第 3 章

使用 LeNet 进行图像分类

千里之行，始于足下。

——老子

深度学习是一个不断发展的领域。从基本的神经网络开始，到现在能够解决大量业务问题的复杂架构，基于深度学习的图像处理和计算机视觉能力能够创造更好的癌症检测解决方案、降低污染水平、实施监控系统并改善消费者的体验。有时候，解决业务问题需要定制具有个性化的方法。根据现有的图像质量设计出需要的网络模型，以适应当前的业务问题需要。网络模型设计还要考虑使用可用的计算能力训练网络模型和执行网络模型的计算。跨组织集团和大学里的研究人员花费了大量的时间来收集和管理数据集，对数据进行了必要的清理和分析，对网络模型和架构进行系统的设计、训练和测试，并进行迭代计算以不断提高网络模型的性能。通常需要花费大量的时间和大量的耐心来制定一个具有开创性的深度学习解决方案。

前两章讨论了关于神经网络的基础知识，并使用 Keras 和 Python 创建了一个深度学习解决方案。这一章开始讨论更加复杂的神经网络结构。首先介绍的是 LeNet 网络架构，包括 LeNet 网络的设计、各个网络层以及激活函数等，并使用 LeNet 网络模型开发可用于图像分类的应用解决方案。

本章将覆盖如下主题：

- LeNet 架构及其变体；
- 设计 LeNet 架构；
- MNIST 数字分类；
- 德国交通标识分类。

本章有关代码和数据集已经上传到 GitHub 链接 https://github.com/Apress/computer-vision-using-deep-learning/tree/main/Chapter 3 中，建议使用 Jupyter Notebook 代码编辑器。对于本章内容，常用计算机的 CPU 就足以执行全部的代码。但是，如果需要的话，也可以使用 Google Colaboratory。如果读者不会设置 Google Colaboratory，可以参

考本书附录列出的参考信息。

3.1　深度学习的网络架构

讨论深度学习网络模型时，首先会想到的是网络模型的构件信息，例如神经元的数量是多少、网络层数是多少、使用什么样的激活函数、使用什么样的损失函数等。这些构件的参数取值对网络模型的设计和性能起着至关重要的作用。当提到神经网络模型深度时，指的就是网络模型中隐藏层的数量。随着计算能力的提高，网络层数变得越来越深，对计算能力的要求也越来越高。

提示：一般认为增加网络模型的层数会提高模型预测的准确性，但情况并非总是如此。这正是新型网络模型 ResNet 诞生的原因。

世界各地的研究人员和科学家花费了大量的时间和努力构建出多种不同的神经网络架构，其中最为流行的架构有：LeNet-5、AlexNet、VGGNet、GoogLeNet、ResNet、基于区域的 CNN(R-CNN)、YOLO(You Only Look Once)、SqueezeNet、SegNet、生成对抗网络(GAN)等。这些网络模型使用不同数量的隐藏层、神经元、激活函数、优化方法等。

LeNet 是一个容易理解的模型架构，也是深度学习架构的先驱之一，本章将详细讨论 LeNet 架构，并基于架构开发实际用例，考察使用不同超参数对模型预测性能的影响。

3.2　LeNet 架构

LeNet 是本书讨论的第一个网络架构，它是一种比较简单的 CNN 架构。LeNet 模型之所以具有非常重要的意义，就是因为在它被发明出来之前，字符识别是一个非常烦琐且耗时的过程。1998 年，LeNet 架构首先应用于对银行支票的手写数字进行分类，随后逐渐变得流行起来。

LeNet 架构有几种形式：LeNet-1、LeNet-4 和 LeNet-5，都是由 Yann LeCun 发明且经常被引用的模型架构。出于对空间信息的兴趣，这里将详细考察 LeNet-5 架构，读者可以使用相同的方法来考察和理解其余的模型架构。

3.2.1　LeNet-1 架构

LeNet-1 架构是第一个被概念化的 LeNet，很容易理解。如图 3-1 所示的示例，其网络层的维度如下所述。

(1) 第一层为 28×28 大小的输入图像。

(2) 第二层包含 4 个 24×24 大小的卷积层(大小为 5×5)。

(3) 第三层是平均池化层(大小为 2×2)。

(4) 第四层包含 8 个 12×12 的卷积层(大小为 5×5)。

(5) 第五层为平均池化层(大小为 2×2)。

(6) 最后是输出层。

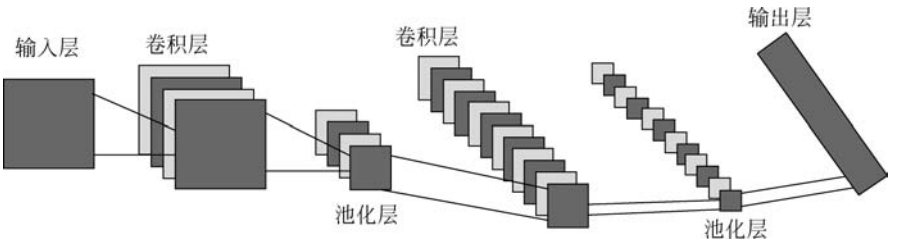

图 3-1 LeNet-1 架构

提示:当引入 LeNet 时,研究人员并没有提出最大池化;相反,他们使用的是平均池化。建议读者使用平均池化和最大池化两种方式进行测试。

LeNet-1 架构首先是输入层,然后是卷积层,接着是池化层,然后又是卷积层和池化层,最后是输出层。图像信息在整个网络模型中根据配置进行转换。在后续讨论 LeNet-5 架构时,会详细解释网络模型中所有层的功能和各自的输出。

