第 9 章 卷积神经网络 - TensorFlow

在实践中，卷积神经网络可能包括多个卷积层/池化层或者全连接层，使得卷积神经网络包含几十甚至几百个隐藏层。如果我们使用 Numpy 等实现卷积神经网络，需要把很多精力放在正向传播算法、反向传播算法和梯度下降法等技术细节中。而使用深度学习框架可以让我们集中精力进行数据预处理、神经网络的结构设计以及超参数的选择等方面，从而提高建模效率和算法的运行效率。本章主要学习 3 方面内容。

- 学习 TensorFlow 实现卷积层和池化层的函数。
- 学习使用 TensorFlow 分析 mnist 数据（黑白图片）和 CIFAR-10 数据（彩色图片）。
    - 在计算机视觉中，黑白图片通常使用灰度值表示成二维数组的形式。图片的维度有宽度（W，Width）和高度（H，Height），如图 9-1 所示。

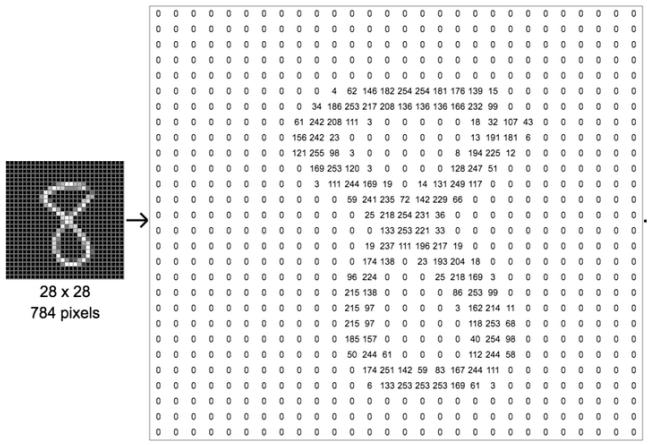

图 9-1 黑白图片，用二维灰度值表示

    - 彩色图片通常用 RGB 表示。彩色图片的每一个像素点都由 R（红色）、G（绿色）、B（蓝色）3 个数值表示（如图 9-2 所示）。R、G、B 的数值范围都是 0～255。彩色图片除了图片宽度（W，Width）和高度（H，Height）之外，还有深度（也称为颜色通道，Channel）。因此，彩色图片用 3 维数组表示。

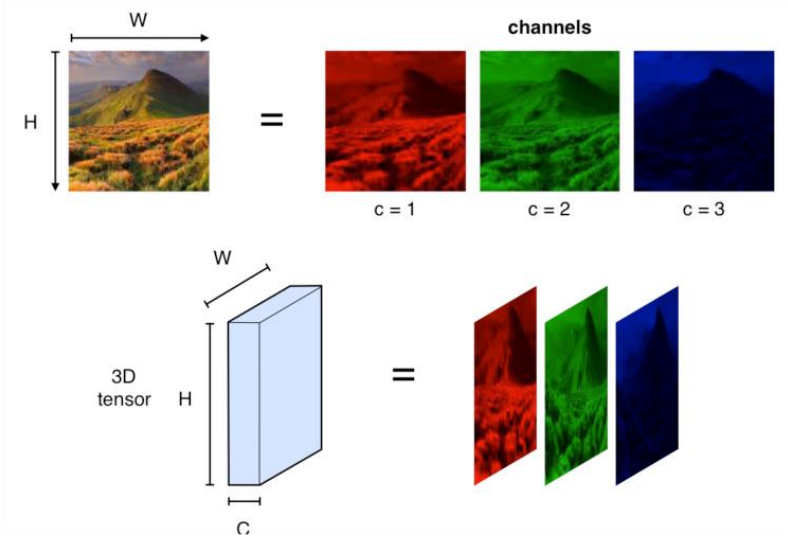

图 9-2 彩色图片，用 3 维数组表示

- 学习卷积神经网络的建模技巧。

9.1 卷积层和池化层-TensorFlow

首先学习两个新的 TensorFlow 函数：`tf.keras.layers.Conv2D()` 和 `tf.keras.layers.MaxPooling2D()`，这两个函数分别实现卷积层和池化层。下面代码载入所需要的包，包含 Tensorflow、Numpy、idx2numpy 和 matplotlib。

```
%config InlineBackend.figure_format = 'retina'
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import idx2numpy
```

为了更清楚地看到两个函数的运行结果，先考虑一个简单例子。假设输入数据维度为4×4。首先把这个4×4的数据转化为维度为$1 \times 4 \times 4 \times 1$的数据。其中，第一个维度1表示观测点的个数，这里只有一个观测点；第二和第三个维度(4,4)表示数据的宽度和高度；第四个维度1表示数据的深度。TensorFlow 函数 `tf.keras.layers.Conv2D()` 要求输入数据为 4 维形式。

```
tf.random.set_seed(1)
images = np.array([[0, 1, -1, 2],
                   [-1, 2, 1, 0.1],
                   [0, 0.1, 1, -1],
                   [3, 1, 1, 1]])
images = images.reshape(1,4,4,1)
```

下面的代码调用函数 `tf.keras.layers.Conv2D()` 计算卷积层：

- 第一个参数 `1`，表示该函数输出结果的深度为 1，即只使用一个卷积核 。

- kernel_size=(3, 3)表示核的宽度和高度都为 3
- strides=(1, 1)表示步幅，核无论是横向还是纵向，每次都是移动一格
- input_shape=(4,4,1)表示输入数据的宽度、高度和深度分别为(4,4,1)
- padding='same'表示该函数输出结果的宽度和高度保持与输入数据一致

```
conv_layer = tf.keras.layers.Conv2D(1, kernel_size=(3, 3), \
             strides=(1, 1), input_shape=(4,4,1), padding='same')
```

现在，conv_layer 只是定义了卷积运算，并设置了相关参数，还没有产生核的初始值，更没有实现卷积运算。conv_layer(images)输入简单数据 images，并实现卷积运算。conv_layer(images)为一个 TensorFlow 的 Tensor，可以使用函数.numpy()把结果转化成 Numpy 数组。可以看到，卷积层的计算结果为一个$1 \times 4 \times 4 \times 1$的数组。把卷积层变换为$4 \times 4$的数组后，可以方便查看卷积层的计算结果。

```
conv_layer(images).numpy().shape

(1, 4, 4, 1)

conv_layer(images).numpy().reshape(4,4)

array([[-0.12557002,  0.12896288, -1.50176408, -1.3329796 ],
       [-0.87352764, -1.65900396, -0.01473809, -0.67228483],
       [-0.71944782, -3.10842334, -1.39798264, -0.37900039],
       [-1.24817389,  0.09585495, -0.72222156,  0.41610027]])
```

通过函数 conv_layer.get_weights()可以得到该卷积层的核及其截距项的初始值。

```
conv_layer.get_weights()[0].shape
(3, 3, 1, 1)
print("Kernel Weights:\n",
conv_layer.get_weights()[0].reshape(3,3))
print("Bias:\n", conv_layer.get_weights()[1])

Kernel Weights:
 [[-0.20663663 -0.52216303 -0.18993356]
 [ 0.35366662 -0.25309275 -0.4699023 ]
 [-0.44492979 -0.28197704  0.03117762]]
Bias:
 [0.]
```

