第5章

视频讲解

分类识别技术与应用

在当今信息化社会中，分类识别技术作为人工智能领域的重要组成部分，其应用场景日益广泛，从自动驾驶中的障碍物识别到医疗影像诊断，再到日常生活中的人脸解锁，无一不展现出其实用价值。随着车辆数量的急剧增加，车牌识别技术成为智能交通系统中的关键技术之一，广泛应用于自动收费、车辆追踪、安全监控等领域。本章聚焦于循环卷积神经网络在车牌识别中的应用，结合卷积神经网络对图像的出色处理能力和循环神经网络对序列数据的处理优势，实现高精度的车牌识别。

5.1 应用背景

随着交通基础设施的完善和汽车产业的快速发展，国内汽车保有量持续上升，占全球比例不断增加。然而，机动车数量的迅速增长给城市道路交通带来了巨大压力，导致交通拥堵和事故频发，交通事故数量逐年增加。2018—2022年内我国的交通事故数量和类别统计如图5-1和图5-2所示。

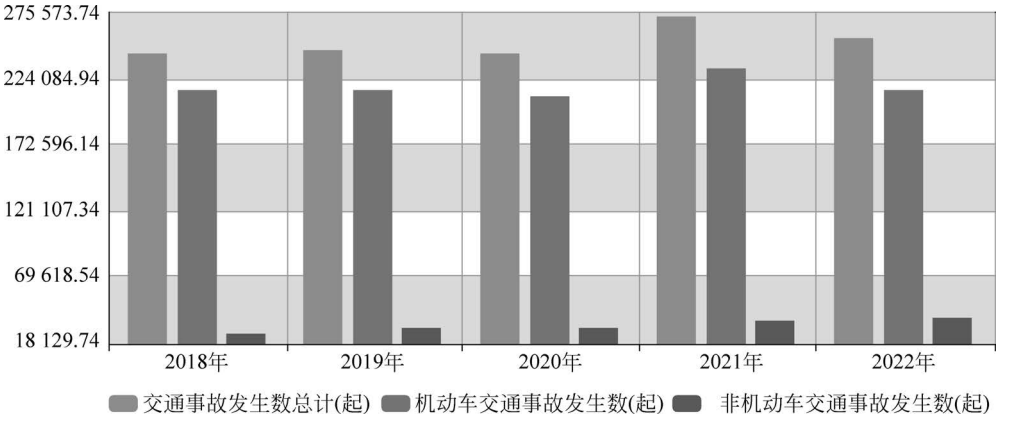

图5-1 2018—2022年内交通事故数量统计

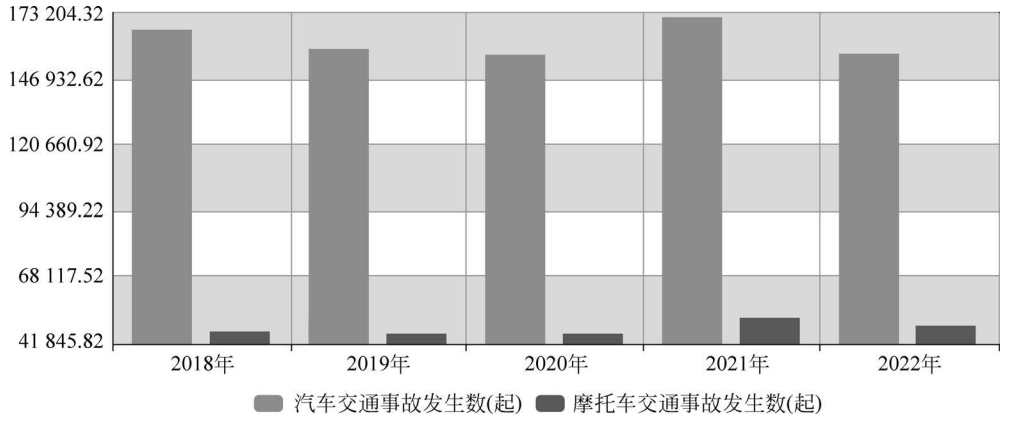

图 5-2　2018—2022 年内交通事故类别统计

要真正降低交通事故的发生率，必须着重管理机动车辆。除了对非机动车的管理外，机动车的管理措施也需进一步加强。国家统计局数据显示，2021 年汽车事故达到 17 万起。事故数量居高不下的一个主要原因是驾驶员在驾驶过程中存在如刹车不及时、疲劳驾驶、酒驾等违法行为。随着对机动车的限行措施日益严格，一些驾驶员开始采用涉牌违法手段逃避处罚，如伪造、套用其他车辆的号牌等。这种行为不仅给社会造成严重损害，也对道路安全管理带来不利影响。

涉牌违法，是对与机动车号牌有关的交通违法行为的统称，其违法行为主要包括：伪造或者使用伪造号牌；变造或者使用变造号牌；使用其他车辆号牌；故意遮挡、污损号牌；不悬挂号牌；不按规定安装号牌；号牌不清晰、不完整等。由于涉牌违法行为人绝大多数为主观故意，所以其危害后果也远远大于一般的交通违法行为。这些行为不仅侵犯了他人的合法权益，也对道路交通安全构成潜在威胁。因此，对涉牌违法行为的识别对于优化道路交通管理、配合交管部门规划道路等具有重要意义。

在当今快速发展的信息化社会中，车牌号识别系统作为一项前沿科技，承载着多重社会价值与使命，其研究意义和功能需求分别如下。

1. 研究意义

通过 CRNN 技术的应用，车牌号识别系统可实现车牌识别的高度自动化与精准化。这一技术革新不仅可以提高交通违法监管的效率、降低人力成本，还促使交通监管体系向着更为智能化、高效化的方向演进，为构建智慧型城市奠定坚实的基础。

在复杂多变的交通环境下，车牌号识别系统如同一道无形的屏障，能够迅速、准确地捕捉潜在的违规违法行为，有效预防交通事故的发生，为公众出行提供了一个更加安全、有序的交通环境。它通过实时监测与预警，增强交通系统的整体安全水平，降低因人为疏忽或违法行为导致的事故风险。

在维护社会治安与打击犯罪活动中，车牌号识别技术发挥着不可或缺的作用。它能够为执法机构提供及时有效的线索，加速案件侦破进程，为维护社会稳定与安全提供了有力的技术支撑。

2. 功能需求

车牌号识别系统的核心竞争力在于利用 CRNN 技术实现车牌的高效准确识别，无论

是常规车牌还是特殊类型，均需达到良好的识别效果。尤其在复杂多变的环境条件下，如光照变化、角度偏斜或部分遮挡，系统必须具有适应能力，确保在任何场景下都能维持高识别率，为交通管理提供稳定可靠的支持。

在瞬息万变的交通环境中，系统需具备高可靠的实时响应能力，确保在极短时间内完成车牌识别，为交通指挥中心提供即时信息，以便快速响应违规事件，有效控制交通流量，优化道路通行效率。

