example, a  $4 \times 4$  feature map can be reduced to  $2 \times 2$  regions, which are then pooled as a  $2 \times 2$  feature map, as shown in Fig. 5.6. There are two common types of

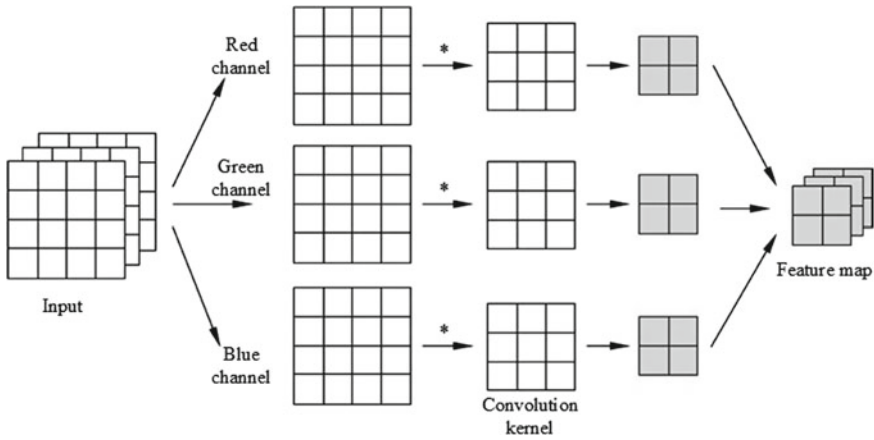


Fig. 5.5 Multi-channel convolution operation

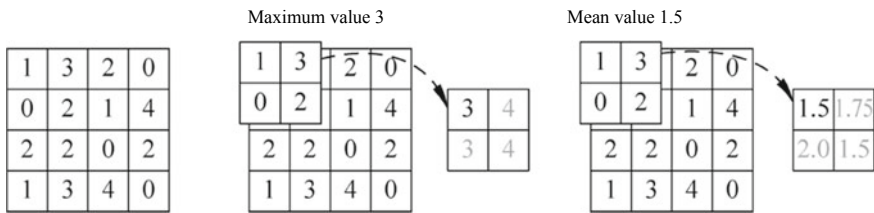


Fig. 5.6 Pooling

pooling: max-pooling and mean-pooling. As the names imply, max-pooling selects the maximum value of a local region, whereas mean-pooling calculates the mean value of a local region.

Max-pooling can obtain local information and preserve texture features more accurately. It is ideal if we want only to determine whether an object appears in an image, not for observing the specific location of the object in the image. Mean-pooling, on the other hand, can usually preserve the features of the overall data and is more suitable for highlighting background information. Through pooling, some unimportant information is discarded, while information that is more important and more favorable to a particular task is reserved, to reduce dimensionality and computational complexity.

Similar to the convolution operation, the pooling operation can be adapted to different application scenarios by overlapping and defining parameters such as stride. Unlike the convolution operation, however, the pooling operation is performed on a single matrix, and convolution is a kernel matrix operation on an input matrix. We can understand pooling as a special kernel matrix.

With a basic understanding of the convolution and pooling operations, we can take LeNet in Fig. 5.7 as an example to examine what comprises a CNN. Given a  $1 \times 32$