我们尝试自行计算卷积层的部分元素验证函数 tf.keras.layers.Conv2D()的计算结果。首先，为了让输出结果和输入数据的维度一样，先对输入数据 images 进行零填充。需要填充几行和几列呢？回忆第 8 章学习的公式：$W_{\text{out}} = (W - K + 2P)/S + 1$和$H_{\text{out}} = (H - K + 2P)/S + 1$。现在，$W = 4$，$K = 3$，$S = 1$，$W_{\text{out}} = 4$，因此，可以解得$P = 1$。在卷积运算前，需要在输入数据上下左右各加一列或者一行 0，如下面的代码所示。

```
padding = np.zeros((6, 6))
```

```
padding[1:5,1:5] = images.reshape(4,4)
print("Padded Input: \n", padding)

Padded Input:
 [[ 0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   1.  -1.   2.   0. ]
 [ 0.  -1.   2.   1.   0.1  0. ]
 [ 0.   0.   0.1  1.  -1.   0. ]
 [ 0.   3.   1.   1.   1.   0. ]
 [ 0.   0.   0.   0.   0.   0. ]]
```

根据卷积的运算规则，我们尝试验证几个输出结果，例如：

- 卷积运算结果的第 1 行第 1 列的数字 $-0.12557002 = 0 \times (-0.20663663) + 0 \times (-0.52216303) + 0 \times (-0.18993356) + 0 \times (0.35366662) + 0 \times (-0.25309275) + 1 \times (-0.4699023) + 0 \times (-0.44492979) + (-1) \times (-0.28197704) + 2 \times 0.03117762$
- 卷积运算结果的第 3 行第 3 列的数字 $-1.39798264 = 2 \times (-0.20663663) + 1 \times (-0.52216303) + 0.1 \times (-0.18993356) + 0.1 \times (0.35366662) + 1 \times (-0.25309275) + (-1) \times (-0.4699023) + 1 \times (-0.44492979) + 1 \times (-0.28197704) + 1 \times 0.03117762$

可以看到，函数 `tf.keras.layers.Conv2D()`的计算结果和我们自行计算的结果确实是一样的。

在函数 `tf.keras.layers.Conv2D()`中，可以通过参数 `activation` 设置卷积运算之后的激活函数。例如，下面的代码设置 `conv_layer2` 的激活函数为 ReLU 函数。

```
conv_layer2 = tf.keras.layers.Conv2D(1, kernel_size=(3, 3), \
              strides=(1, 1), input_shape=(4,4,1), \
              padding='same', activation='relu')
```

函数 `tf.keras.layers.MaxPooling2D()`可以实现最大池化。在下面代码中：

- 第一个参数(2,2)表示池化层的核的宽度和高度都为 2；
- 函数 `tf.keras.layers.MaxPooling2D()`还有一个关键的参数 `strides`。在这里，`strides` 设为`(2,2)`。实际上，参数 `strides` 的默认值等于 `pool_size`。因此，在下面的代码中也可以省略参数 `strides` 的设置。

```
max_pool_layer = tf.keras.layers.MaxPooling2D((2,2), \
                                              strides=(2, 2))
```

通过下面的代码可以看到池化层的结果。我们也可以如下自行计算最大池化层，

- 池化层结果第 1 行第 1 列：$0.12896288 = \max(-0.12557002, 0.12896288, -0.87352764, -1.65900396)$；

- 池化层结果第 1 行第 2 列：$-0.01473809 = \max(-1.50176408, -1.3329796, -0.01473809, -0.67228483)$；
- 池化层结果第 2 行第 1 列：$0.09585495 = \max(-0.71944782, -3.10842334, -1.24817389, 0.09585495)$；
- 池化层结果第 2 行第 2 列：$0.41610027 = \max(-1.39798264, -0.37900039, -0.72222156, 0.41610027)$。

```
max_pool_layer(conv_layer(images).numpy()).numpy().reshape(2,2)
array([[ 0.12896288, -0.01473809],
       [ 0.09585495,  0.41610027]])
```

9.2 CNN 实例：mnist 数据和 CIFAR-10 数据

9.2.1 卷积神经网络：mnist 数据

在第 7 章中，我们没有使用 CNN，而是把 mnist 数据28×28的输入数据转化成784的向量，然后建立具有两个隐藏层并且使用了 Dropout 的全连接神经网络，得到的测试准确率为98.3%。本节将尝试用 TensorFlow 为 mnist 数据建立卷积神经网络模型，看看是否可以进一步提高预测精度。

首先载入 mnist 数据，并对数据进行预处理。与之前的章节一样，把数据分成三个部分，训练数据（50000 幅图片）、验证数据（10000 幅图片）和测试数据（10000 幅图片）。这里把输入数据的维度变为`(None, 28, 28, 1)`，因变量不需要 one-hot 编码。

```
"""
载入数据，并做预处理
"""
x_train = idx2numpy.convert_from_file(\
                      './data/mnist/train-images.idx3-ubyte')
y_train  = idx2numpy.convert_from_file(\
                      './data/mnist/train-labels.idx1-ubyte')
x_test = idx2numpy.convert_from_file(\
                      './data/mnist/t10k-images.idx3-ubyte')
y_test = idx2numpy.convert_from_file(\
                      './data/mnist/t10k-labels.idx1-ubyte')

np.random.seed(1)
train_images = x_train.reshape(-1, 28, 28, 1)/255
train_labels = y_train


"""
把训练数据分成训练数据和验证数据
"""
index = np.arange(len(train_images))
np.random.shuffle(index)
# 验证数据
valid_images = train_images[index[-10000:]]
```

```
valid_labels = train_labels[index[-10000:]]
# 训练数据
train_images, train_labels = train_images[index[:50000]], \
                             train_labels[index[:50000]]

# 测试数据
test_images = x_test.reshape(len(x_test), 28, 28, 1)/255
test_labels = y_test
```

现在使用函数 `tf.keras.models.Sequential()` 建立如图 9-3 所示的卷积神经网络模型。该卷积神经网络具有两个卷积层、两个池化层和 1 个全连接层。两个卷积层的核的宽和高都是3，卷积层的深度分别为 32 和 64。两个池化层的宽和高都是2，步幅都设为2。

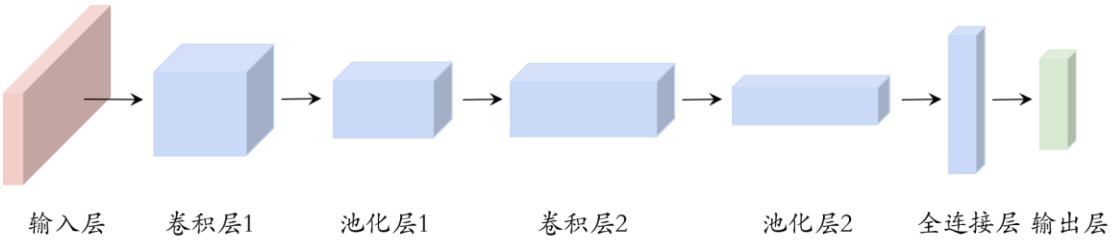

图 9-3 卷积神经网络，具有两个卷积层、两个池化层和 1 个全连接层

```
mnist_cnn_model = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(32, kernel_size=(3, 3), \
                   strides=(1, 1), padding='same',
                   activation='relu', input_shape=(28, 28, 1)),
            tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
            tf.keras.layers.Conv2D(64, kernel_size=(3, 3), \
                   strides=(1, 1), padding='same', \
                   activation='relu'),
            tf.keras.layers.MaxPooling2D((2, 2)),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.Dense(10, activation='softmax')])
```