5.2　卷积神经网络的设计与构建

5.2.1　CRNN 神经网络架构

本章的车辆号牌识别系统核心算法部分，采用的是由 CNN＋LSTM＋CTC 算法组合而成的网络，整个网络可以分为 3 部分。

（1）卷积层，从输入图像中提取一个特征序列。

（2）循环层，预测每一帧的标签分布。

（3）转录层，将每一帧的预测转换为最终的标签序列。

典型的 CRNN 架构如图 5-3 所示。

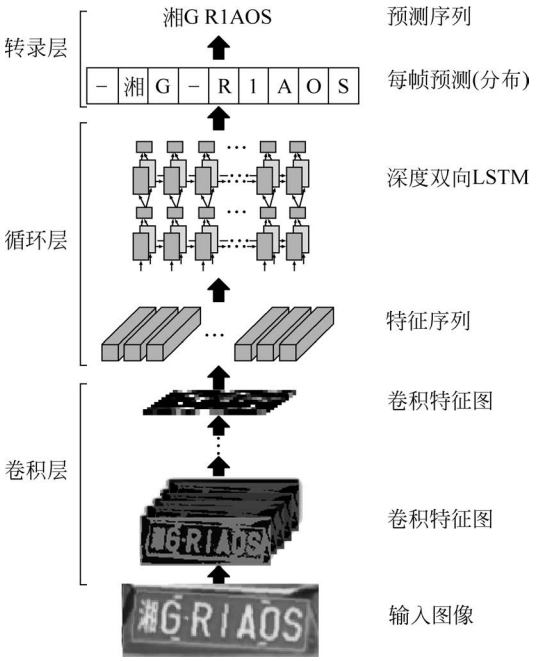

图 5-3　典型的 CRNN 架构

通过图 5-3 可以看出，CRNN 由卷积层、循环层和转录层 3 部分组成。在 CRNN 的底部，卷积层自动从每个输入图像中提取一个特征序列。在卷积网络的基础上，建立一个递归网络，由卷积层输出，对特征序列的每一帧进行预测。CRNN 顶部的转录层，将循环层的每帧预测转换为标签序列。虽然 CRNN 是由不同类型的网络架构组成的（CNN

和 RNN),但它可以用一个损失函数进行联合训练。

5.2.2 卷积核的作用与选择

卷积核(也称作滤波器或特征检测器)是一种小型矩阵,通常在空间上较小(如 3×3,5×5 等),用于扫描输入图像的每一个位置,与局部像素进行加权求和运算。卷积核的权重(每个元素的值)是通过训练过程学习得到的,用于检测图像中的特定特征,如边缘、纹理或模式。

在卷积核神经网络的基本设计中,一个核心思想是随着网络的层数增加,网络需要逐步"变宽",即增加通道数。这一演进的设计理念旨在使网络能够更有效地学习和表达不同层次的特征,从基础信息到更抽象的概念。

在网络的初期阶段,例如前三层,每个通道实际上专注于学习任务中的特定基础特征。这可能涉及边缘信息、几何信息和颜色信息等基础视觉元素。这些基础信息在许多视觉任务中都是相似的,因此相对较少的通道(如 64 个)已经足以充分表达这些基础特征。

然而,随着网络的深度发展,网络开始学习越来越抽象的特征,并且这些特征更加针对具体的网络设计任务。这些抽象的特征实际上是前面层次中不那么抽象特征的组合。以第 n 层的第一个通道为例,其特征可以看作是第 $n-1$ 层所有通道对应位置特征的加权和。

随着网络深度的增加,可以组合的可能性变得越来越多,也就是说,对每种关键属性的描述变得更加具体。为了使网络具有更强的表达能力,需要增加通道数,以覆盖尽可能多的关键特征。这样,神经网络就能够逐渐从基础信息中提炼出更为复杂和抽象的知识,使其在特定任务上表现更出色。这一设计原则在神经网络的演进中发挥了关键作用,为其在各种复杂任务中取得成功奠定了基础。

卷积核的通道数等于输入的通道数,卷积核的个数等于输出的通道数。其中,典型的卷积核的输入与输出如图 5-4 所示。

5.2.3 特征图提取与表示

在深度学习框架下,特征图的提取与表示是卷积神经网络(CNN)的核心机制之一,是模型理解和解析输入数据的关键。这一过程始于卷积操作,卷积层利用卷积核在输入数据上滑动,通过局部区域的处理来捕捉并提炼出数据中的模式与特征。卷积操作凭借其局部连接和参数共享的特性,不仅降低了计算成本,还使得网络能逐步构建从低级到高级的抽象表示,加深对数据本质的理解。为了增强网络的非线性表达能力,卷积操作之后通常接续非线性激活函数(如 ReLU),这一步骤使得模型能够捕捉更为复杂和多层次的特征,提升其对多样数据结构的拟合能力。

进一步地,池化技术如最大池化或平均池化可以减少特征图的维度和数量,实现下采样的同时保留核心特征信息,既减轻了计算负担,又促进了模型的泛化能力。多尺度特征图的策略通过采用多种大小的卷积核和池化层组合,使网络能够有效识别和处理不同尺度下的物体或场景,可增强模型的感知灵活性。

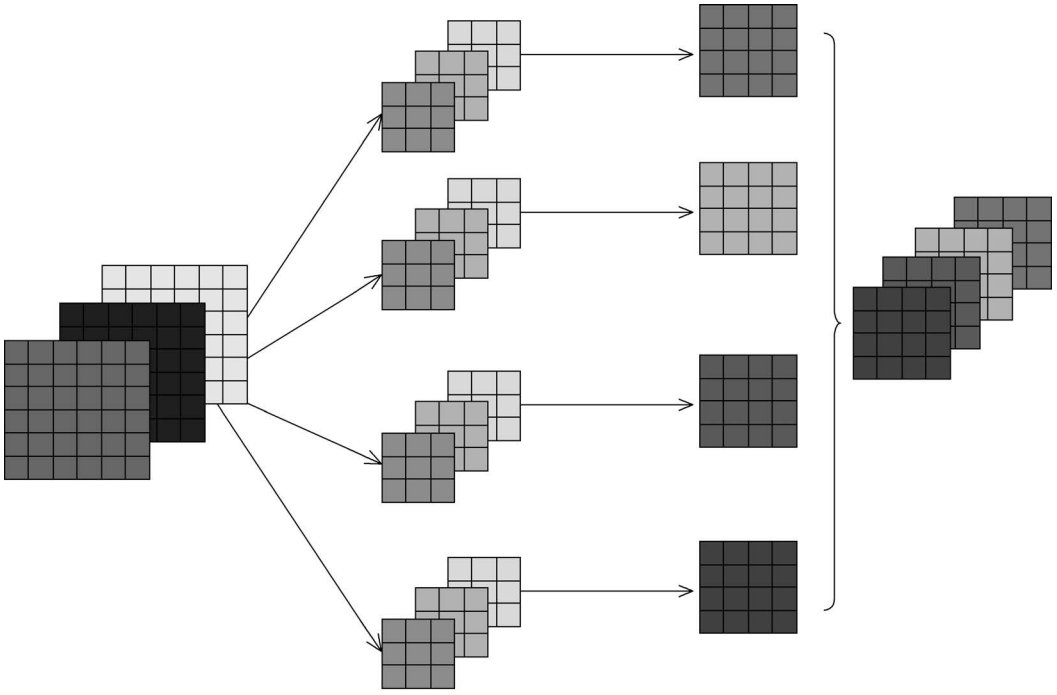

图 5-4　典型的卷积核的输入与输出

为了直观体会特征图的作用，特征图的可视化成为理解神经网络内部运作的重要工具，可帮助使用者洞察每一层所学到的特征类型，为模型的优化和调试提供直观依据。

举个例子，输入图像维度为 [1,3,224,224]，如图 5-5 所示。image[1,3,224,224]表示一个四维的张量(Tensor)，通常在深度学习中用来表示图像数据。这个张量的维度是 [batch_size, channels,height,width]。

图 5-5　输入图像维度为 [1,3,224,224]

batch_size：表示一个批次(Batch)中包含的图像数量，这里是 1，即一个单独的图像。

channels：表示图像的通道数，这里是 3，通常对应 RGB(红、绿、蓝)颜色通道。

height：表示图像的高度，这里是 224 像素。

width：表示图像的宽度，这里也是 224 像素。

因此，image[1,3,224,224] 表示一个包含一个 RGB 彩色图像的批次，该图像的尺寸为 224 像素×224 像素。

现在将这张图片输入卷积 conv [1,64,112,112]中，参考特征图可视化示例，如代码

5-1 所示，得到一个特征张量 output_tensor，将其可视化可得 conv [1，64，112，112]后的特征图可视化输出，如图 5-6 所示。

【代码 5-1】 特征图可视化示例。

```
import torch
import torch.nn as nn
#输入尺寸[1, 3, 224, 224]
input_tensor = torch.randn(1, 3, 224, 224)
#定义卷积层
conv_layer = nn.Conv2d(in_channels = 3,out_channels = 64,kernel_size = 3,stride = 1,
padding = 1)
#计算输出
output_tensor = conv_layer(input_tensor)
```