3.2.2 LeNet-4 架构

LeNet-4 比 LeNet-1 略有改进,其架构如图 3-2 所示,引入了一个全连接层,可以提供更多的特征图。

(1) 第一层为 32×32 输入图像。

(2) 第二层包含 4 个 24×24 的卷积层(大小为 5×5)。

(3) 第三层为平均池化层(大小为 2×2)。

(4) 第四层包含 16 个 12×12 的卷积层(大小为 5×5)。

(5) 第五层为平均池化层(大小为 2×2)。

(6) 输出与 120 个神经元进行全连接,这些神经元又与 10 个作为最终输出的神经元进行全连接。

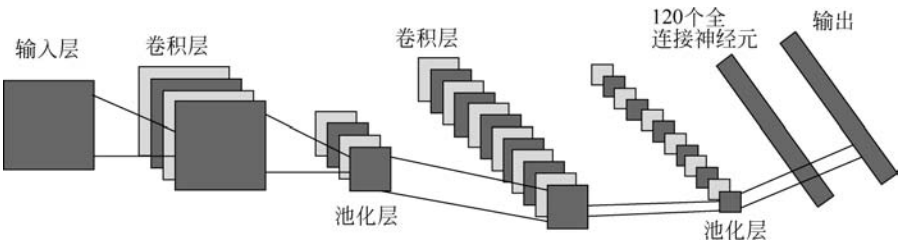

图 3-2 LeNet-4 是对 LeNet-1 架构的改进

3.2.3　LeNet-5 架构

在所有的 LeNet 架构中，引用最多的是 LeNet-5。它通常被用于解决具体的业务问题。

LeNet-5 架构如图 3-3 所示，最初在 LeCun 等的论文 *Gradient-Based Learning Applied to Document Recognition* 中被提出。

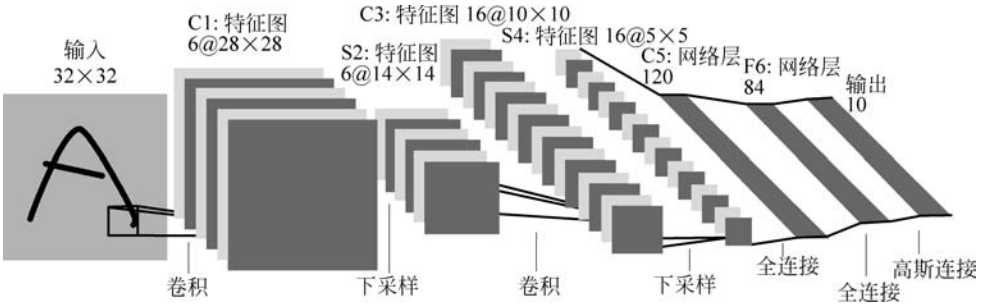

图 3-3　LeNet 架构(来源：http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf)

LeNet-5 是最常用的架构，各个网络层如下所述。

(1) LeNet-5 的第一层是一个 32×32 输入图像层。这个网络层接收的是一个灰度图像，通过一个包含 6 个 5×5 的过滤器的卷积块，将图像尺寸从 32×32×1 缩小为 28×28×6。这里的 1 表示通道数，因为输入是灰度图像，所以通道数是 1。如果接收的是 RGB 图像，那就有 3 个通道。

(2) 第二层为池化层，也称为下采样层，其过滤器大小为 2×2，步长为 2。图像尺寸缩小到 14×14×6。

(3) 第三层又是一个卷积层，包含 16 个特征图，大小为 5×5，步长为 1。注意，在这一层，16 个特征图中只有 10 个与上一层的 6 个特征图相连。这样处理具有明显的优势：计算成本更低，连接数量从 24 万个减少到 151600 个；这一层的训练参数总数是 1516，而不是 2400；打破了架构的对称性，因此网络模型的学习效果会更好。

(4) 第四层是一个池化层，过滤器大小为 2×2，步长为 2，输出为 5×5×16。

(5) 第五层是一个完连接卷积层，有 120 个特征图，每个大小为 1×1。每个神经元连接到上一层的 400 个节点。

(6) 第六层是一个有 84 个神经元的全连接层。

(7) 最后的输出层是一个 Softmax 层，每个数值为取某个数字字符的概率。

LeNet-5 架构的信息摘要如表 3-1 所示。

LeNet 是一个小巧且易于理解的网络架构。然而，它的功能足够强大、足够成熟，足以产生良好的预测结果，也易于执行。当然，建议使用不同的网络架构来测试解决方案，通过测试模型准确性选择最佳方案。

表 3-1 LeNet 架构的信息摘要

层	操作	特征图	输入尺寸	内核尺寸	步长	激活函数
输入		1	32×32			
1	卷积	6	28×28	5×2	1	tanh
2	平均池化	6	14×14	2×2	2	tanh
3	卷积	16	10×10	5×2	1	tanh
4	平均池化	16	5×5	2×2	2	tanh
5	卷积	120	1×1	5×2	1	tanh
6	完全连接	—	84			tanh
输出		—	10			Softmax

3.2.4 增强 LeNet-4 架构

增强是一种集成技术，在迭代中不断改进，逐步将弱学习器结合成强学习器。在增强 LeNet-4 中，增加了模型架构的输出功能，其中的最大输出值所对应的类别被确定为被预测的类别。增强 LeNet-4 架构如图 3-4 所示，通过结合多个弱学习器进行性能提升，使得模型最终的输出结果更具有准确性和鲁棒性。