在模型中，隐藏层的激活函数都为 ReLU 函数。函数 `mnist_cnn_model.summary()` 可以得到卷积神经网络每一层的节点数量和维度。

- 卷积层 1 的维度为(28,28,32)，该卷积层 `padding='same'`。每一个核的参数个数为3×3，共有 32 个核，每个核有 1 个截距项，因此，该卷积层参数个数为$3 \times 3 \times 32 + 32 = 320$。
- 池化层 1 的维度为(14,14,32)。2×2的核和2×2的步幅，使得池化层维度减半。池化层没有参数。
- 卷积层 2 的维度为(14,14,64)，该卷积层 `padding='same'`。每一个核的参数个数为$3 \times 3 \times 32$，共有 64 个核，每个核还有 1 个截距项，因此，该卷积层参数个数为$3 \times 3 \times 32 \times 64 + 64 = 18496$。关于输入是 3 维的卷积运算请参见本节最后的备注。

- 池化层 2 的维度为(7,7,64)。2×2的核和2×2的步幅，使得池化层维度减半。池化层没有参数。
- 函数 `tf.keras.layers.Flatten()`把池化层 2 的节点向量化，向量长度为$7 \times 7 \times 64 = 3136$。
- 全连接层的维度为128，参数个数为$3136 \times 128 + 128 = 401536$。
- 输出层的维度为10，参数个数为$128 \times 10 + 10 = 1390$。

该卷积神经网络总的参数个数为 421642。

```
mnist_cnn_model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 128)               401536
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 421,642
Trainable params: 421,642
Non-trainable params: 0
_________________________________________________________________
```

接着使用函数 `mnist_cnn_model.compile()`设置最优化方法、损失函数和评价指标，使用函数`mnist_cnn_model.fit()`训练模型。

```
mnist_cnn_model.compile(optimizer=tf.keras.optimizers.Adam(),
                        loss='sparse_categorical_crossentropy',
                        metrics=['accuracy'])

mnist_cnn_history = mnist_cnn_model.fit(train_images,     \
                  train_labels, epochs=5, batch_size = 128, \
                  validation_data=(valid_images, valid_labels))

Train on 50000 samples, validate on 10000 samples
Epoch 1/5
50000/50000 [====] - 23s 455us/sample - loss: 0.2121 - accuracy:
0.9374 - val_loss: 0.0701 - val_accuracy: 0.9799
Epoch 2/5
50000/50000 [====] - 21s 419us/sample - loss: 0.0554 - accuracy:
0.9828 - val_loss: 0.0560 - val_accuracy: 0.9844
```

```
Epoch 3/5
50000/50000 [====] - 21s 426us/sample - loss: 0.0387 - accuracy:
0.9880 - val_loss: 0.0458 - val_accuracy: 0.9871
Epoch 4/5
50000/50000 [====- 21s 426us/sample - loss: 0.0301 - accuracy:
0.9908 - val_loss: 0.0449 - val_accuracy: 0.9862
Epoch 5/5
50000/50000 [====] - 21s 417us/sample - loss: 0.0224 - accuracy:
0.9927 - val_loss: 0.0332 - val_accuracy: 0.9908


mnist_cnn_model.evaluate(test_images, test_labels)
10000/10000 [====] - 1s 136us/sample - loss: 0.0277 - accuracy:
0.9898

[0.027659693840309047, 0.9898]
```

从上面代码的运行结果可以得到，该卷积神经网络测试准确率为 99%！在这里，卷积神经网络的参数个数只有大约 42 万个（而且，大部分的参数都是全连接层的）。作为对比，在第 7 章中，我们建立的全连接神经网络的参数个数超过 140 万个，测试准确率为98.3%。从图 9-4 可以看到，刚建立的卷积神经网络的过拟合程度较小。

```
plt.rcParams['font.sans-serif'] = ['SimHei'] #用来正常显示中文标签

plt.plot(mnist_cnn_history.epoch, \
         mnist_cnn_history.history['accuracy'], \
         label="训练准确率")
plt.plot(mnist_cnn_history.epoch, \
        mnist_cnn_history.history['val_accuracy'], \
        label="验证准确率")

plt.xlabel("迭代次数", fontsize=16)
plt.ylabel("预测准确率", fontsize=16)
_ = plt.legend()
```

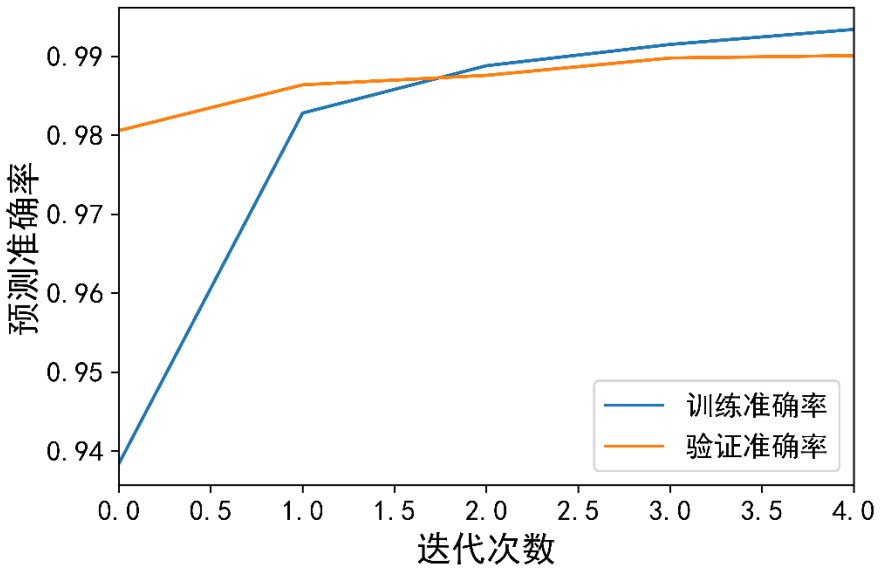

图 9-4 训练误差和验证误差的变化曲线（mnist 手写识别数据）

备注：输入数据是 3 维数组的卷积运算。在模型 `mnist_cnn_model` 中，卷积层 1 的维度为$28 \times 28 \times 32$，池化层 1 的维度为$14 \times 14 \times 32$。接着，池化层 1 的结果成为卷积层 2 的输入，因此，卷积层 2 的输入数据为一个 3 维数组。我们在第 8 章学过输入数据为二维数组的卷积运算规则。当输入数据的维度是 3 维时，要怎么做卷积运算呢？本质上，输入数据为 3 维数组的卷积运算规则与输入数据为 2 维数组的卷积运算一样，只是在细节上略有不同。我们通过图 9-5 所示的简单例子来学习输入数据是 3 维数组的卷积运算。