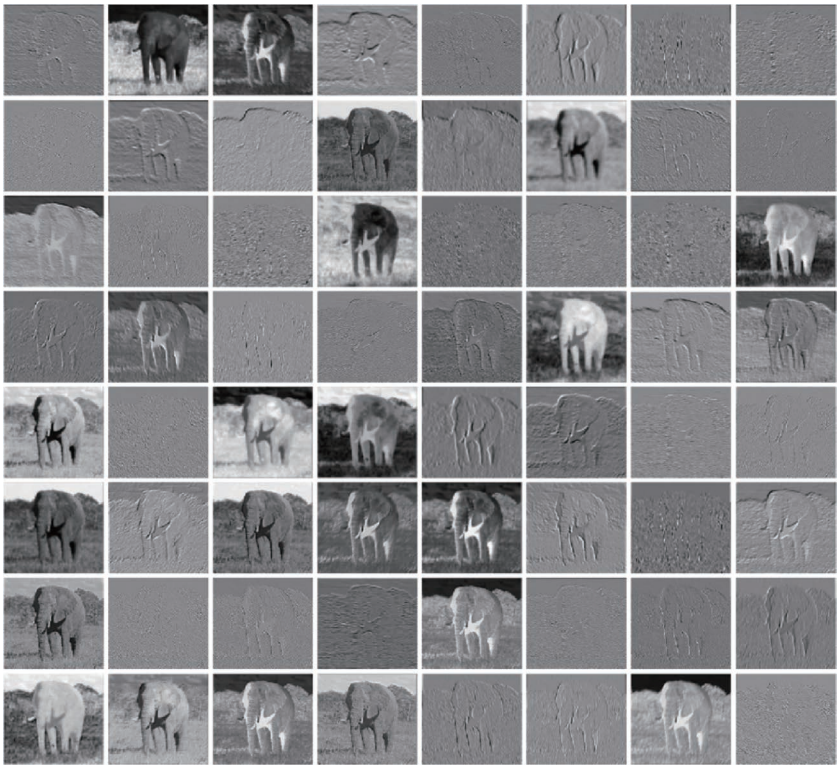

图 5-6 conv [1，64，112，112]后的特征图可视化输出

同理，经过 conv [1，256，56，56]，可以对特征图进行可视化，观察特征的提取情况，如图 5-7 所示。

再经过 conv [1，512，28，28]可得到 conv [1，512，28，28]特征图，如图 5-8 所示。

CRNN 的特征图提取与表示：

对于车牌识别，采用 CRNN 架构，对其中的一层卷积的随机 100 个通道进行特征图提取并可视化，参考 CRNN 的特征图提取与表示代码，如代码 5-2 所示。其中，特征提取的原图如图 5-9 所示。

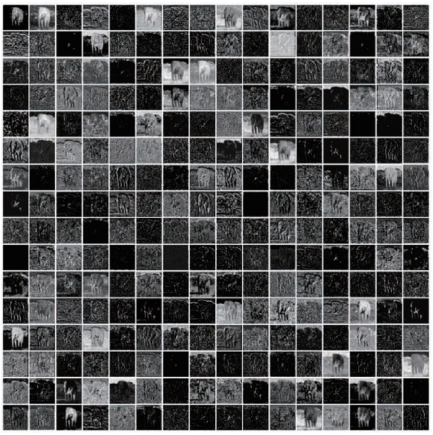

图 5-7　conv [1,256,56,56]特征图

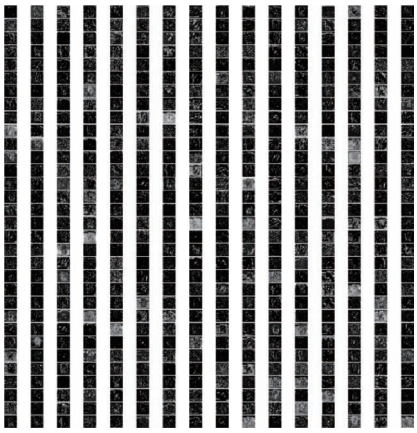

图 5-8　conv [1,512,28,28]特征图

图 5-9　特征提取的原图

【代码 5-2】 CRNN 的特征图提取与表示。

```
import os
import random
import torch
import torchvision
from matplotlib import pyplot as plt
from torch import nn
from torch.autograd import Variable
import utils
import dataset
from PIL import Image
import models.crnn as crnn

#模型路径和图像路径
model_path = './trained_model/netCRNN_24_10100.pth'
img_path = './data/000000000.jpg'

#初始化 CRNN 模型
model = crnn.CRNN(32, 1, 77, 256)
#将模型移动到 GPU
if torch.cuda.is_available():
    model = model.cuda()
print('loading pretrained model from %s' % model_path)

#加载预训练模型
model = nn.DataParallel(model)
model.load_state_dict(torch.load(model_path))

```

```
28
29 #创建模型转换器和图像变换器
30 converter = utils.strLabelConverter(alphabet)
31 transformer = dataset.resizeNormalize((100, 32))
32
33 #读取图像并进行预处理
34 image = Image.open(img_path).convert('L')
35 image = transformer(image)
36
37 #移动图像到 GPU(如果可用)
38 if torch.cuda.is_available():
39     image = image.cuda()
40
41 #调整图像维度
42 image = image.view(1, *image.size())
43 image = Variable(image)
44
45 #设置模型为评估模式
46 model.eval()
47
48 #对图像进行预测
49 preds = model(image)
50
51
52 #_____可视化_____开始
53 #若程序中引入多个 OpenMP 运行时库,可能由于混合使用了不同的库或静态链接导致的。这
#可能会降低性能或导致不正确的结果。设置以下环境变量可解决
54 os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'
55 #从 DataParallel 中获取底层模型
56 underlying_model = model.module
57
58 #打印底层模型的命名子模块
59 for name, child in underlying_model.named_children():
60     pass
61     print(name, child)
62
63 #创建 CNN 特征提取器
64 cnn_feature_extractor = torchvision.models._utils.IntermediateLayerGetter(underlying_
model.cnn, {"conv1":underlying_model.cnn.conv0,
65
                "conv2":underlying_model.cnn.conv3,
66
    "pooling3":underlying_model.cnn.pooling3})
67
68 #out = cnn_feature_extractor(image)
69  # rnn _ feature _ extractor = torchvision. models. _ utils. IntermediateLayerGetter
(underlying_model.rnn[0], {'rnn':0,'embedding':1})
70
71 #提取特征图
72 out = cnn_feature_extractor(image)
73 tensor_ls = [(k,v) for k,v in out.items()]
```

```
#选取 conv2 的输出
v = tensor_ls[1][1]
#取消 Tensor 的梯度并转成三维张量,否则无法绘图
v = v.data.squeeze(0).cpu()
print(v.shape)

#定义函数,随机从 0～end 的一个序列中抽取 size 个不同的数
def random_num(size, end):
    range_ls = [i for i in range(end)]
    num_ls = []
    for i in range(size):
        num = random.choice(range_ls)
        range_ls.remove(num)
        num_ls.append(num)
    return num_ls

#随机选取 100 个通道的特征图
channel_num = random_num(100, v.shape[0])
plt.figure(figsize = (20, 10))
for index, channel in enumerate(channel_num):
    ax = plt.subplot(10, 10, index + 1, )
    plt.imshow(v[channel, :, :])        #灰度图参数 cmap = "gray"
    #plt.imshow(v[channel, :, :], cmap = "gray")       #加上 cmap 参数就可以显示灰
#度图
plt.savefig("feature.jpg", dpi = 600)
plt.show()
#_____________________可视化_____________________结束#

    #进行文本解码
    _, preds = preds.max(2)
    preds = preds.transpose(1, 0).contiguous().view( - 1)

    preds_size = Variable(torch.IntTensor([preds.size(0)]))
    raw_pred = converter.decode(preds.data, preds_size.data, raw = True)
    sim_pred = converter.decode(preds.data, preds_size.data, raw = False)
    print('% - 20s => % - 20s' % (raw_pred, sim_pred))
```