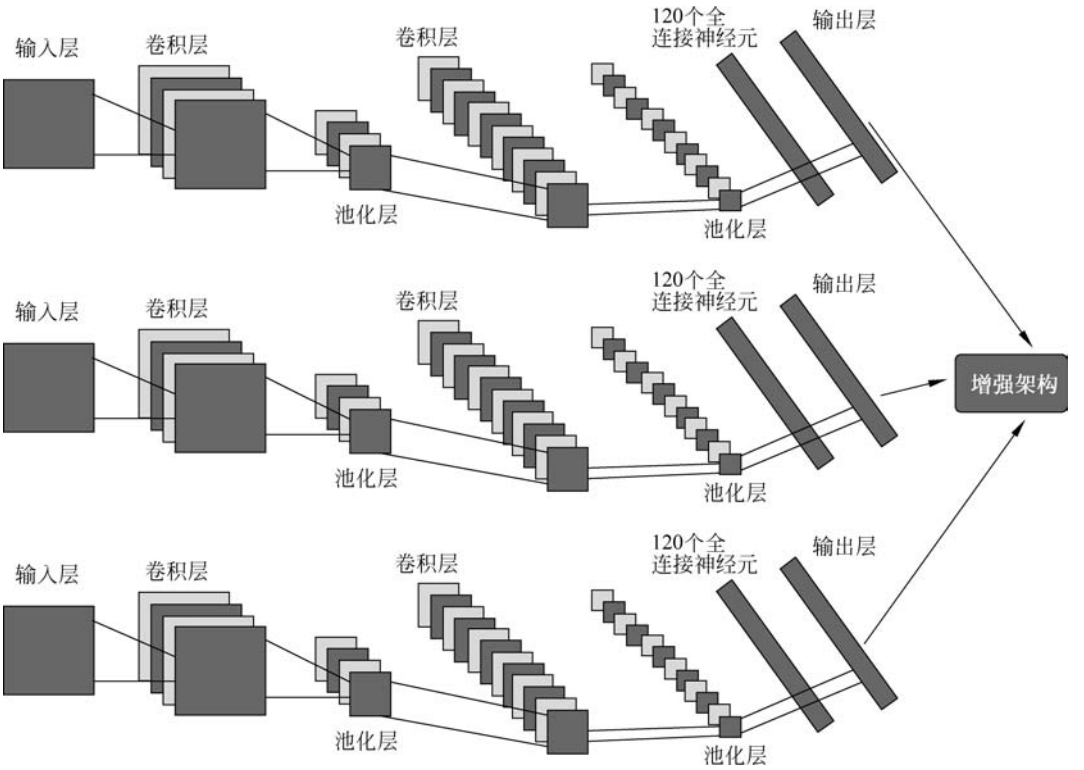

图 3-4 增强 LeNet-4 架构

3.3　使用 LeNet 创建图像分类模型

LeNet 是第一个流行起来并用于解决深度学习问题的网络架构。随着相关研究逐渐展开，已经开发出了很多更加先进的网络架构和算法，但 LeNet 在深度学习领域仍保留了特殊的位置。现在尝试创建第一个基于 LeNet 架构解决方案。

本节将使用 LeNet 架构开发两个应用。LeNet 是一个简单的架构，可以在 CPU 上实现对代码的编译。此处对这个架构做一些细微的调整，使用了最大池化激活函数而不是平均池化激活函数。

提示：与平均池化相比，最大池化可以提取一些重要特征。平均池化则会使图像变更加平滑，因此可能无法识别出比较尖锐的特征。

张量中通道的位置决定了应该如何重塑数据，所以在开始编写代码之前，应该关注一个小设置，在 3.3.1 节案例研究的第(8)步可以观察到这个设置。

每个图像可以用高度、宽度和通道数或者通道数、高度和宽度进行表示。如果通道数位于输入数组的第一个位置，那就使用 channels_first 条件进行重塑。这就意味着通道数位于张量(n 维数组)的第一位置。对于 channels_last 来说，通道数的位置则正好相反。

3.3.1　使用 LeNet 进行 MNIST 分类

这个应用是第 2 章中已经使用的 MNIST 数据集的延续。代码可以从本章开头给出的 GitHub 链接中获得。具体步骤如下所示。

(1) 导入所需的程序库。

```
import keras
from keras.optimizers import SGD
from sklearn.preprocessing import
LabelBinarizer from sklearn.model_selection
import train_test_split from sklearn.metrics
import classification_report from sklearn
import datasets
from keras import backend as K
import matplotlib.pyplot as plt
import numpy as np
```

(2) 导入数据集，然后从 Keras 导入一系列的网络层。

```
from keras.datasets import mnist # # Data set is
imported here
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
```

```
from keras.layers.convolutional import
MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras import backend as K
```

(3) 定义超参数。这一步类似于在第 2 章开发的 MNIST 和狗/猫分类。

```
image_rows, image_cols = 28, 28
batch_size = 256
num_classes = 10
epochs = 10
```

(4) 加载数据集。MNIST 是在库中默认添加的数据集。

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

(5) 将图像数据转换为浮点数,然后将其归一化。

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32') x_train /= 255
x_test /= 255
```

(6) 输出训练数据集和测试数据集的形状。

```
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

(7) 将变量转换为 one-hot 编码,其中使用了 Keras 的分类方法。

```
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

提示:使用 print 语句分析每个步骤的输出。如果需要的话,它允许在后面阶段对代码进行调试。

---

(8) 对数据进行相应的重塑。

```
if K.image_data_format() == 'channels_first':
x_train = x_train.reshape(x_train.shape[0], 1, image_rows, image_cols)
x_test = x_test.reshape(x_test.shape[0], 1, image_rows, image_cols)
input_shape = (1, image_rows, image_cols)
else:
x_train = x_train.reshape(x_train.shape[0],image_rows, image_cols, 1)
x_test = x_test.reshape(x_test.shape[0],image_rows, image_cols, 1)
input_shape = (image_rows, image_cols, 1)
```

(9) 创建网络模型,首先添加顺序层,然后添加卷积层和最大池化层。

```
model = Sequential()
model.add(Conv2D(20, (5, 5),
padding = "same",input_shape = input_shape))
model.add(Activation("relu")) model.
add(MaxPooling2D(pool_size = (2, 2),
strides = (2, 2)))
```

(10) 添加多个卷积层、最大池层、扁平化数据。