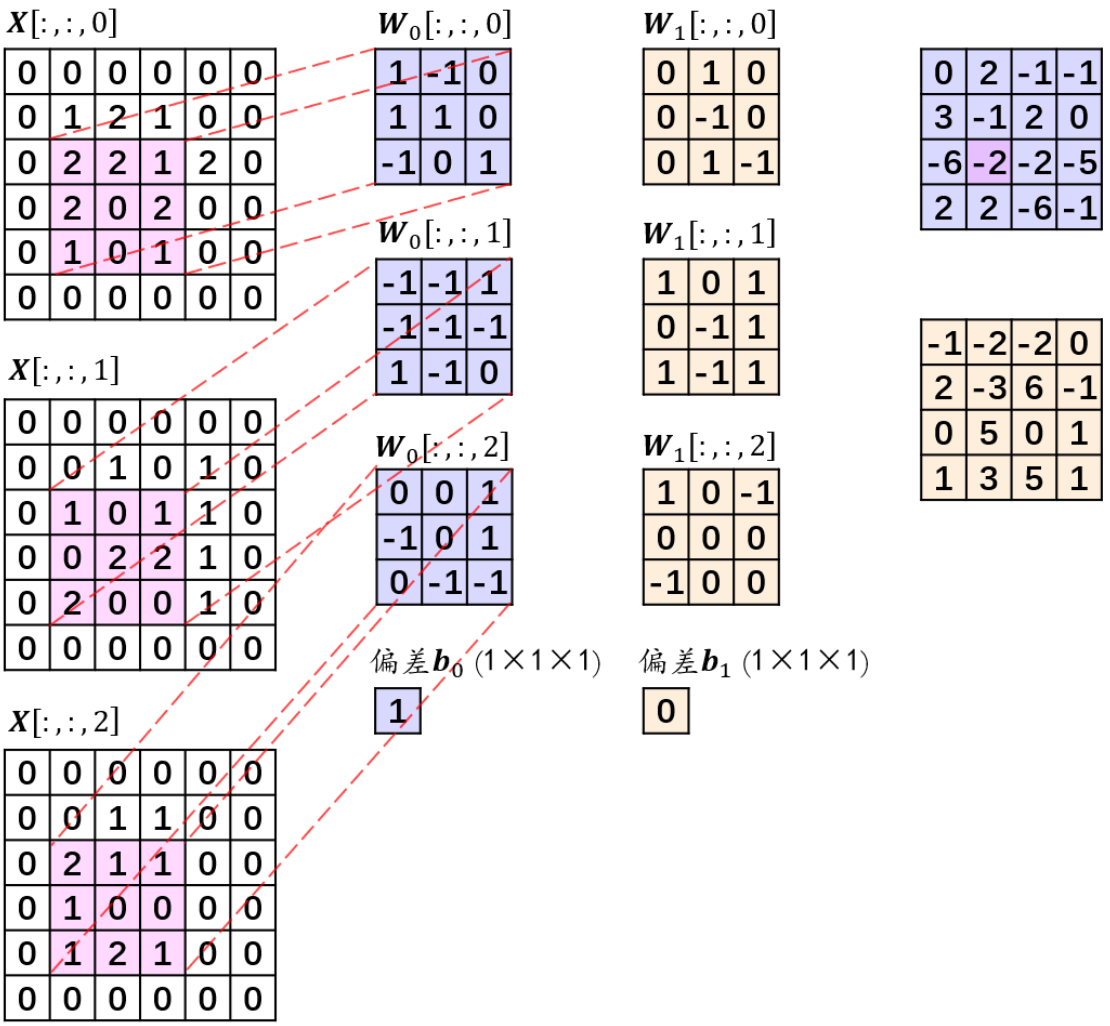

图 9-5 输入数据是 3 维数组的卷积运算

在图 9-5 的例子中，输入数据是$4 \times 4 \times 3$。为了让输出结果的宽和高与输入数据保存一致，使用零填充在输入数据上下左右都加一行或者一列 0。图 9-5 的例子有两个核，每一组核的维度都是$3 \times 3 \times 3$，外加一个截距项。每一组核的宽和高都小于输入数据的宽和高，但是核的深度一定与输入数据的深度保持一致。输出结果的维度为$4 \times 4 \times 2$。记输入数据为$\boldsymbol{X}$，第 1 组核为W_0，第 2 组核为W_1。输入数据$\boldsymbol{X}$与核W_0的卷积运算可以分解为 3 个二维的卷积运算。

- $x[:,:,0]$与$W_0[:,:,0]$做二维的卷积运算，得到一个4×4数组；
- $x[:,:,1]$与$W_0[:,:,1]$做二维的卷积运算，得到一个4×4数组；
- $x[:,:,2]$与$W_0[:,:,2]$做二维的卷积运算，得到一个4×4数组。

3 个二维卷积运算的结果都是4×4数组，把 3 个4×4数组对应的元素相加，相加的结果再加上截距项即为 3 维卷积运算的结果。例如，在输入数据$\boldsymbol{X}$与核W_0的卷积运算结果中，第 3 行第 2 列的元素-2可以通过以下方式计算得到。

- $\boldsymbol{X}[:,:,0]$与$W_0[:,:,0]$的卷积运算：$2 \times 1 + 2 \times (-1) + 1 \times 0 + 2 \times 1 + 0 \times 1 + 2 \times 0 + 1 \times (-1) + 0 \times 0 + 1 \times 1 = 2$
- $\boldsymbol{X}[:,:,1]$与$W_0[:,:,1]$的卷积运算：$1 \times (-1) + 0 \times (-1) + 1 \times 1 + 0 \times (-1) + 2 \times (-1) + 2 \times (-1) + 2 \times 1 + 0 \times (-1) + 0 \times 0 = -2$
- $\boldsymbol{X}[:,:,2]$与$W_0[:,:,2]$的卷积运算：$2 \times 0 + 1 \times 0 + 1 \times 1 + 1 \times (-1) + 0 \times 0 + 0 \times 1 + 1 \times 0 + 2 \times (-1) + 1 \times (-1) = -3$

卷积运算的最终结果为$2 + (-2) + (-3) + 1 = -2$。第二组核W_1与$\boldsymbol{X}$做同样的运算可以得到输出结果的第二个4×4数组。总的来说，当输入数据是 3 维时，核也是 3 维的，核的深度和输入数据的深度保持一致。进行卷积运算时，核从输入数据的宽和高方向对应到输入数据中，核与输入数据对应部分的所有元素相乘，然后再把乘积的结果相加，得到卷积层的一个元素的值，如图 9-6 所示。

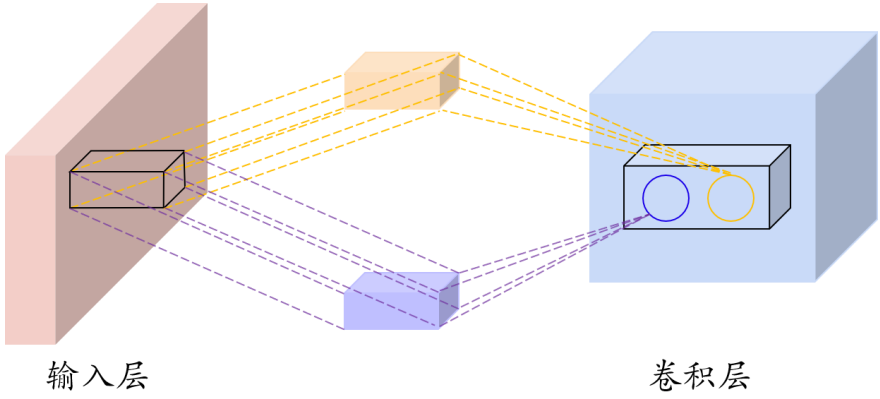

图 9-6 卷积层运算，输入数据为 3 维数组，两个核

9.2.2 卷积神经网络：CIFAR-10 数据

CIFAR-10 数据是由 Alex Krizhevsky, Vinod Nair 和 Geoffrey Hinton 收集的一个拥有 8000 万幅图片的数据的一部分。CIFAR-10 数据包含 60, 000 幅32×32像素的彩色图片。这 60, 000 幅图片有 10 个物体（飞机（airplane）、汽车（automobile）、鸟（bird）、猫（cat）、鹿（deer）、狗（dog）、青蛙（frog）、马（horse）、轮船（ship）和卡车（卡车）），每个物体有 6, 000 幅图片。