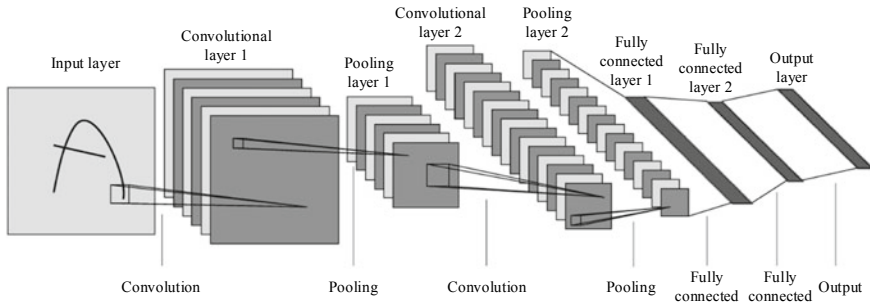


Fig. 5.7 LeNet

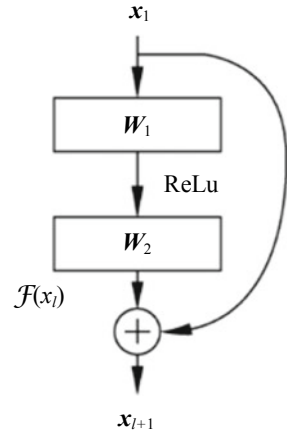
$\times 32$  single-channel grayscale image, we can first use six  $5 \times 5$  convolution kernels to obtain a feature map of  $6 \times 28 \times 28$ , which is the first layer of the network. The second layer is the pooling operation, which reduces the dimension of the feature map to obtain one that is  $6 \times 14 \times 14$ . At the last two layers, the convolution kernel performs further pooling operations, giving us a  $16 \times 5 \times 5$  feature map. After the convolution operation is performed at the last layer, each output is a  $1 \times 1$  point. In addition to this, we obtain an eigenvector with a length of 120. Finally, we obtain an output vector, that is, a category expression, through two fully connected layers. This is the classical LeNet model, in which the CNN is used to extract the feature map, and the fully connected layer is used to convert the feature map into the vector expression and output form.

### 5.3 Residual Network

By increasing the number of layers in the CNN, we are able to extract deeper and more general features. In other words, we can deepen the network level in order to enrich the feature level. However, when the number of network layers increases, gradient disappearance or explosion may occur, making it difficult to train the network. This section introduces residual network (ResNet), a solution that effectively solves the problem caused by increasing the depth of the neural network.

The basic element of the residual network is called a residual block and is shown in Fig. 5.8. Unlike the common connection network, the residual block includes a special edge, which is called a shortcut. The shortcut enables the input  $x_l$  of the upper layer to be directly connected to the output  $x_{l+1}$ , that is,  $x_{l+1} = x_l + \mathcal{F}(x_l)$ , where  $\mathcal{F}(x_l) = \mathbf{W}_2 \text{ReLU}(\mathbf{W}_1 x_l)$  indicates a nonlinear transformation and is also called residual. Let us assume that we want to learn a mapping function  $\mathcal{H}(x) = x$ . In this case, learning  $\mathcal{F}(x) = 0$  is much easier than learning  $\mathcal{F}(x) = x$ , because it is easier to fit the residuals. This is why such a structure is called a residual block.

As mentioned earlier, the residual network can solve the problem of gradient disappearance or explosion. We are able to observe this by deducing backpropagation

**Fig. 5.8** Residual block

in the residual network. If we assume that the network includes  $L$  layers, we can obtain the output from any layer  $l$  through recursion:

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

Assuming that the loss function is  $E$ , then we can obtain the gradient of the input  $\mathbf{x}_l$  according to the chain rule:

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

The independent “1” allows the gradient of the output layer to be propagated directly back to  $\mathbf{x}_l$ , thereby avoiding gradient disappearance. Although the gradient expression does not explicitly give the reason for preventing the gradient explosion problem, the use of the residual network helps solve this problem in practical applications.

In Fig. 5.9,<sup>1</sup> we can see that the residual network includes layers of residual blocks. Each intermediate residual block adjusts the padded value to ensure that the number of input dimensions is equal to the number of output dimensions, and the shortcut allows us to add or reduce the number of network layers in order to ensure the feasibility of model training. The residual network, therefore, has significant influence in the development of the CNN.

<sup>1</sup> Source: <https://arxiv.org/pdf/1512.03385.pdf>.