```
model.add(Conv2D(50, (5, 5), padding = "same"))
model.add(Activation("relu")) model.
add(MaxPooling2D(pool_size = (2, 2),
strides = (2, 2)))
model.add(Flatten()) model.add(Dense(500))
model.add(Activation("relu"))
```

(11) 添加一个密集层,然后是 Softmax 层(Softmax 主要用于多分类模型),并对模型进行编译。

```
model.add(Dense(num_classes)) model.
add(Activation("softmax"))
model.compile(loss = keras.losses.categorical_
crossentropy, optimizer = keras.optimizers.
Adadelta(), metrics = ['accuracy'])
```

(12) 创建好的模型已准备好接受训练,可以看到每个历元的计算对模型的预测准确度和损失函数取值的影响。可以尝试使用不同的超参数,比如历元数目和小批量的大小。根据超参数的不同,网络模型训练需要的时间也会有所不同。

```
theLeNetModel = model.fit(x_train, y_train,
batch_size = batch_size,
epochs = epochs,
verbose = 1, validation_data = (x_test, y_test))
```

如图 3-5 所示,可以分析损失函数值和模型预测的准确度如何随着每个历元的变化而变化。经过 10 个历元之后,模型在验证集上的准确率为 99.07%。对计算结果进行可视化展示。

(13) 绘制模型训练和模型测试准确度。从图 3-6 可以看出,随着时间的推移,模型在训练集和验证集上的准确度都在不断地增加。在经过第 7 和第 8 历元之后,准确度开始稳定。可以使用不同的超参数值进行模型测试。

```
import matplotlib.pyplot as plt
f, ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['acc'], 'o-')
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 36s 607us/step - loss: 0.2736 - acc: 0.9127 - val_loss: 0.1051 - val_a
cc: 0.9649
Epoch 2/10
60000/60000 [==============================] - 37s 622us/step - loss: 0.0590 - acc: 0.9813 - val_loss: 0.0490 - val_a
cc: 0.9835
Epoch 3/10
60000/60000 [==============================] - 37s 614us/step - loss: 0.0387 - acc: 0.9879 - val_loss: 0.0939 - val_a
cc: 0.9671
Epoch 4/10
60000/60000 [==============================] - 37s 625us/step - loss: 0.0285 - acc: 0.9910 - val_loss: 0.0267 - val_a
cc: 0.9905
Epoch 5/10
60000/60000 [==============================] - 37s 615us/step - loss: 0.0215 - acc: 0.9933 - val_loss: 0.0305 - val_a
cc: 0.9896
Epoch 6/10
60000/60000 [==============================] - 37s 614us/step - loss: 0.0164 - acc: 0.9949 - val_loss: 0.0228 - val_a
cc: 0.9920
Epoch 7/10
60000/60000 [==============================] - 37s 614us/step - loss: 0.0136 - acc: 0.9955 - val_loss: 0.0236 - val_a
cc: 0.9918
Epoch 8/10
60000/60000 [==============================] - 37s 616us/step - loss: 0.0106 - acc: 0.9969 - val_loss: 0.0279 - val_a
cc: 0.9909
Epoch 9/10
60000/60000 [==============================] - 37s 617us/step - loss: 0.0082 - acc: 0.9976 - val_loss: 0.0246 - val_a
cc: 0.9917
Epoch 10/10
60000/60000 [==============================] - 37s 620us/step - loss: 0.0062 - acc: 0.9983 - val_loss: 0.0316 - val_a
cc: 0.9907
```

图 3-5　分析损失函数值和模型预测的准确度

```
ax.plot([None] + theLeNetModel.history['val_acc'], 'x-')
ax.legend(['Train acc', 'Validation acc'], loc = 0)
ax.set_title('Training/Validation acc per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

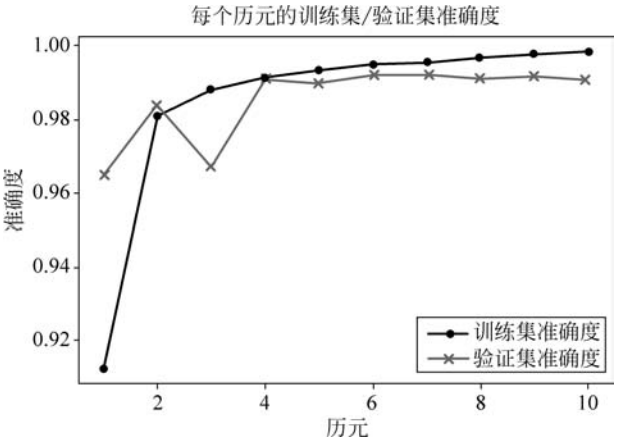

图 3-6　模型在训练集和验证集上的准确度

（14）分析损失函数值。如图 3-7 所示，随着每个历元的计算陆续完成，模型在训练集和验证集上的损失值都在持续降低。在经过第 7 和第 8 个历元的计算之后，模型的损失值趋于稳定。可以使用不同的超参数值进行测试。

```
import matplotlib.pyplot as plt f,
```

```
ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['loss'], 'o-')
ax.plot([None] + theLeNetModel.history['val_loss'], 'x-')
ax.legend(['Train loss', 'Validation loss'], loc = 0)
ax.set_title('Training/Validation loss per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

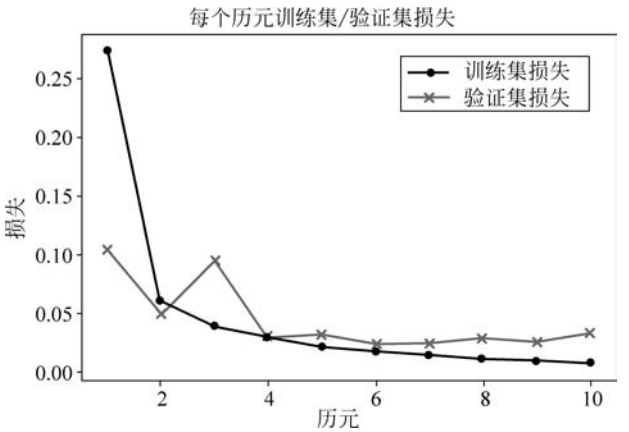

图 3-7　模型在训练集和验证集上损失值的变化

现在已经创建了一个有效进行图像分类的 LeNet 模型，在这个应用中，使用 LeNet-5 架构完成了一个图像分类模型的训练。

3.3.2　使用 LeNet 进行德国交通标志分类

第二个应用实例是自然识别德国交通标志，可以将这个实例应用于自动驾驶解决方案。这个应用实例使用 LeNet-5 架构建立一个深度学习模型。

（1）首先导入程序库。