CRNN 的特征图提取与表示代码执行后的输出如代码 5-3 所示。

【代码 5-3】 CRNN 的特征图提取与表示代码执行后的输出。

```
loading pretrained model from ./trained_model/netCRNN_24_10100.pth
cnn Sequential(
  (conv0): Conv2d(1, 64, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
  (relu0): ReLU(inplace = True)
  (pooling0): MaxPool2d(kernel_size = 2, stride = 2, padding = 0, dilation = 1, ceil_mode = False)
  (conv1): Conv2d(64, 128, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
  (relu1): ReLU(inplace = True)
  (pooling1): MaxPool2d(kernel_size = 2, stride = 2, padding = 0, dilation = 1, ceil_mode = False)
  (conv2): Conv2d(128, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
  (batchnorm2): BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_
running_stats = True)
```

```
11   (relu2): ReLU(inplace = True)
12   (conv3): Conv2d(256, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
13   (relu3): ReLU(inplace = True)
14   (pooling2): MaxPool2d(kernel_size = (2, 2), stride = (2, 1), padding = (0, 1), dilation = 1,
ceil_mode = False)
15   (conv4): Conv2d(256, 512, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
16   (batchnorm4): BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_
running_stats = True)
17   (relu4): ReLU(inplace = True)
18   (conv5): Conv2d(512, 512, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
19   (relu5): ReLU(inplace = True)
20   (pooling3): MaxPool2d(kernel_size = (2, 2), stride = (2, 1), padding = (0, 1), dilation = 1,
ceil_mode = False)
21   (conv6): Conv2d(512, 512, kernel_size = (2, 2), stride = (1, 1))
22   (batchnorm6): BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_
running_stats = True)
23   (relu6): ReLU(inplace = True)
24 )
25 rnn Sequential(
26   (0): BidirectionalLSTM(
27     (rnn): LSTM(512, 256, bidirectional = True)
28     (embedding): Linear(in_features = 512, out_features = 256, bias = True)
29   )
30   (1): BidirectionalLSTM(
31     (rnn): LSTM(256, 256, bidirectional = True)
32     (embedding): Linear(in_features = 512, out_features = 77, bias = True)
33   )
34 )
35 torch.Size([256, 8, 25])
36 鲁 ---- NN - DDDNN - 113 -- 3999 ---  => 鲁 NDN1339
```

此外，(conv3)：Conv2d(256，256，kernel_size＝(3，3)，stride＝(1，1)，padding＝(1，1))的卷积特征图可视化也将输出，如图 5-10 所示。

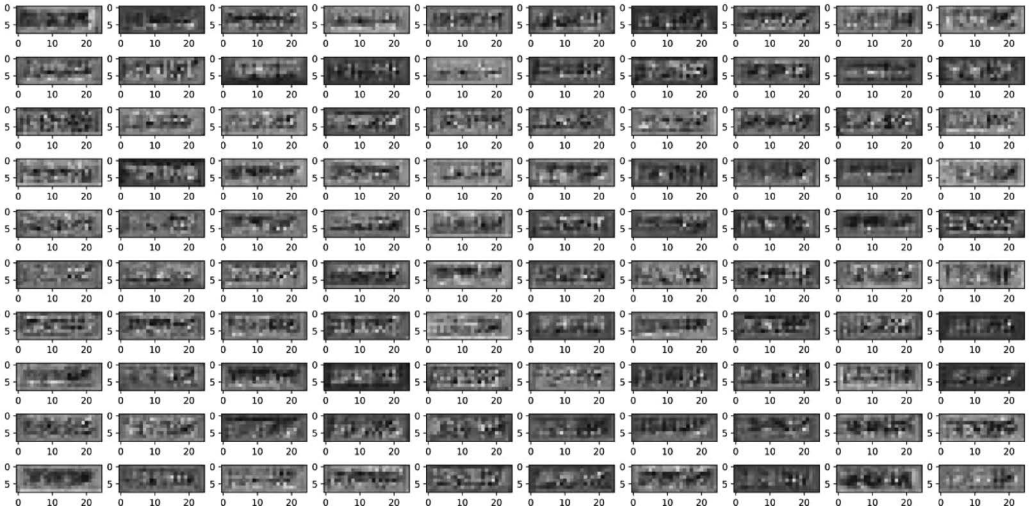

图 5-10　conv3 的卷积特征图可视化

5.2.4　池化操作的降维效果

在卷积神经网络架构中，池化层作为与卷积层并重的组件，扮演着不可或缺的角色，其核心功能在于通过降维操作降低计算复杂度，同时精练图像的关键特征。池化层通过将图像划分为多个非重叠区域，对每个区域执行特定操作，减少特征图的尺寸，加速后续计算流程，提升模型的整体效率。其中，最大池化是最常用的池化策略，它选取每个区域的最大值作为输出，这种机制不仅能保留区域内的显著特征，还能增强图像的不变性，提升模型的鲁棒性。与此同时，池化层对于多通道图像的处理采取独立通道操作，确保每种颜色信息都能得到充分且独立的特征提取，增加特征表示的丰富性和准确性。

池化层的多样化策略，如平均池化，为不同任务和数据分布提供了灵活的选择。平均池化通过计算区域内的平均值，适用于那些要求平滑响应的任务，其效果可能优于最大池化。更重要的是，池化操作在减少模型复杂度的同时，也有助于抑制过拟合现象，通过削减参数数量，增强模型对未知数据的泛化能力。合理配置池化层的超参数，如滤波器大小和步长，以及选择恰当的池化类型，对于优化卷积神经网络的性能和确保其在实际应用中的有效性至关重要。池化层的巧妙设计，提升了模型对图像特征的捕捉和理解能力，是卷积神经网络高效运作和广泛适用性的关键驱动力。

举个例子，对 CRNN 的最后一个池化层的降维效果进行特征图可视化，该池化层作用后的结果是大小为 torch.Size([512,2,27])的一个张量，现在随机选择 25 个进行可视化，得到(pooling3)：MaxPool2d(kernel_size=(2,2),stride=(2,1),padding=(0,1),dilation=1,ceil_mode=False)池化特征图可视化输出，如图 5-11 所示。

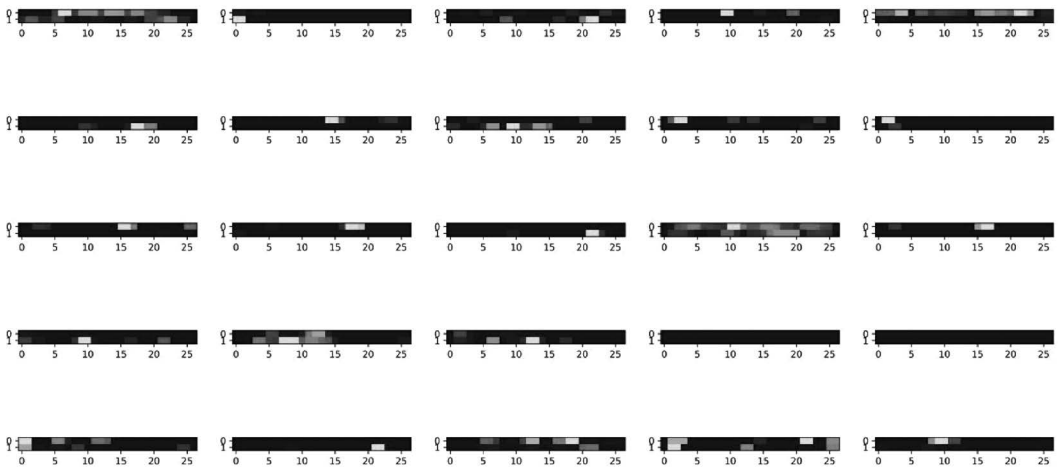

图 5-11　pooling3 池化特征图可视化

5.2.5　网络结构的定义与构建

CRNN 架构结合了卷积神经网络(CNN)和循环神经网络(RNN)的优势。它首先通过一系列卷积层和池化层对输入图像进行特征提取，随后利用双向 LSTM 层进行序列建模，最终实现从图像到文本序列的转换。网络结构定义如表 5-1 所示。