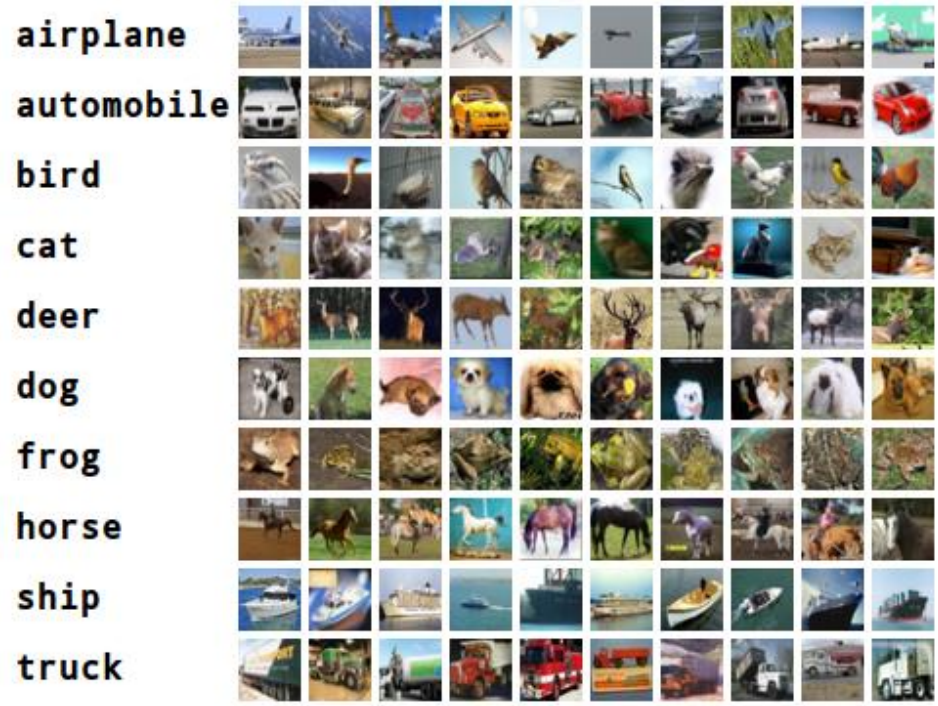

图 9-7 CIFAR-10 数据

CIFAR-10 数据的 Python 版文件名为 cifar-10-python.tar.gz，解压之后数据保存在文件夹 cifar-10-batches-py 中。在该文件夹中，数据被分成包含 50, 000 幅图片的训练数据和 10, 000 幅图片的测试数据。测试数据包含每一类物体的 1, 000 幅图片。因为训练数据占用存储空间较大，被分成 5 个文件保存。我们用函数 `pickle.load()`逐个打开训练数据文件，把数据的维度变为(None, 32, 32, 3)，同时把像素值除以 255。然后把训练数据进一步划分为90%的训练数据和10%的验证数据，把每个文件的训练数据和验证数据分别拼接在一起。

```
import pickle
np.random.seed(1)

n_batches = 5
train_images = np.empty((0, 32, 32, 3))
train_labels = np.empty((0, 10))
valid_images = np.empty((0, 32, 32, 3))
valid_labels = np.empty((0, 10))

for batch_i in range(1, n_batches + 1):
    with open('./data/cifar10/cifar-10-batches-py' + \
              '/data_batch_' + str(batch_i), mode='rb') as file:
         batch = pickle.load(file, encoding='latin1')
    # 数据的原始维度是 3*32*32，需要把数据变换成 32*32*3
    images = batch['data'].reshape((len(batch['data']), \
                        3, 32, 32)).transpose(0, 2, 3, 1)/255
    labels = batch['labels']
```

```
    validation_count = int(len(images) * 0.1)

    train_images = np.append(train_images, \
                             images[:-validation_count],  axis=0)
    train_labels = np.append(train_labels, \
                             labels[:-validation_count])
    valid_images = np.append(valid_images, \
                             images[-validation_count:],  axis=0)
    valid_labels = np.append(valid_labels, \
                             labels[-validation_count:])

with open('./data/cifar10/cifar-10-batches-py'+'/test_batch', \
                                          mode='rb') as file:
    batch = pickle.load(file, encoding='latin1')

    # 测试数据
test_images = batch['data'].reshape((len(batch['data']), \
                       3, 32, 32)).transpose(0, 2, 3, 1)/255
test_labels = np.array(batch['labels'])

class_names = ['airplane', 'automobile', 'bird', 'cat', \
               'deer','dog', 'frog', 'horse', 'ship', 'truck']
```

图 9-8 画出了 CIFAR-10 前 25 幅图片，并在图片下面标出了所属类别。可以看到，这些图片像素点较少，使得图片质量比较粗糙，但是，我们还是可以分辨出大部分图片的物体。

```
plt.figure(figsize=(8,8))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[int(train_labels[i])])
plt.show()
```