```
import keras
from keras.optimizers import SGD
from sklearn.preprocessing import
LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn import datasets
from keras import backend as K
import matplotlib.pyplot as plt
import numpy as np
```

（2）导入 Keras 库以及绘图所需的所有包。

```
from keras.models import Sequential
```

```
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import
MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras import backend as K
```

(3) 导入常用代码库,如 NumPy、matplotlib、OpenCV 等。

```
import glob
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import random
import matplotlib.image as mpimg
import cv2
import os
from sklearn.model_selection import train_test_
split
from sklearn.metrics import confusion_matrix
from sklearn.utils import shuffle
import warnings
from skimage import exposure
# Load pickled data
import pickle
% matplotlib inline matplotlib.style.
use('ggplot')
% config InlineBackend.figure_format = 'retina'
```

(4) 以 pickle 文件格式提供数据集,并以 train.p 和 text.p 格式保存文件。该数据集可以从下面的 Kaggle 网站上下载:www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-trafficsign。

```
training_file = "train.p"
testing_file = "test.p"
```

(5) 打开文件并将数据保存在训练变量和测试变量中。

```
with open(training_file, mode = 'rb') as f: train = pickle.load(f)
with open(testing_file, mode = 'rb') as f: test = pickle.load(f)
```

(6) 将数据集划分为测试集和训练集。这里确定的测试集规模是 4000,建议读者尝试不同规模的测试集。

```
x,y = train['features'], train['labels']
x_train, x_valid, y_train, y_valid = train_
test_split(X, y, stratify = y,
```

```
test_size = 4000, random_state = 0)
x_test,y_test = test['features'],test['labels']
```

(7) 观察使用的图像样本,输出结果如图 3-8 所示。由于使用随机函数可以随机地选择图像数据,如果得到的输出结果与笔者的不一样,请不要担心。

```
figure, axiss = plt.subplots(2,5, figsize = (15, 4))
figure.subplots_adjust(hspace = .2, wspace = .001)
axiss = axiss.ravel()
for i in range(10):
index = random.randint(0, len(x_train))
image = x_train[index]
axiss[i].axis('off')
axiss[i].imshow(image)
axiss[i].set_title( y_train[index])
```

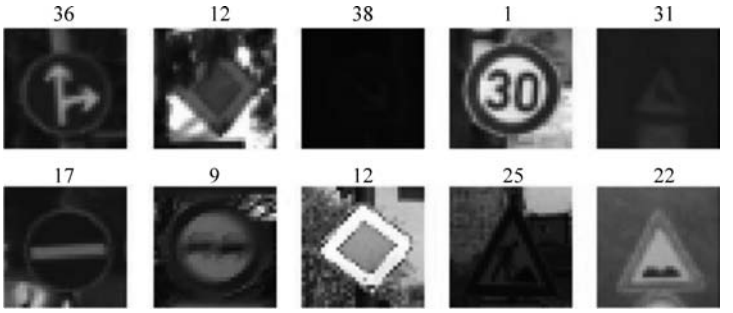

图 3-8　德国交通标志分类数据集的样本数据

(8) 选择模型训练的超参数。本示例有 43 个不同的类别,此处选择 10 个历元作为开始,读者也可以使用不同的历元数值检查模型性能。

```
image_rows, image_cols = 32, 32
batch_size = 256
num_classes = 43
epochs = 10
```

(9) 对数据进行探索性的分析,考察图像数据集,得到不同类别的分布频率直方图,图 3-9 给出了数据分布结果。不同类别的样本数量是不同的,有一些类别很好地呈现出来,而另外一些类别则不行。在理想情况下,应该为较少出现的类别收集更多数量的样本数据。在针对实际问题的解决方案中,希望有一个样本出现频率分布比较平衡的一种数据集,第 8 章将对此进行更详细的讨论。

```
histogram, the_bins = np.histogram(y_train,bins = num_classes)
the_width = 0.7 * (the_bins[1] - the_bins[0])
center = (the_bins[:-1] + the_bins[1:]) / 2
plt.bar(center, histogram, align = 'center',width = the_width) plt.show()
```

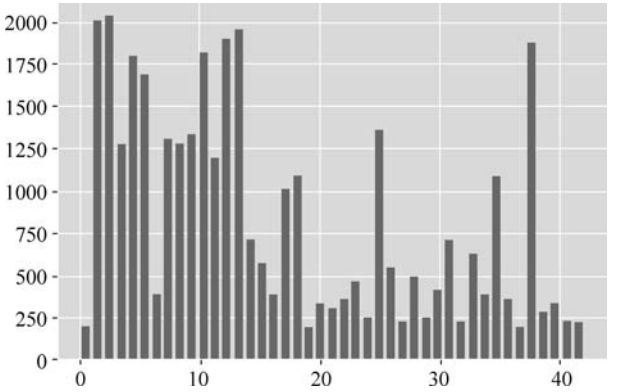

图 3-9　各个类别的样本出现频率分布

(10) 样本的数量分布如何跨越不同的类别。它是 NumPy 库中的一个正则直方图函数。