表 5-1 网络结构定义

类 型	设 置
转录	—
双向 LSTM	隐藏单元：256
双向 LSTM	隐藏单元：256
映射到序列	—
卷积	输出通道 512，卷积核大小 2×2，步长 1，填充 0
最大池化层	池化窗口 1×2，步长 2
批标准化	—
卷积	输出通道 512，卷积核大小 3×3，步长 1，填充 1
批标准化	—
卷积	输出通道 512，卷积核大小 3×3，步长 1，填充 1
最大池化层	池化窗口 1×2，步长 2
卷积	输出通道 256，卷积核大小 3×3，步长 1，填充 1
卷积	输出通道 256，卷积核大小 3×3，步长 1，填充 1
最大池化层	池化窗口 2×2，步长 2
卷积	输出通道 128，卷积核大小 3×3，步长 1，填充 1
最大池化层	池化窗口 2×2，步长 2
卷积	输出通道 64，卷积核大小 3×3，步长 1，填充 1
输入	W 图像宽度，H 图像高度，RGB 3 个通道，每个通道 8 位

在输入层中，接收 $W\times H$ 尺寸的 RGB 图像，每个像素点由 3 个 8 位通道组成，代表红、绿、蓝三色。

卷积层和池化层用于提取图像的基本特征。例如，最后一层卷积层使用 512 个输出通道，2×2 的卷积核大小，步长为 1，不进行填充。而池化层的例子比如最后一层卷积层之前的最大池化层，使用的是 1×2 的池化窗口，步长为 2，用于降低空间维度。其他的几层卷积层和池化层也是用于细化特征表示。整个过程由下而上，逐渐增大图像的宽度、高度和深度，以捕获更复杂的特征。批标准化(Batch Normalization)层穿插其中，用于加速训练过程并提高模型稳定性。

在一系列卷积和池化操作之后，特征图被映射到序列并送入两个双向 LSTM 层，每个隐藏单元数为 256。双向 LSTM 能够从前向后和从后向前同时处理序列信息，这在识别字符序列时尤为重要，因为它能利用上下文信息来增强预测的准确性。

整个 CRNN 通过整合 CNN 的强大视觉特征提取能力和 LSTM 的时间序列建模能力，可高效地将输入图像转换为可读的文本序列，其整合过程如代码 5-4～代码 5-9 所示。

(1) CRNN 结构定义的源码如代码 5-4 所示。

【代码 5-4】 CRNN 结构定义的源码。

```
1  #这段代码定义了一个车牌号识别的 CRNN 模型,其中网络结构包含卷积神经网络(CNN)和双向
   #长短时记忆网络(Bidirectional LSTM,BiLSTM)
2  #
3
4  import torch.nn as nn
```

```
5
6
7  class BidirectionalLSTM(nn.Module):
8
9      def __init__(self, nIn, nHidden, nOut):
10         super(BidirectionalLSTM, self).__init__()
11
12         self.rnn = nn.LSTM(nIn, nHidden, bidirectional = True)
13         self.embedding = nn.Linear(nHidden * 2, nOut)
14
15     def forward(self, input):
16         recurrent, _ = self.rnn(input)
17         T, b, h = recurrent.size()
18         t_rec = recurrent.view(T * b, h)
19
20         output = self.embedding(t_rec) #[T * b, nOut]
21         output = output.view(T, b, -1)
22
23         return output
24
25 class CRNN(nn.Module):
26
27     def __init__(self, imgH, nc, nclass, nh, n_rnn = 2, leakyRelu = False):
28         super(CRNN, self).__init__()
29         assert imgH % 16 == 0, 'imgH has to be a multiple of 16'
30
31         ks = [3, 3, 3, 3, 3, 3, 2]
32         ps = [1, 1, 1, 1, 1, 1, 0]
33         ss = [1, 1, 1, 1, 1, 1, 1]
34         nm = [64, 128, 256, 256, 512, 512, 512]
35
36         cnn = nn.Sequential()
37
38         def convRelu(i, batchNormalization = False):
39             nIn = nc if i == 0 else nm[i - 1]
40             nOut = nm[i]
41             cnn.add_module('conv{0}'.format(i),
42                            nn.Conv2d(nIn, nOut, ks[i], ss[i], ps[i]))
43             if batchNormalization:
44                 cnn.add_module('batchnorm{0}'.format(i), nn.BatchNorm2d(nOut))
45             if leakyRelu:
46                 cnn.add_module('relu{0}'.format(i),
47                                nn.LeakyReLU(0.2, inplace = True))
48             else:
49                 cnn.add_module('relu{0}'.format(i), nn.ReLU(True))
50
51         convRelu(0)
52         cnn.add_module('pooling{0}'.format(0), nn.MaxPool2d(2, 2))    #64×16×64
53         convRelu(1)
54         cnn.add_module('pooling{0}'.format(1), nn.MaxPool2d(2, 2))    #128×8×32
55         convRelu(2, True)
```

```
        convRelu(3)
        cnn.add_module('pooling{0}'.format(2),
                       nn.MaxPool2d((2, 2), (2, 1), (0, 1)))   #256×4×16
        convRelu(4, True)
        convRelu(5)
        cnn.add_module('pooling{0}'.format(3),
                       nn.MaxPool2d((2, 2), (2, 1), (0, 1)))   #512×2×16
        convRelu(6, True)                #512×1×16

        self.cnn = cnn
        self.rnn = nn.Sequential(
            BidirectionalLSTM(512, nh, nh),
            BidirectionalLSTM(nh, nh, nclass))

    def forward(self, input):
        #conv features
        conv = self.cnn(input)
        b, c, h, w = conv.size()
        assert h == 1, "the height of conv must be 1"
        conv = conv.squeeze(2)
        conv = conv.permute(2, 0, 1)    #[w, b, c]

        #rnn features
        output = self.rnn(conv)
        return output
```

（2）在类 CRNN 中，有 4 个卷积层的参数数组，如代码 5-5 所示。

【代码 5-5】 卷积层的参数数组。

```
ks = [3,3,3,3,3,3,2]
ps = [1,1,1,1,1,1,0]
ss = [1,1,1,1,1,1,1]
nm = [64,128,256,256,512,512,512]
```

代码 5-5 定义了一组卷积层的参数，包括卷积核大小 ks、padding 大小 ps、stride 大小 ss，以及每层输出通道数 nm。cnn＝nn. Sequential()表示创建一个序列容器，用于存放卷积神经网络的层。

（3）函数 convReLU 的定义，用于向 CNN 中添加卷积层、批量归一化层（可选）、激活函数层，如代码 5-6 所示。函数参数 i 表示层的索引。

【代码 5-6】 函数 convReLU 的定义。

```
def convRelu(i, batchNormalization=False):
    #卷积层的定义
    nIn = nc if i == 0 else nm[i - 1]
    nOut = nm[i]
    cnn.add_module('conv{0}'.format(i),
                   nn.Conv2d(nIn, nOut, ks[i], ss[i], ps[i]))
    #是否使用批量归一化
    if batchNormalization:
```

```
            cnn.add_module('batchnorm{0}'.format(i), nn.BatchNorm2d(nOut))
        #是否使用 LeakyReLU 激活函数
        if leakyRelu:
            cnn.add_module('relu{0}'.format(i),
                           nn.LeakyReLU(0.2, inplace = True))
        else:
            cnn.add_module('relu{0}'.format(i), nn.ReLU(True))
```