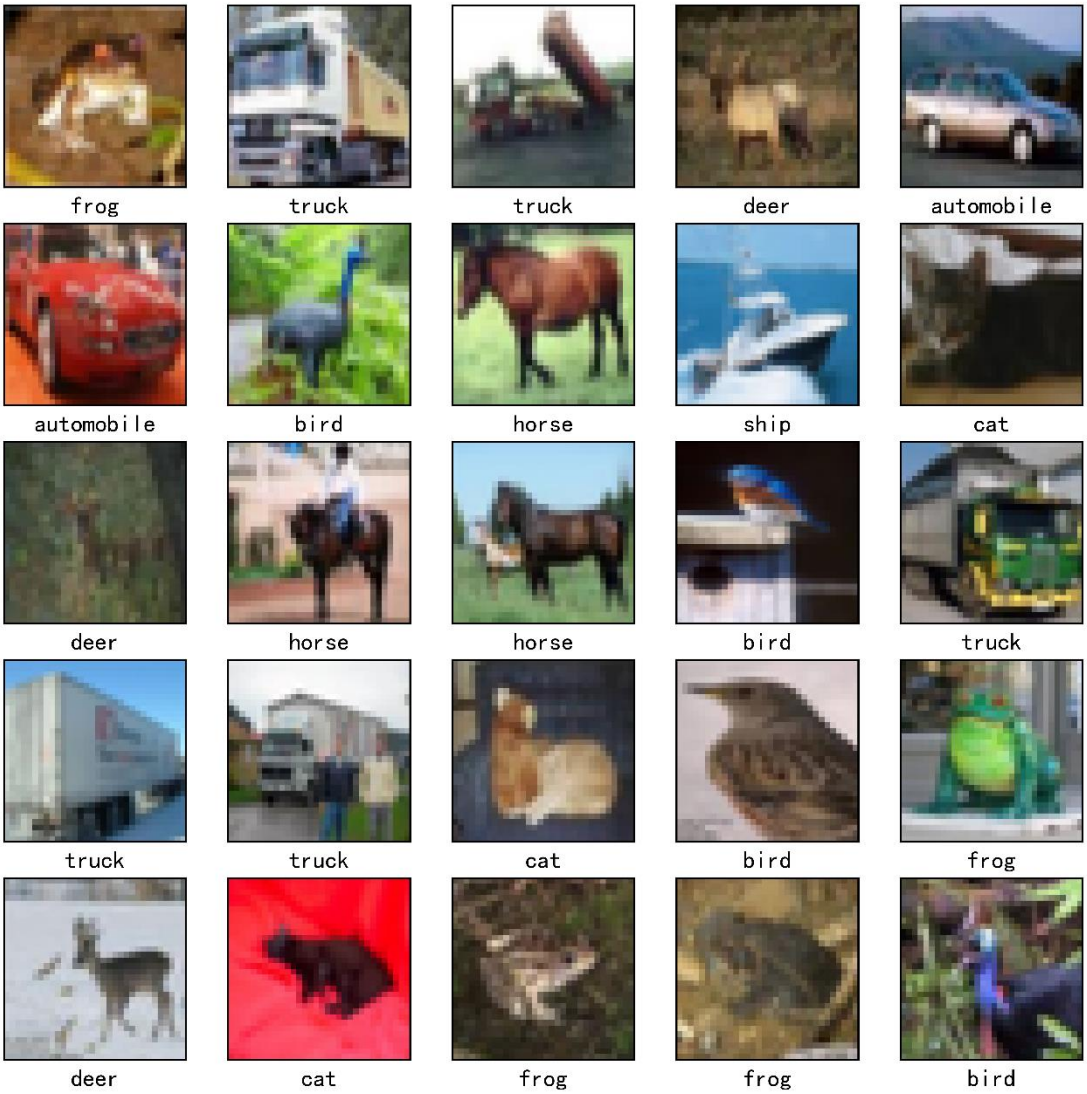

图 9-8 CIFAR-10 前 25 幅图

现在开始建立卷积神经网络模型。由于计算机性能以及训练时间的限制，我们建立了一个不算复杂的模型。该模型包含 6 个卷积层、两个个池化层、1 个全连接层和 1 个输出层。该卷积神经网络模型是到现在为止我们建立的最为复杂的神经网络模型。

```
cifar10_cnn_model = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(32, kernel_size=(3, 3), \
                strides=(1, 1), padding='same', \
                activation='relu', input_shape=(32, 32, 3)),\
            tf.keras.layers.Conv2D(32, kernel_size=(3, 3), \
                strides=(1, 1), padding='same',\
                activation='relu'),
```

```
            tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),\
            tf.keras.layers.Conv2D(64, kernel_size=(3, 3), \
                strides=(1, 1), padding='same',\
                activation='relu'),
             tf.keras.layers.Conv2D(64, kernel_size=(3, 3),\
                  strides=(1, 1), padding='same',\
                  activation='relu'),
             tf.keras.layers.MaxPooling2D((2, 2)),
             tf.keras.layers.Conv2D(128, kernel_size=(3, 3),\
                  strides=(1, 1), padding='same',
                  activation='relu'),
             tf.keras.layers.Conv2D(128,  kernel_size=(3, 3),\
                  strides=(1, 1), padding='same',
                  activation='relu'),
             tf.keras.layers.Flatten(),
             tf.keras.layers.Dense(128, activation='relu'),
             tf.keras.layers.Dropout(0.5),
             tf.keras.layers.Dense(10, activation='softmax')])
```

从函数 `cifar10_cnn_model.summary()`的结果可以看到，该模型的参数个数超过130万个。

```
cifar10_cnn_model.summary()
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               1048704
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,337,002
Trainable params: 1,337,002
Non-trainable params: 0
```

```
_________________________________________________________________
```

接着使用函数 cifar10_cnn_model.compile()设置最优化方法、损失函数和评价指标，然后使用函数 cifar10_cnn_model.fit()训练模型。

```
cifar10_cnn_model.compile(optimizer=tf.keras.optimizers.Adam(),
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])
cifar10_cnn_history = cifar10_cnn_model.fit(train_images, \
              train_labels, epochs=10, batch_size = 64,\
              validation_data=(valid_images, valid_labels),\
              verbose=0)
cifar10_cnn_model.evaluate(test_images, test_labels)

10000/10000 [====] - 6s 568us/sample - loss: 0.7723 - accuracy:
0.7666

[0.7723461737632752, 0.7666]
```

从图 9-9 可以看到，验证准确率在第 7 次循环之后便上升得很平缓，训练准确率一直在上升；说明第 7 次循环之后模型开始过拟合。最终的测试准确率为76%。可见 CIFAR-10 数据的难度比 mnist 数据大很多。现在，文献记载的CIFAR-10 分类模型测试准确率大约可以达到90%，而 mnist 分类模型测试准确率几乎已经达到100%。

```
plt.plot(cifar10_cnn_history.epoch, \
         cifar10_cnn_history.history['accuracy'], \
         label="训练准确率")
plt.plot(cifar10_cnn_history.epoch, \
         cifar10_cnn_history.history['val_accuracy'], \
         label="验证准确率")

plt.xlabel("迭代次数")
plt.ylabel("预测准确率")
_ = plt.legend()
```

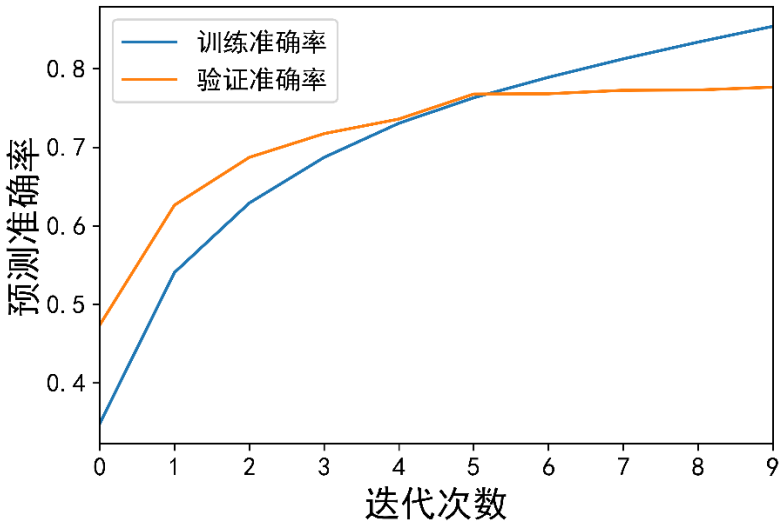

图 9-9 训练误差和测试误差（CIFAR 10）

现在，我们具体看一些测试数据的结果。可以使用函数 cifar10_cnn_model.predict()得到预测值。测试数据的维度为(10000, 32, 32, 3)，得到的预测结果的维度为(10000,10)。

```
predictions = cifar10_cnn_model.predict(test_images)
predictions.shape
(10000, 10)
```

预测结果 predictions 的每一行表示对应图片的预测概率。因为模型输出层的激活函数为 softmax，因此预测结果每一行的和为 1。

```
predictions[0]
array([5.0578096e-06, 1.8853748e-06, 5.1180672e-05, 9.7904932e-
01, 8.1300550e-06, 1.5404076e-02, 5.1538371e-03, 6.9920893e-06,
3.1839608e-04, 9.9961483e-07], dtype=float32)
```

预测结果每一行最大值对应的位置表示对应图片的预测。可以看到，对第一幅图片预测概率最大值是 5.2554417e-01，对应的位置是 3。其真实的类别也为 3。说明模型对第一幅图的预测是正确的。

```
np.argmax(predictions[0])
3
test_labels[0]
3
```

下面通过一些图更直观地展示预测效果。首先定义两个画图函数：plot_image()和 plot_value_array()。

- 函数 plot_image()可以画出图片，并在图片下面标注模型对该图片的预测和预测概率。标注中括号外的文字表示预测类别；括号内的文字表示真实类别。
- 函数 plot_value_array()可以显示模型把图片预测为每一个类别的概率，用柱状图的方式表示。

```
def plot_image(i, predictions_array, true_label, img):
    predictions_array = predictions_array[i]
    true_label = true_label[i]
    img = img[i]

    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format(\
                                class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
    predictions_array = predictions_array[i]
    true_label = true_label[i]

    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, \
                       color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
```

图 9-10 是第一幅图的结果。左图为一只猫，模型预测该图片为猫的概率为53%；右图画出了各个概率的柱状图。有的类别概率很小，因此在右图中没有显示。

```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
```

```
plot_value_array(i, predictions,  test_labels)
plt.show()
```