```
train_hist, train_bins = np.histogram(y_train,
bins = num_classes)
test_hist, test_bins = np.histogram(y_test,
bins = num_classes)
train_width = 0.7 * (train_bins[1] - train_ bins[0])
train_center = (train_bins[:-1] + train_bins[1:]) / 2
test_width = 0.7 * (test_ bins[1] - test_bins[0])
test_center = (test_ bins[:-1] + test_bins[1:]) / 2
```

(11) 画出直方图,训练集中的样本数量显示为红色,测试集中的样本数量显示为绿色,输出结果如图 3-10 所示。

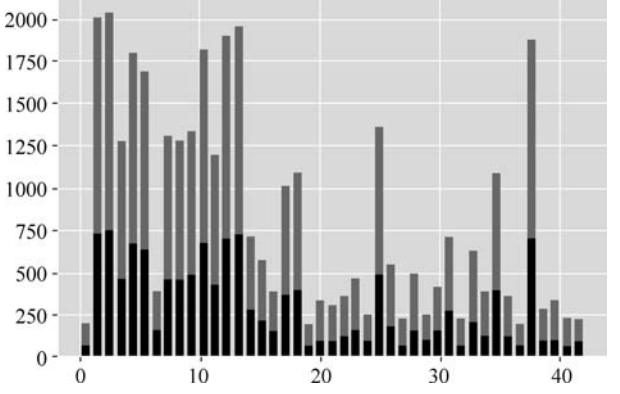

图 3-10　训练集和测试集中不同类别样本的出现频率分布

```
plt.bar(train_center, train_hist,
align = 'center', color = 'red', width = train_width)
```

```
plt.bar(test_center, test_hist, align = 'center',
color = 'green', width = test_width)
plt.show()
```

(12) 分析样本数据的类别分布,查看直方图中样本在训练集和测试集上占比的差异。将图像数据转换为浮动数据,然后做归一化处理。

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train / = 255
x_test / = 255
print('x_train shape:', x_train.shape)
print(x_ train.shape[0], 'train samples')
print(x_test. shape[0], 'test samples')
```

基于 35209 个训练样本数据点和 12630 个测试样本数据点,将类别向量转换为基于二进制的类别矩阵。

```
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

以下代码与之前开发的 MNIST 图像分类模型中给出的代码相同。channels_first 表示通道数位于数组中的第一个位置,根据 channels_first 的位置改变 input_shape。

```
if K.image_data_format() == 'channels_first':
input_shape = (1, image_rows, image_cols)
else:
input_shape = (image_rows, image_cols, 1)
```

(13) 先添加顺序层和卷积层,再添加池化层和卷积层。

```
model = Sequential() model.add(Conv2D(16,(3,3),
input_shape = (32,32,3)))
model.add(Activation("relu")) model.
add(MaxPooling2D(pool_size = (2, 2),
strides = (2, 2)))
model.add(Conv2D(50, (5, 5), padding = "same"))
model.add(Activation("relu")) model.
add(MaxPooling2D(pool_size = (2, 2),
strides = (2, 2)))
model.add(Flatten()) model.add(Dense(500))
model.add(Activation("relu"))
model.add(Dense(num_classes)) model.
add(Activation("softmax"))
```

(14) 输出关于模型的摘要信息,输出结果如图 3-11 所示。

```
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_15 (Conv2D)           (None, 30, 30, 16)        448
_________________________________________________________________
activation_13 (Activation)   (None, 30, 30, 16)        0
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 15, 15, 16)        0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 15, 15, 50)        20050
_________________________________________________________________
activation_14 (Activation)   (None, 15, 15, 50)        0
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 7, 7, 50)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 2450)              0
_________________________________________________________________
dense_7 (Dense)              (None, 500)               1225500
_________________________________________________________________
activation_15 (Activation)   (None, 500)               0
_________________________________________________________________
dense_8 (Dense)              (None, 43)                21543
_________________________________________________________________
activation_16 (Activation)   (None, 43)                0
=================================================================
Total params: 1,267,541
Trainable params: 1,267,541
Non-trainable params: 0
```

图 3-11　模型摘要信息

（15）对模型进行编译，训练模型。每个历元的准确度和损失值的变化如图 3-12 所示，经过 10 个历元后，模型在验证集上的准确率为 91.16%。

```
model.compile(loss = keras.losses.categorical_ossentropy, optimizer = keras.optimizers.
Adadelta(), metrics = ['accuracy'])
theLeNetModel = model.fit(x_train, y_train,
batch_size = batch_size,
epochs = epochs,
verbose = 1,
validation_data = (x_test, y_test))
```

```
WARNING:tensorflow:From /Users/vaibhavverdhan/anaconda3/lib/python3.6/site-packages/tensorflow/python/ops/math_ops.p
y:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 35209 samples, validate on 12630 samples
Epoch 1/10
35209/35209 [==============================] - 22s 632us/step - loss: 2.2025 - acc: 0.3971 - val_loss: 1.4474 - val_a
cc: 0.5604
Epoch 2/10
35209/35209 [==============================] - 22s 631us/step - loss: 0.5801 - acc: 0.8291 - val_loss: 0.6247 - val_a
cc: 0.8179
Epoch 3/10
35209/35209 [==============================] - 22s 629us/step - loss: 0.1937 - acc: 0.9501 - val_loss: 0.4696 - val_a
cc: 0.8833
Epoch 4/10
35209/35209 [==============================] - 22s 623us/step - loss: 0.0956 - acc: 0.9770 - val_loss: 0.4750 - val_a
cc: 0.8850
Epoch 5/10
35209/35209 [==============================] - 23s 651us/step - loss: 0.0564 - acc: 0.9876 - val_loss: 0.5317 - val_a
cc: 0.8864
Epoch 6/10
35209/35209 [==============================] - 23s 642us/step - loss: 0.0364 - acc: 0.9925 - val_loss: 0.4336 - val_a
cc: 0.9140
Epoch 7/10
35209/35209 [==============================] - 22s 630us/step - loss: 0.0251 - acc: 0.9953 - val_loss: 0.4621 - val_a
cc: 0.9138
Epoch 8/10
35209/35209 [==============================] - 22s 631us/step - loss: 0.0186 - acc: 0.9964 - val_loss: 0.4819 - val_a
cc: 0.9117
Epoch 9/10
35209/35209 [==============================] - 22s 628us/step - loss: 0.0121 - acc: 0.9981 - val_loss: 0.5061 - val_a
cc: 0.9124
Epoch 10/10
35209/35209 [==============================] - 22s 619us/step - loss: 0.0112 - acc: 0.9983 - val_loss: 0.5421 - val_a
cc: 0.9116
```

图 3-12　每个历元的准确度和损失值的变化

（16）接下来对计算结果进行可视化展示，首先绘制网络模型在训练集和测试集上的准确度，如图 3-13 所示，在经过第 5 和第 6 个历元后，准确度的取值趋于稳定。