(4) 利用 convReLU 函数构建卷积神经网络结构，包括卷积层、批量归一化层(可选)、激活函数层和最大池化层，如代码 5-7 所示。

【代码 5-7】 利用 convReLU 函数构建卷积神经网络结构。

```
#构建卷积神经网络
convRelu(0)
cnn.add_module('pooling{0}'.format(0), nn.MaxPool2d(2, 2))        #64×16×64
convRelu(1)
cnn.add_module('pooling{0}'.format(1), nn.MaxPool2d(2, 2))        #128×8×32
convRelu(2, True)
convRelu(3)
cnn.add_module('pooling{0}'.format(2),nn.MaxPool2d((2, 2), (2, 1), (0, 1)))   #256×4×16
convRelu(4, True)
convRelu(5)
cnn.add_module('pooling{0}'.format(3),nn.MaxPool2d((2, 2), (2, 1), (0, 1)))   #512×2×16
convRelu(6, True)        #512×1×16
```

(5) CRNN 神经网络的存储，如代码 5-8 所示。将构建好的卷积神经网络存储在 self.cnn 中，用以表示 CNN，并定义一个序列容器 nn.Sequential，其中包含两个双向 LSTM 模块，用以表示 RNN。

【代码 5-8】 CRNN 神经网络的存储。

```
self.cnn = cnn
self.rnn = nn.Sequential(BidirectionalLSTM(512, nh, nh), BidirectionalLSTM(nh, nh,
nclass))
```

(6) forward 函数的定义，如代码 5-9 所示。该函数表示了模型的前向传播过程。首先，通过卷积神经网络处理输入 input，得到卷积特征 conv。然后，对 conv 进行维度调整，将其转换为适合输入 LSTM 中的形状。最后，将调整后的特征传递给 LSTM 层，得到最终的输出 output。

【代码 5-9】 forward 函数的定义。

```
def forward(self, input):
    #conv features
    conv = self.cnn(input)
    b, c, h, w = conv.size()
    assert h == 1, "the height of conv must be 1"
    conv = conv.squeeze(2)
    conv = conv.permute(2, 0, 1)        #[w, b, c]
```

```
        #rnn features
        output = self.rnn(conv)

        return output
```

CRNN 模型通过卷积神经网络提取图像特征，然后通过双向 LSTM 处理时序信息，最终输出车牌号的识别结果。这样的结构对于处理图像序列任务，尤其是文字识别任务，是一种常见的深度学习模型结构。

5.3 卷积神经网络的训练与评测

5.3.1 数据集准备与预处理

在计算机视觉的发展中，一个全面且质量卓越的数据集对于训练、测试和评估模型的性能起到关键作用。本节将详细探讨所采用的数据集——CBLPRD-330k（China Balanced License Plate Recognition Dataset 330k），它不仅为车牌识别模型提供了多样性丰富的图像，还考虑了平衡分布、图像质量等方面的因素。CBLPRD-330k 数据集样例如图 5-12 所示。

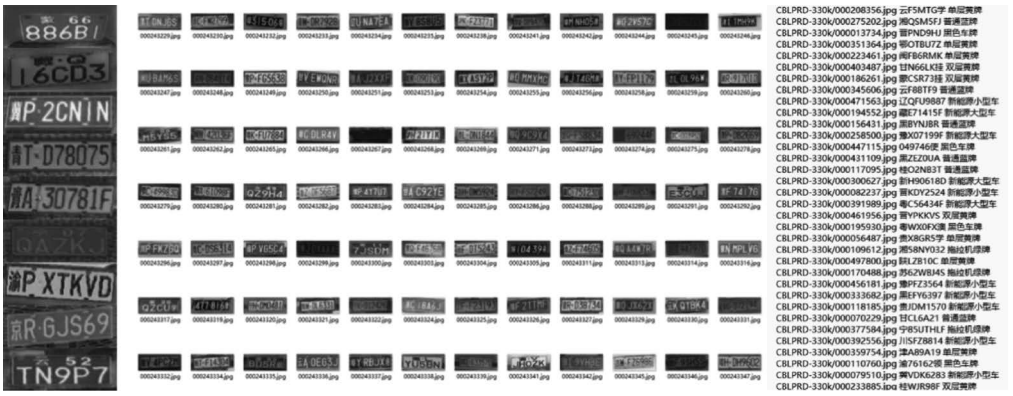

图 5-12 CBLPRD-330k 数据集样例

以下是对数据集的详细介绍。

1. 数据集背景

CBLPRD-330k 包含了 33 万张各类中国车牌的图片，旨在为车牌识别模型的研究和开发提供有力支持。

2. 理念和特点

数据集中的图像是精心采集得到的，以确保它们具有良好的图像质量。这一特点对于模型的训练和泛化能力至关重要。CBLPRD-330k 致力于涵盖各种类型的中国车牌，从普通私家车到特殊车辆，以确保数据集的平衡性。这有助于防止由于数据集偏斜而引起的模型训练问题。该数据集规模达到了 33 万张车牌图片，使其成为进行大规模模型训练的理想选择。这对于深度学习模型的性能提升至关重要。为鼓励研究者和开发者

充分利用该数据集，该数据集完全开源并免费提供，以促进学术研究和行业创新。

此外，该数据集已经按照“车牌图片路径—车牌号—车辆类型”进行了预处理，只需要关注“车牌图片路径—车牌号”即可。

3. 数据集验证与模型应用

数据提供方采用了 ResNet18 模型的前三个层进行验证，并使用 CTC loss 作为损失函数进行训练。验证结果表明，直接使用 CBLPRD-330k 训练的模型在一般停车场的车牌识别任务上表现优异。

在智能交通和智能安防领域，车牌识别技术得到了广泛应用。在研究和开发的道路上，一个全面且质量上乘的数据集是必不可少的。CBLPRD-330k 的推出旨在为车牌识别领域提供一个强大的数据集。通过不断优化数据集和模型，可以为研究者和开发者提供更好的资源，以促进车牌识别技术的不断创新和进步。

5.3.2　数据集解析与样本分析

在深度学习领域中，数据集的质量和构建方式对于模型的性能和泛化能力至关重要。数据集解析和样本分析是数据预处理的关键步骤之一，它直接影响到训练过程和模型的准确性。本章将深入讨论一个数据集解析的案例，重点介绍如何通过 Python 脚本处理文本数据集，并构建用于训练卷积循环神经网络（CRNN）的轻量级嵌入式数据库（LMDB）数据集。

（1）数据集解析。

数据集解析的目标是将原始数据集转换为模型可以接受的格式。在这个案例中，读取了一个文本文件 train.txt，其中包含了图像路径、标签以及其他可能的信息。数据集标签每一行的格式如代码 5-10 所示。

【代码 5-10】 数据集标签每一行的格式。

```
1 CBLPRD-330k/000272981.jpg 粤 Z31632D 新能源大型车
```

通过 Python 脚本，解析每一行的内容，提取图像路径和标签，构建用于训练的数据列表，如代码 5-11 所示。

【代码 5-11】 构建用于训练的数据列表。

```
1  #读取文本文件
2  file_path = '../data/CBLPRD-330k_v1/train.txt'
3  with open(file_path, 'r', encoding='utf-8') as file:
4      text_data = file.read()
5
6  #初始化空列表来存储数据
7  imagePathList = []
8  labelList = []
9
10 #按行分割文本数据
11 lines = text_data.strip().split('\n')
12
13 #遍历每一行数据
```

```
for line in lines:
    #按空格分割每行数据
    parts = line.split()

    #获取图像路径、标签和其他信息
    image_path = parts[0]
    label = ''.join(parts[1:-1])      #获取除了第一个和最后一个元素外的其他元素
#作为标签

    #将数据添加到相应的列表中
    imagePathList.append('../data/CBLPRD-330k_v1/' + image_path)
    labelList.append(label)
```

(2) LMDB 数据集创建。

LMDB 是一种轻量级嵌入式数据库,适用于大规模的深度学习数据集。通过使用 LMDB,可以高效地存储和读取数据。下面的 Python 脚本演示了如何将图像路径和标签写入 LMDB 数据库,如代码 5-12 所示。