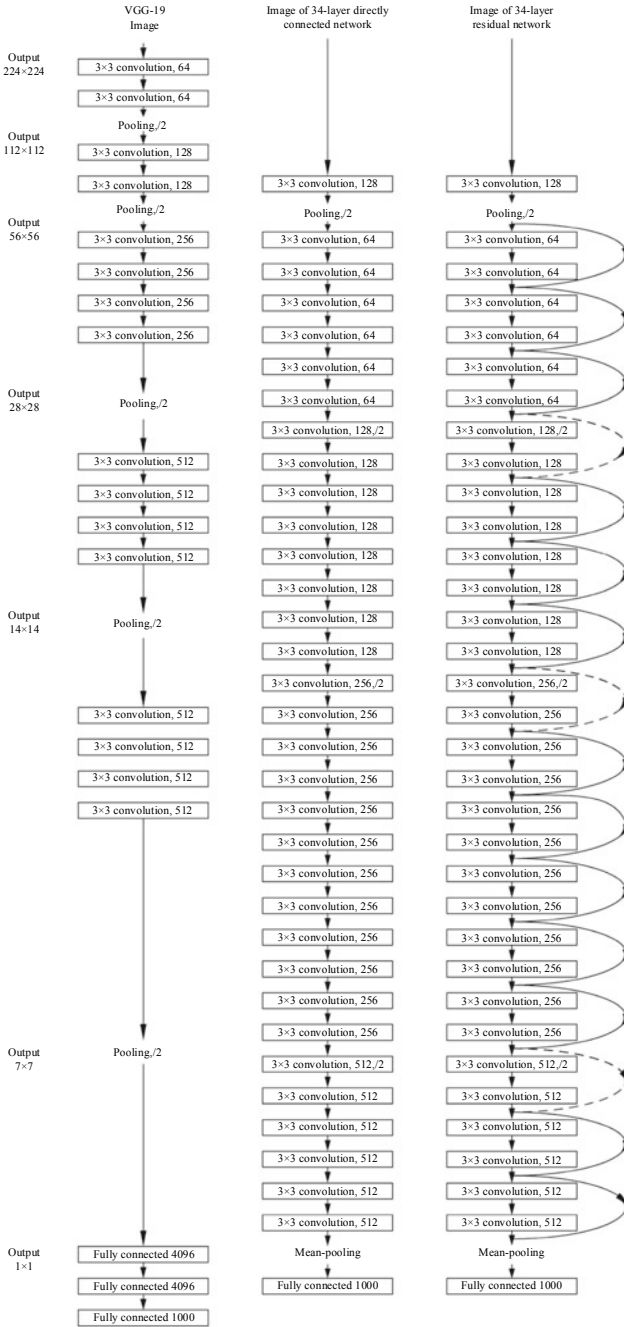


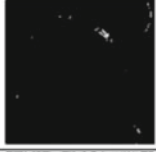
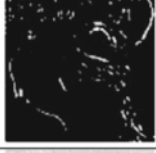



Fig. 5.9 Residual network

## 5.4 Application: Image Classification

Image classification is a simple task for humans, but is a difficult one for computers. The traditional method used for image classification is heavily dependent on humans having strong image processing skills. In this method, humans manually design features, extract local appearances, shapes, and textures on the image and then use a standard classifier, such as an SVM, to classify the image. However, the emergence of the CNN has had a significant impact on image classification, promoting its ongoing development. The DNN can directly extract deep semantics from the original image level, enabling the computer to understand the information in the image and distinguish between different categories. Taking Fig. 5.10 as an example, different convolution kernels can perform different operations on images, such as edge contour extraction and image sharpening. Unlike the traditional method mentioned earlier, where features are manually extracted, the CNN can automatically learn feature extraction according to specific task requirements. This means that the CNNs are

Operation	Convolution kernel	Result
Self-mapping	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpening	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	

**Fig. 5.10** Functions of different convolution kernels on images





Fig. 5.11 MNIST handwritten image recognition

able to deliver better image classification effects as well as being suitable for more task data scenarios.

The earliest application of image classification is MNIST handwritten image recognition—one that has subsequently become a classic. As shown in Fig. 5.11, data samples are 10 handwritten numbers ranging from 0 to 9, and each image is a grayscale image of  $28 \times 28$  pixels. If a fully connected network is used for classification, each image needs to be expanded into a vector with a length of 784. This approach will result in the loss of the image’s spatial information, and requires too many training parameters, potentially leading to overfitting. We can solve these two problems by using CNNs. First, the operations of the convolution kernels will not change the spatial pixel distribution of the images, meaning that no spatial information is lost. Second, because the convolution kernels are shared on images, the overfitting problem can be solved more effectively.

The CNN first extracts the contour information of the numeral image by using the lower convolution kernel. It then reduces the dimension of the image, abstracts the information into features that the computer can understand, and finally classifies the number through the fully connected layer. As shown in Fig. 5.12, many of the images that are misclassified by the neural network are also difficult for humans to identify. However, this indicates that the CNNs have actually learned the semantics of the numbers in the images.

Now we will look at the application of color image classification—Canadian Institute for Advanced Research-10 (CIFAR-10) data classification. The dataset includes 60,000  $32 \times 32$  color images that represent 10 categories of natural objects, such as aircrafts, automobiles, and birds. Figure 5.13 shows the 10 categories and some examples of each category. The semantic information in CIFAR-10 is more complex



Fig. 5.12 Misclassified MNIST images

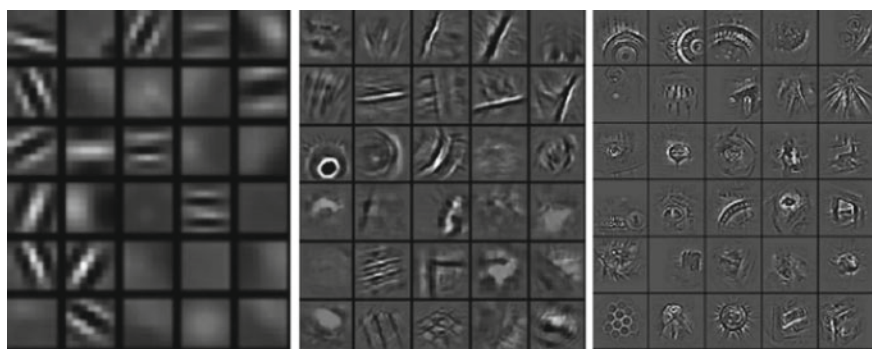
than that in numbers, and the input color data includes three channels rather than only one in the grayscale image.