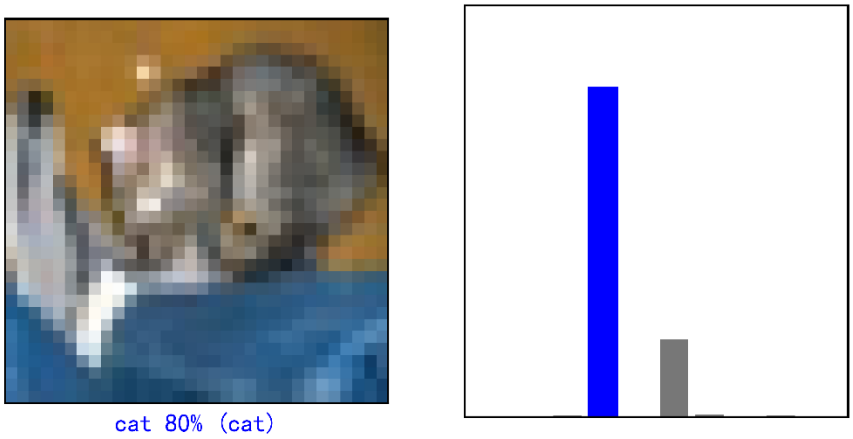

图 9-10 CIFAR 10 第一幅图预测概率

图 9-11 显示模型对前 15 幅图的预测结果。可以看到，前 15 幅图片中有 11 幅图片模型预测正确了。第 3 幅、第 6 幅、第 7 幅和第 8 幅预测错误。其中，第 3 幅图真实应该是船(ship)，模型预测为飞机（airplane）；第 8 幅图真实应该是青蛙（frog），模型预测为鹿（deer）。

```
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions, test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions, test_labels)
plt.show()
```

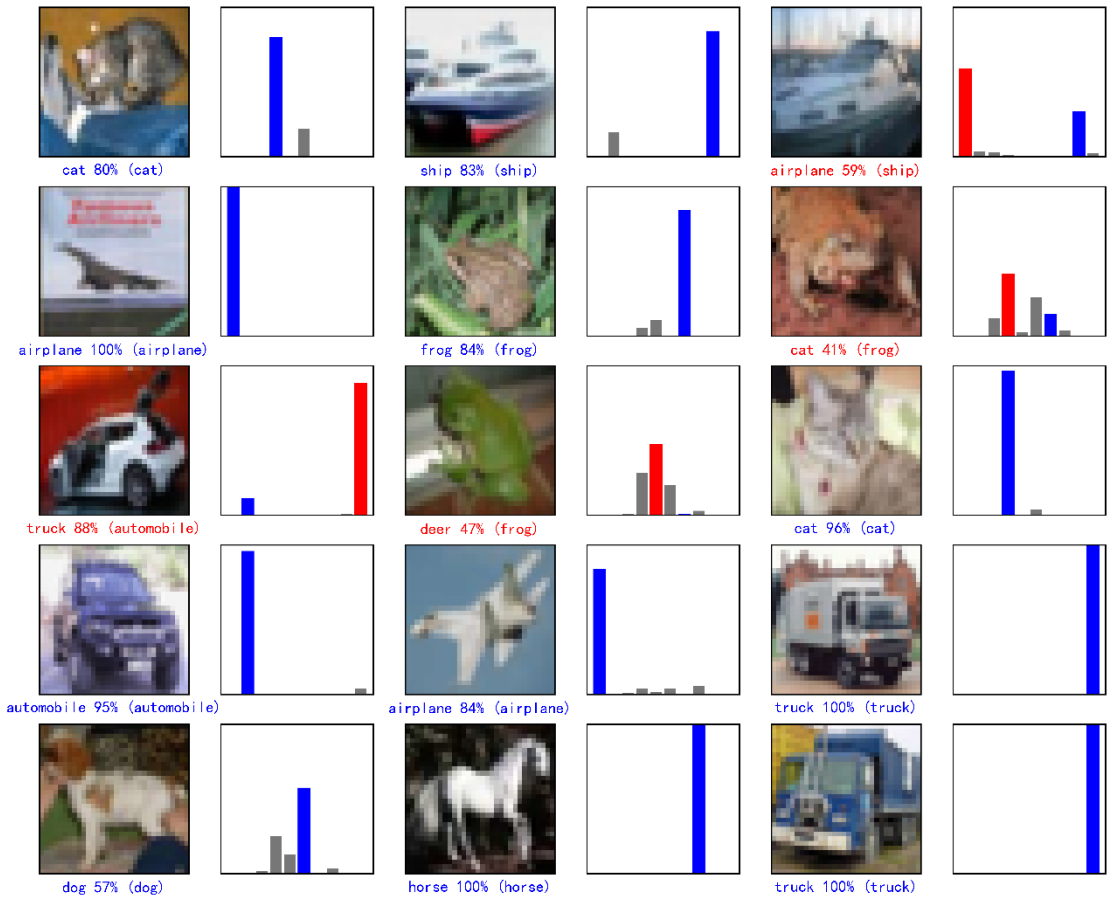

图 9-11 CIFAR 10 测试数据前 15 幅图预测概率

9.3 CNN 建模技巧

9.3.1 卷积神经网络的结构

到现在为止，我们已经实现了多个卷积神经网络模型。总的来说，卷积神经网络包含 3 种层，卷积层、池化层和全连接层。对于一般情况，我们应该如何合理安排卷积层、池化层和全连接层？如何选择合适的超参数使得卷积神经网络预测的精度更高呢？

我们通常会使用多层卷积层，在卷积层之间夹杂着池化层。这样会使得神经网络的层的宽度和高度变小，而深度变大。最后使用全连接层。常用的卷积神经网络可以表示成如下形式：

$$\text{Input} \to [[\text{CONV} \to \text{ReLU}] \times N \to \text{POOL?}] \times M \to [\text{FC} \to \text{ReLU}] \times K \to \text{FC}$$

其中，乘号×表示重复次数，POOL?表示这里的池化层是可选的，N，M，K表示重复的次数。通常情况下，$N \geq 1$且$N \leq 3$，$M \geq 1$，$K \geq 1$且$K \leq 3$。 下面是一些常见的卷积神经网络结构。

- $\text{Input} \rightarrow \text{FC}$，线性回归模型。这时，$N = M = K = 0$。
- $\text{Input} \rightarrow [\text{CONV} \rightarrow \text{ReLU}] \rightarrow \text{FC}$，只有一个卷积层。这时，$N = M = 1$，$K = 0$，没有池化层。
- $\text{Input} \rightarrow [[\text{CONV} \rightarrow \text{ReLU}] \rightarrow \text{POOL}] \rightarrow [\text{FC} \rightarrow \text{ReLU}] \rightarrow \text{FC}$，一个卷积层，一个池化层，一个全连接层。这时，$N = M = K = 1$。
- $\text{Input} \rightarrow [[\text{CONV} \rightarrow \text{ReLU}] \rightarrow \text{POOL}] \times 2 \rightarrow [\text{FC} \rightarrow \text{ReLU}] \rightarrow \text{FC}$，卷积层加池化层重复两次，一个全连接层。这时，$N = 1$，$M = 2$，$K = 1$。
- $\text{Input} \rightarrow [[\text{CONV} \rightarrow \text{ReLU}] \times 2 \rightarrow \text{POOL}] \times 3 \rightarrow [\text{FC} \rightarrow \text{ReLU}] \times 2 \rightarrow \text{FC}$ ，两个卷积层后加一个池化层重复 3 次，再加两个全连接层。这时，$N = 2$，$M = 3$，$K = 2$。连续使用多个卷积层，可以使得模型不会快速减少维度，帮助建立更深的卷积神经网络模型；这种结构可以使得模型辨认出更加复杂的数据结构，最终可能使得模型表现得更好。

9.3.2 卷积层和池化层的超参数选择

在 9.3.1 节中，我们讨论了卷积神经网络模型总的结构。本节将探讨卷积层和池化层的超参数选择。

- 卷积层有两个重要的参数，核维度（kernel_size，F）和步幅（strides，S）。通常情况下，我们会使用较小的核维度，例如，3×3或者5×5；让核小步移动，例如，$S = 1$；同时，我们还会设置 `padding='same'`。这样的设置可以让模型以较少的参数找出数据变量间复杂的结构。有时候也可以把前几个卷积层的核维度稍微设置的大一些，例如，7×7。