【代码 5-12】 将图像路径和标签写入 LMDB 数据库。

```
def checkImageIsValid(imageBin):
    #... 检查图像是否为有效的函数 ...

def writeCache(env, cache):
    #... 将缓存写入 LMDB 数据库的函数 ...

def createDataset(outputPath, imagePathList, labelList, lexiconList=None, checkValid=True):
    #... 创建 LMDB 数据集的函数 ...

if __name__ == '__main__':
    #... 读取文本文件,解析数据 ...

    output_path = '../data/train_data'
    createDataset(output_path, imagePathList, labelList)
```

数据集解析和样本分析是构建高效深度学习模型的必要步骤之一。在代码 5-13 所示的示例中,演示了如何使用 Python 脚本解析文本数据集,构建 LMDB 格式的数据集。这个过程对于光学字符识别(OCR)等任务具有重要意义,它确保了模型在训练时能够有效地利用数据。在实际应用中,可以根据具体任务的需求进行相应的修改和调整。

【代码 5-13】 解析文本数据集,构建 LMDB 格式的数据集。

```
import os
import lmdb #install lmdb by "pip install lmdb"
import cv2
import numpy as np

def checkImageIsValid(imageBin):
    if imageBin is None:
        return False
```

```
    imageBuf = np.frombuffer(imageBin, dtype=np.uint8)
    img = cv2.imdecode(imageBuf, cv2.IMREAD_GRAYSCALE)
    imgH, imgW = img.shape[0], img.shape[1]
    if imgH * imgW == 0:
        return False
    return True

def writeCache(env, cache):
    with env.begin(write=True) as txn:
        for k, v in cache.items():
            if type(v) is str:
                txn.put(k.encode(), v.encode())
            else:
                txn.put(k.encode(), v)

def createDataset(outputPath, imagePathList, labelList, lexiconList=None, checkValid=True):
    """
    Create LMDB dataset for CRNN training.

    ARGS:
        outputPath    : LMDB output path
        imagePathList : list of image path
        labelList     : list of corresponding groundtruth texts
        lexiconList   : (optional) list of lexicon lists
        checkValid    : if true, check the validity of every image
    """
    assert(len(imagePathList) == len(labelList))
    nSamples = len(imagePathList)
    env = lmdb.open(outputPath, map_size=10995116277)
    cache = {}
    cnt = 1
    for i in range(nSamples):
        imagePath = imagePathList[i]
        label = labelList[i]
        if not os.path.exists(imagePath):
            print('%s does not exist' % imagePath)
            continue
        with open(imagePath, 'rb') as f:
            imageBin = f.read()
        if checkValid:
            if not checkImageIsValid(imageBin):
                print('%s is not a valid image' % imagePath)
                continue

        imageKey = 'image-%09d' % cnt
        labelKey = 'label-%09d' % cnt
        cache[imageKey] = imageBin
        cache[labelKey] = label
        if lexiconList:
            lexiconKey = 'lexicon-%09d' % cnt
            cache[lexiconKey] = lexiconList[i]
```

```
        if cnt % 1000 == 0:
            writeCache(env, cache)
            cache = {}
            print('Written %d / %d' % (cnt, nSamples))
        cnt += 1
    nSamples = cnt - 1
    cache['num-samples'] = str(nSamples)
    writeCache(env, cache)
    print('Created dataset with %d samples' % nSamples)

if __name__ == '__main__':
    #读取文本文件
    file_path = '../data/CBLPRD-330k_v1/train.txt'
    with open(file_path, 'r',encoding='utf-8') as file:
        text_data = file.read()

    #初始化空列表来存储数据
    imagePathList = []
    labelList = []
    lexiconList = []

    #按行分割文本数据
    lines = text_data.strip().split('\n')

    #遍历每一行数据
    for line in lines:
        #按空格分割每行数据
        parts = line.split()

        #获取图像路径、标签和其他信息
        image_path = parts[0]
        label = ''.join(parts[1:-1])      #获取除了第一个和最后一个元素外的其他
#元素作为标签
        #lexicon = parts[-1]

        #将数据添加到相应的列表中
        imagePathList.append('../data/CBLPRD-330k_v1/'+(image_path))
        labelList.append(label)
        #lexiconList.append(lexicon)       #假设所有数据都是有效的,也可以根据需
#要进行检查

    #打印结果
        #print("imagePathList:", imagePathList)
        print("labelList:", labelList)
        #print("lexiconList:", lexiconList)

        output_path='../data/train_data'
        createDataset(output_path,imagePathList,labelList,lexiconList)
```

5.3.3　网络模型的训练过程

深度学习模型的训练是构建高性能文本识别系统的关键步骤。本章将详细介绍基于卷积循环神经网络(CRNN)的文字识别模型的训练过程,包括数据集准备、模型配置、训练参数设置、数据加载、模型初始化、训练迭代以及模型评估,详见代码 5-14～代码 5-20。

(1) 引入必要的库和模块,如代码 5-14 所示。

【代码 5-14】 引入必要的库和模块。

```
#引入必要的库和模块
from __future__ import print_function
from __future__ import division
import argparse
import random
import torch
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable
import numpy as np
import os
import utils
import dataset
import models.crnn as crnn
import torch.nn as nn
```

(2) 数据集的准备,如代码 5-15 所示。训练数据集采用 LMDB 格式,包括训练集和验证集。通过解析 train.txt 和 val.txt 文件,提取图像路径、标签等信息,为模型提供标记好的数据样本。LMDB 格式具有高效读取的优势,适用于大规模深度学习任务。

【代码 5-15】 数据集的准备。

```
#读取文本文件
file_path = '../data/CBLPRD-330k_v1/train.txt'
with open(file_path, 'r', encoding='utf-8') as file:
 text_data = file.read()

#初始化空列表来存储数据
imagePathList = []
labelList = []
lexiconList = []

#按行分割文本数据
lines = text_data.strip().split('\n')

#遍历每一行数据
for line in lines:
    #按空格分割每行数据
    parts = line.split()
```

```
18
19      #获取图像路径、标签和其他信息
20      image_path = parts[0]
21      label = ''.join(parts[1:-1])      #获取除了第一个和最后一个元素外的其他元素
#作为标签
22
23      #将数据添加到相应的列表中
24      imagePathList.append('../data/CBLPRD-330k_v1/' + (image_path))
25      labelList.append(label)
```

(3) 模型的配置,如代码 5-16 所示,输入图片大小为高 32、宽 100,LSTM 隐藏状态大小为 256,字符集包含中文和英文字符,共 77 个字符。

【代码 5-16】 数据集的准备。

```
1  #模型配置
2  opt.imgH = 32
3  opt.imgW = 100
4  opt.nh = 256
5  opt.alphabet = '京沪津渝冀晋蒙辽吉黑苏浙皖闽赣鲁豫鄂湘粤桂琼川贵云藏陕甘青宁新港
澳挂学领使临 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZIO'
```

(4) 训练参数设置,如代码 5-17 所示。学习率为 0.01,使用 CTC(Connectionist Temporal Classification)损失函数,支持 Adam 和 Adadelta 两种优化器选择。

【代码 5-17】 训练参数设置。

```
1  #训练参数设置
2  opt.lr = 0.01
3  opt.adadelta = True
4  criterion = nn.CTCLoss()
```

(5) 模型初始化和加载,如代码 5-18 所示。模型采用 CRNN 结构,通过 weights_init 函数对网络参数进行初始化。支持加载预训练模型,方便在已有基础上进行进一步训练。

【代码 5-18】 模型初始化和加载。

```
1  #模型初始化和加载
2  crnn = crnn.CRNN(opt.imgH, nc, nclass, opt.nh)
3  crnn.apply(weights_init)
4  if opt.pretrained != '':
5      print('loading pretrained model from %s' % opt.pretrained)
6      crnn.load_state_dict(torch.load(opt.pretrained))
7  print(crnn)
```