```
import matplotlib.pyplot as plt
f, ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['acc'], 'o-')
ax.plot([None] + theLeNetModel. history['val_acc'], 'x-')
ax.legend(['Train acc', 'Validation acc'], loc = 0)
ax.set_ title('Training/Validation acc per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

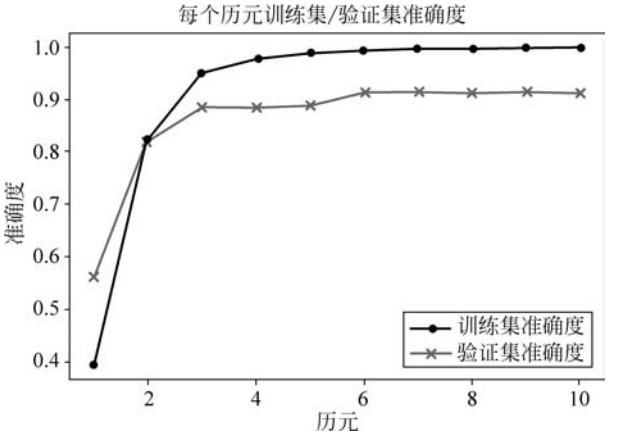

图 3-13　模型在训练集和验证集上的准确度

（17）绘制模型在训练集和验证集上的损失函数值的变化。如图 3-14 所示，在第 5 和第 6 历元之后，损失函数值的变化趋于稳定，只有很微小的降低。可以生成关于模型准确度和模型损失函数值的变化曲线图来衡量模型的性能。这些曲线图与 MNIST 图像分类模型中给出的曲线图比较类似。

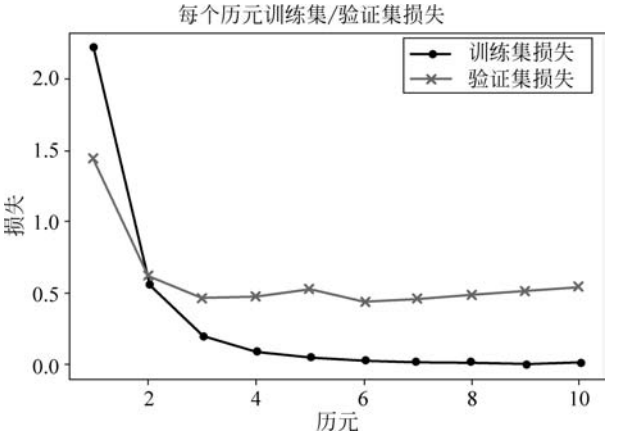

图 3-14　模型在训练集和验证集上的损失函数值

```
import matplotlib.pyplot as plt
f, ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['loss'], 'o-')
ax.plot([None] + theLeNetModel. history['val_loss'], 'x-')
ax.legend(['Train loss', 'Validation loss'], loc = 0)
ax.set_ title('Training/Validation loss per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

提示：模型的所有性能参数都保存在 theLeNetModel 或者 theLeNetModel. model. metrics 的内部。

在本例的模型开发过程中，还增加了一个额外的步骤，就是为模型的预测结果创建一个混淆矩阵。为此，必须首先使用模型对测试集样本数据做出预测，然后将模型预测结果与图像的实际标签进行比较。

（18）使用预测函数进行预测。

```
predictions = theLeNetModel.model.predict(x_test)
```

（19）创建混淆矩阵，可以在 scikit-learn 库中找到相关的代码。

```
from sklearn.metrics import confusion_matrix
import numpy as np
confusion = confusion_matrix(y_test,np.argmax(predictions,axis = 1))
```

（20）创建名为 cm 的变量表示混淆矩阵。

```
cm = confusion_matrix(y_test,np.argmax(predictions,axis = 1))
```

（21）对混淆矩阵进行可视化表示。seaborn 是用于可视化开发的代码库，可以与 matplotlib 一起使用。代码输出结果如图 3-15 所示，由于维度的数量问题，但是由于维度的数量问题，输出的混淆矩阵不是很清晰，读者可以进行改进。