Figure 5.14 shows convolution kernels at different layers of CNNs. The layers progressively get deeper from left to right. We can see that the convolution kernels at shallow layers are used to learn the features of edges. With the deepening of the layer, the local contour and even the overall semantics are gradually learned, and the initial states of these convolution kernels are all random noises. We can also see that the CNNs have a strong ability for learning image features, resulting in the rapid development of computer vision in 2012.

As the CNN has developed, the application of image classification has grown to cover the classification of complex objects in photographs (as shown in Fig. 5.15), facial recognition (as shown in Fig. 5.16), and other fields such as plant identification. We can therefore conclude that the application of image classification is inseparable from the CNN.



**Fig. 5.13** CIFAR-10 dataset



**Fig. 5.14** Convolution kernels at different layers of the CNN



Fig. 5.15 Complex objects in photographs



Fig. 5.16 Application of image classification

## 5.5 Implementing Image Classification Based on the DNN Using MindSpore



The interfaces and processes of MindSpore may constantly change due to iterative development. For all runnable code, see the code in corresponding chapters at <https://mindspore.cn/resource>. You can scan the QR code on the right to access relevant resources.

In Sect. 5.4, we described the functions of the CNN in image classification. Building on that, we use MindSpore in this section to systematically implement an image classification application based on the ResNet50 network.

### 5.5.1 Loading the MindSpore Module

Before network training, it is necessary to import the MindSpore module and third-party auxiliary library. The core code is as follows:

#### Code 5.1 Importing the MindSpore Module and Third-party Library

```
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 Defining the ResNet Network Structure

The steps for defining ResNet50 are as follows:

- (1) Perform operations such as conv, batchnorm, relu, and maxpool for the bottom input connection layer.
- (2) Connect four sets of residual modules, each with a different input, output channel, and stride.
- (3) Perform max-pooling and fully connected layer operations on the network.

The details of each step are as follows.

#### 1. Define basic operations

- (1) Define a variable initialization operation.

Because each operation for constructing the network requires initialization of variables, a variable initialization operation needs to be defined. Here, we use **shape** to construct a tensor that is initialized to 0.01. The core code is as follows:

#### Code 5.2 Defining the Variable Initialization Operation.

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

(2) Define a conv operation.

Before constructing a network, define a set of convolutional networks, that is, conv.

Set the convolution kernel sizes to  $1 \times 1$ ,  $3 \times 3$ , and  $7 \times 7$ , and set the stride to 1. The core code is as follows:

### Code 5.3 Defining 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) Define a BatchNorm operation.**

Define the BatchNorm operation to perform the normalization operation. The core code is as follows:

**Code 5.4 Defining the BatchNorm Operation**

```
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) Define a dense operation.**

Define the dense operation to integrate the features of the previous layers. The core code is as follows:

**Code 5.5 Defining the Dense Operation**

```
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. Define the ResidualBlock module**

Each ResidualBlock operation includes Conv > BatchNorm > ReLU, which are delivered to the MakeLayer module. The core code is as follows:

**Code 5.6 Defining the ResidualBlock Module**

```
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
```



**Code 5.7 Defining the ResidualBlock Module**

```
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. Define the MakeLayer module

Define a set of MakeLayer modules with different blocks. Set the input, output channel, and stride. The core code is as follows:

#### Code 5.8 Defining the MakeLayer Module

```

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. Define the overall network

Once the MakeLayer modules have been created, define the overall ResNet50 network structure. The core code is as follows:

#### Code 5.9 Defining the Overall ResNet50 Network Structure

```
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 *Setting Hyperparameters*

Set hyperparameters related to the loss function and optimizer, such as batches, epochs, and classes. Define the loss function as `SoftmaxCrossEntropyWithLogits`, using `Softmax` to calculate the cross-entropy. Select the momentum optimizer, and set its learning rate to 0.1 and momentum to 0.9. The core code is as follows:

#### **Code 5.10 Defining Hyperparameters**

```
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 *Importing a Dataset*

Create an ImageNet dataset using the MindSpore data format APIs. For details about these APIs and how to implement the `train_dataset()` function, see Chap. 14.

### 5.5.5 *Training a Model*

#### **1. Use `train_dataset()` to read data**

```
ds = train_dataset()
```

#### **2. Use `resnet()` to create the ResNet50 network structure**

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

### 3. Set the loss function and optimizer

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

### 4. Create a model and call the `model.train()` method to start training

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