- 池化层也有两个重要的参数，核维度（kernel_size，F）和步幅（strides，S）。池化层的重要作用是去掉噪声、简化维度和保留信息。我们通常把核维度设置为2×2，步幅设置为$S = 2$。同时，`padding='valid'`。按照这样的设置，每添加一个池化层，节点的宽度和高度的维度都减少一半，也就是说，75%的节点将会被丢掉。有时候，我们也会设置池化层核维度3×3，$S = 2$。池化层的核维度越大，丢掉的节点数也将越多，因此，通常不会让池化层的核维度超过 3。

总的来说，我们会在卷积层中保持层的宽度和高度不变，然后在池化层中减少节点数，从而去除噪声和保留信号。在实践中，最大池化比平均池化用得多一些。

9.3.3 一些经典的卷积神经网络

在卷积神经网络的发展历史中，有一些卷积神经网络取得了突破性的进展。这些神经网络通常以发明者的名字命名。下面列举几个经典的卷积神经网络，以激发我们建立卷积神经网络模型的灵感。

1. LeNet 发明于 20 世纪 90 年代，由称为深度学习之父的雅恩·乐昆(Yann LeCun)提出。LeNet 是卷积神经网络的第一个成功的例子，主要用于邮编和数字的识别。LeNet 的结构为

$$\text{Input} \to [[\text{CONV} \to \text{sigmoid}] \to \text{POOL}] \times 2 \to [\text{FC} \to \text{sigmoid}] \times 2 \to \text{FC}$$

虽然现在看该模型很简单，但是在当时这已经是一个非常复杂的模型了。

2. AlexNet 是 ILSVRC 2012（ILSVRC 是著名的大规模视觉识别挑战赛）的冠军，由阿莱克斯·克里泽夫斯基(Alex Krizhevsky)、伊利亚·莎士科尔(Ilya Sutskever) 和杰弗里·辛顿(Geoffrey Hinton)发明。杰弗里·辛顿也被称为深度学习之父。AlexNet 的发明使得卷积神经网络在计算机视觉领域开始流行。AlexNet 的结构比 LeNet 更加复杂。

$$\text{Input} \to \big[[\text{CONV} \to \text{ReLU}] \to \text{POOL}\big] \times 2 \to \big[[\text{CONV} \to \text{ReLU}] \times 3 \to \text{POOL}\big]$$

$$\to [\text{FC} \to \text{ReLU}] \times 2 \to \text{FC}$$

AlexNet 使用了激活函数 ReLU，并且在全连接层用 Dropout 控制过拟合。

3. ZF Net 是 ILSVRC 2013 的冠军，由马修·泽勒(Matthew Zeiler)和罗伯·费格斯(Rob Fergus)发明。ZF net 在 AlexNet 的基础上调整了很多超参数，例如 ZF Net 增大了中间卷积层的深度，并且在前几个卷积层中使用更小维度的核和更小的步幅。

4. GoogLeNet 是 ILSVRC 2014 的冠军，发明团队来自谷歌。GoogLeNet 的主要贡献在于使用了 Inception Module，即在某个卷积层中同时使用多个不同维度的核。GoogLeNet 没有使用全连接层，采用平均池化，而不是常用的最大池化。

5. VGGNet 是 ILSVRC 2014 的亚军，由凯伦·西蒙尼扬(Karen Simonyan) 和安德鲁·西塞曼(Andrew Zisserman) 发明。VGGNet 的一个重要贡献是让大家意识到神经网络的深度非常重要。VGGNet 有 13 个卷积层、5 个池化层和 2 个全连接层。VGGNet 的具体结构如下面代码所示，总的参数个数超过了 1.3 亿。

```
VGGNet = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), \
```

```
        input_shape=(224, 224, 3), padding='same', \
        activation='relu'),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.MaxPool2D((2,2)),
    tf.keras.layers.Conv2D(128, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.Conv2D(128, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.MaxPool2D((2,2)),
    tf.keras.layers.Conv2D(256, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.Conv2D(256, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.Conv2D(256, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.MaxPool2D((2,2)),
    tf.keras.layers.Conv2D(512, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.Conv2D(512, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.Conv2D(512, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.MaxPool2D((2,2)),
    tf.keras.layers.Conv2D(512, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.Conv2D(512, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.Conv2D(512, kernel_size=(3, 3), \
        padding='same', activation='relu'),
    tf.keras.layers.MaxPool2D((2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dense(1000, activation='softmax'),
    ])
VGGNet.summary()
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 224, 224, 64)      1792
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 224, 224, 64)      36928
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 112, 112, 64)      0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 112, 112, 128)     73856
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 112, 112, 128)     147584
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 56, 56, 128)       0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 56, 56, 256)       295168
_________________________________________________________________
```

```
conv2d_15 (Conv2D)           (None, 56, 56, 256)       590080
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 56, 56, 256)       590080
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 28, 28, 256)       0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 28, 28, 512)       1180160
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 28, 28, 512)       2359808
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 28, 28, 512)       2359808
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 14, 14, 512)       0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 7, 7, 512)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_4 (Dense)              (None, 4096)
102764544
_________________________________________________________________
dense_5 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dense_6 (Dense)              (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

9.4 本章小结

在本章中，我们学习使用 TensorFlow 建立卷积神经网络。具体来说，可以通过函数 `tf.keras.layers.Conv2D()`和 `tf.keras.layers.MaxPooling2D()`分别实现卷积层和池化层。在卷积神经网络中，通常使用多层卷积层提取数据的特征，在一个或者多个卷积层之后使用池化层降低隐藏层的维度，最后几层一般是全连接层，最终卷积神经网络可能包含几十层的隐藏层。近年来，卷积神经网络不包含全连接层也是一个趋势。从 VGGNet 中可以看到，虽然整个神经网络包含超过 1.3 亿参数，但是大部分参数都是由全连接层贡献的。因此，去除全连接层，可以大幅减少卷积神经网络的参数数量。

从卷积神经网络中，人们认识到神经网络的深度非常重要，深度神经网络可以提高模型的预测准确率。神经网络的隐藏层数量增加，也意味着参数数量的增加，这导致神经网络模型更复杂，再加上数据的维度和样本量的增长，最终导致

训练神经网络的计算量异常巨大。为了提高优化速度并降低开发成本，现在人们普遍使用具有更高访存速度、更强浮点运算能力且具有更多核心数的 GPU 来训练深度神经网络。谷歌于 2016 年甚至发布了为机器学习专门设计的 TPU（Tensor Processing Unit），TPU 可以大幅降低功耗并加快运算速度，从而促进机器学习特别是深度学习的研究和应用。

习题

1. 分析 mnist 数据，使用 TensorFlow 建立卷积神经网络模型，尝试不同的模型结构。例如，不同的卷积层个数、池化层个数，以及卷积层和池化层的参数；尝试使用 Dropout 方法和L_2惩罚，等等。可以进一步提高 mnist 数据的测试准确率吗？
2. 分析 Fashion-mnist 数据，使用 TensorFlow 建立卷积神经网络模型。尝试不同的模型结构，测试准确率最高能达到多少？
3. 分析 CIFAR-10 数据，使用 TensorFlow 建立卷积神经网络模型。尝试不同的模型结构。
4. 分析 CIFAR-10 数据，使用 TensorFlow 建立卷积神经网络模型。尝试经典的模型结构，并尝试调整超参数或者一切可能有效的办法，模型预测准确率可以超越经典卷积神经网络模型吗？