```
import seaborn as sn
df_cm = pd.DataFrame(cm, columns = np.unique(y_test), index = np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
sn.set(font_scale = 1.4) # for label size
sn.heatmap(df_cm, cmap = "Blues",
annot = True,annot_kws = {"size": 16}) # font size
```

（22）再次绘制混淆矩阵。需要注意的是，此处定义了一个函数 plot_ confusion_ matrix，这个函数将混淆矩阵作为输入参数。然后，使用常规的 matplotlib 库及其函数来绘制混淆矩阵。图 3-16 给出了混淆矩阵的图示，模型对有些类别有很好的分类效果。对模型

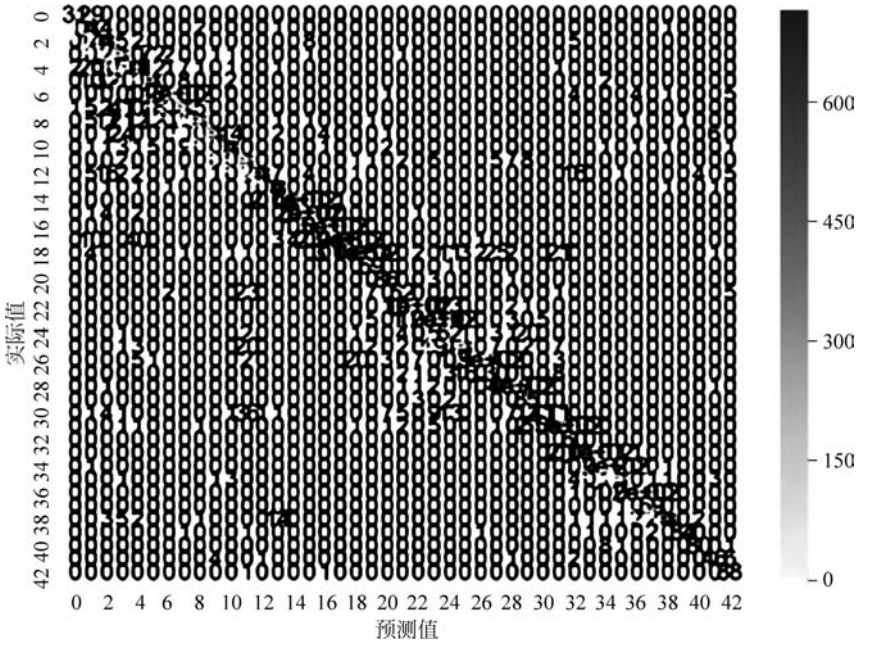

图 3-15　混淆矩阵的可视化表示

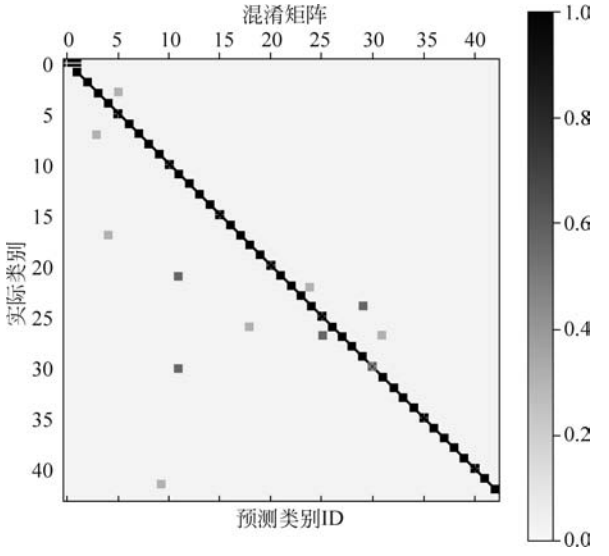

图 3-16　为所有类别生成的混淆矩阵

分类的效果进行分析，并对模型超参数迭代调整，查看超参数的取值对模型预测效果的影响。认真分析模型进行错误分类的类别，找出模型错分的原因。例如，对于手写数字分类的情形，算法可能会在数字 1 和 7 之间产生混淆。因此，一旦对模型进行测试，就应该努力寻

找模型容易产生错误分类的类别并分析模型产生错分的原因，例如从训练数据集中去除令人困惑的图像。改善图像质量和对容易错分的类别进行进一步的细化分类都有助于解决这个问题。

```
def plot_confusion_matrix(cm):
cm = [row/sum(row) for row in cm]
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(111)
cax = ax.matshow(cm, cmap = plt.cm.Oranges) fig.colorbar(cax)
plt.title('Confusion Matrix') plt.
xlabel('Predicted Class IDs') plt.ylabel('True Class IDs')
plt.show()
plot_confusion_matrix(cm)
```

3.4 小结

神经网络架构对于计算机视觉问题来说是一种非常有趣和强大的解决方案。基于这些架构，可以在一个非常大的数据集上进行训练，并且对图像的正确识别很有帮助。这个功能可用于跨域的各种各样的问题。但是这种解决方案的质量在很大程度上取决于训练数据集的质量。

前两章介绍了卷积、最大池化、填充等概念，并开发了基于 CNN 模型的解决方案。本章开始介绍自定义神经网络架构。这些架构在诸如层数、激活函数、跨层连接、卷积核大小等模型设计上各有特点。一般建议测试 4 种不同架构中的 3 种架构来比较模型的准确性。

本章讨论了 LeNet 架构并重点讨论了 LeNet-5。开发实现了两个端到端实际应用案例，完成了从数据加载到网络设计和模型准确性测试的整个应用案例开发流程。

第 4 章将讨论另外一种比较流行的架构——VGGNet。

习题

（1）不同版本的 LeNet 有什么不同？

（2）如何使用不同的历元数来度量模型准确度的变化？

（3）本章讨论了两个实际应用案例。使用不同的超参数值迭代计算相同的解决方案，为第 2 章中完成的应用案例创建模型损失函数和准确性变化曲线。

（4）利用第 2 章给出的数据集，使用 LeNet-5 架构进行测试，比较模型预测效果。

（5）从 www.kaggle.com/puneet6060/intel-imageclassification/version/2 下载图像场景分类数据集，对此数据集执行用于德国交通图像分类的代码。

（6）从 http://chaladze.com/l5/下载林奈 5 数据集，并将该数据集分为浆果、鸟、狗、花和其他等 5 个类别。使用此数据集创建一个基于 CNN 的分类解决方案。

拓展阅读

[1] Xu X，Dehghani A，Corrigan D，et al. Convolutional Neural Network for 3D object recognition using volumetric representation[C]//The First International Workshop on Sensing，Processing and Learning for Intelligent Machines. IEEE，2016.

[2] Sarraf S，Tofighi G. Classification of Alzheimer's Disease using fMRI data and deep learning convolutional neural networks[EB/OL]. https://arxiv.org/abs/1603.08631.

[3] Lecun Y，Bottou L，Bingo Y，et al. Gradient-based learning applied to document recognition[EB/OL]. http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.