(6) 数据加载和模型训练,如代码 5-19 所示。通过 PyTorch 的 DataLoader 加载训练集数据,采用随机采样器(Random Sequential Sampler)或随机采样进行数据加载。采用 trainBatch 函数进行每个 batch 的训练,计算损失并更新模型参数。

【代码 5-19】 数据加载和模型训练。

```
#数据加载和模型训练
train_dataset = dataset.lmdbDataset(root=opt.trainRoot)  #创建 LMDB 格式的训练数据集
assert train_dataset

#根据是否采用随机采样选择相应的采样器
if not opt.random_sample:
    sampler = dataset.randomSequentialSampler(train_dataset, opt.batchSize)  #使用
#随机序列采样器
else:
    sampler = None

train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=opt.batchSize,
    shuffle=True, sampler=sampler,
    num_workers=int(opt.workers),
    collate_fn=dataset.alignCollate(imgH=opt.imgH, imgW=opt.imgW, keep_ratio=
opt.keep_ratio))     #创建训练数据加载器

#初始化损失均值计算器
loss_avg = utils.averager()

#设置优化器
if opt.adam:
    optimizer = optim.Adam(crnn.parameters(), lr=opt.lr,
                           betas=(opt.beta1, 0.999))        #使用 Adam 优化器
elif opt.adadelta:
    optimizer = optim.Adadelta(crnn.parameters())           #使用 Adadelta 优化器
else:
    optimizer = optim.RMSprop(crnn.parameters(), lr=opt.lr)    #使用 RMSProp 优化器

#训练迭代
for epoch in range(opt.nepoch):
    train_iter = iter(train_loader)
    i = 0
    while i < len(train_loader):
        #开启梯度计算
        for p in crnn.parameters():
            p.requires_grad = True
        crnn.train()

        #进行单批次训练
        cost = trainBatch(crnn, criterion, optimizer)
        loss_avg.add(cost)
        i += 1

        #每隔一定间隔显示训练信息
        if i % opt.displayInterval == 0:
            print('[%d/%d][%d/%d] Loss: %f' %
                  (epoch, opt.nepoch, i, len(train_loader), loss_avg.val()))
            loss_avg.reset()
```

```
49
50          #每隔一定间隔在验证集上进行模型评估
51          if i % opt.valInterval == 0:
52              val(crnn, test_dataset, criterion)
53
54          #每隔一定间隔保存模型参数
55          if i % opt.saveInterval == 0:
56              torch.save(
57                  crnn.state_dict(), '{0}/netCRNN_{1}_{2}.pth'.format(opt.expr_dir,
epoch, i))
```

(7) 模型评估,如代码 5-20 所示,通过验证集对训练的模型进行评估。采用 val 函数,对一定数量的验证集样本进行识别并计算准确率。此过程中,模型参数处于冻结状态,不进行梯度更新。

【代码 5-20】 模型评估。

```
1  #模型评估
2  def val(net, dataset, criterion, max_iter=100):
3      print('Start val')
4
5      for p in crnn.parameters():
6          p.requires_grad = False    #关闭梯度计算,因为在验证阶段不进行参数更新
7
8      net.eval()                     #设置模型为评估模式,影响一些模型层的行为,例如
Batch Normalization 和 Dropout
9
10     #创建用于验证集的数据加载器
11     data_loader = torch.utils.data.DataLoader(
12         dataset, shuffle=True, batch_size=opt.batchSize, num_workers=int(opt.workers))
13     val_iter = iter(data_loader)
14
15     i = 0
16     n_correct = 0                              #记录正确预测的样本数量
17     loss_avg = utils.averager()                #记录损失的均值
18
19     max_iter = min(max_iter, len(data_loader))
20     for i in range(max_iter):
21         data = val_iter.__next__()             #获取验证集的一个批次数据
22         i += 1
23         cpu_images, cpu_texts = data
24         batch_size = cpu_images.size(0)
25
26         image = cpu_images.cuda()              #将输入图像数据移动到 GPU 上
27         t, l = converter.encode(cpu_texts)     #对标签进行编码,生成训练所需的张量
28
29         preds = crnn(image).cpu()              #将输入图像数据通过模型进行前向传播
30         preds_size = Variable(torch.IntTensor([preds.size(0)] * batch_size))
31         cost = criterion(preds, t, preds_size, l) / batch_size      #计算 CTC 损失
32         loss_avg.add(cost)
33
```

```
        _, preds = preds.max(2)
        preds = preds.transpose(1, 0).contiguous().view(-1)
        sim_preds = converter.decode(preds.data, preds_size.data, raw=False)   #解码模
#型输出,得到预测的文本

        #计算准确率,逐样本比较预测结果和真实标签
        for pred, target in zip(sim_preds, cpu_texts):
            if pred == target.lower():
                n_correct += 1

        accuracy = n_correct / float(max_iter * opt.batchSize)      #计算准确率
        print('Test loss: %f, accuray: %f' % (loss_avg.val(), accuracy))
```

通过本章对代码的讲解,详细了解深度学习模型在文本识别任务中的训练过程。从数据集准备到模型配置,再到模型初始化和加载,最后进行数据加载和模型训练,形成了一套完整的训练流程。通过合理设置训练参数,不断迭代训练,能够得到在验证集上准确度较高的文本识别模型。

这一过程中,采用了 LMDB 格式的数据集,使用了 CRNN 模型结构,并结合 CTC 损失函数进行训练。通过验证集的评估,可以及时发现模型的性能,并根据需要进行调整和优化。

希望本章内容对深度学习模型的训练有所帮助,读者可以根据具体任务和数据集的特点进行相应的调整和扩展,以获得更好的模型性能。

5.3.4　网络模型的性能测试与评估

根据前文所述,网络模型的训练和测试评估是同时进行的,整个过程共进行了 25 个 epoch。如图 5-13～图 5-16 所示,每个测试数据集的标签均以“gt:”开头,后跟车牌号。

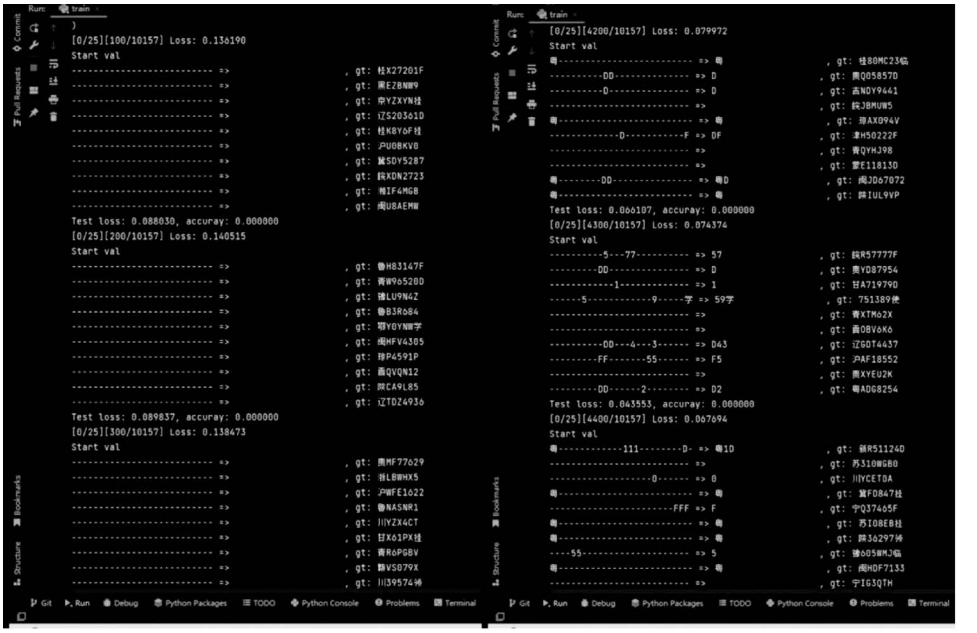

图 5-13　模型刚开始训练时在测试数据集上的评估

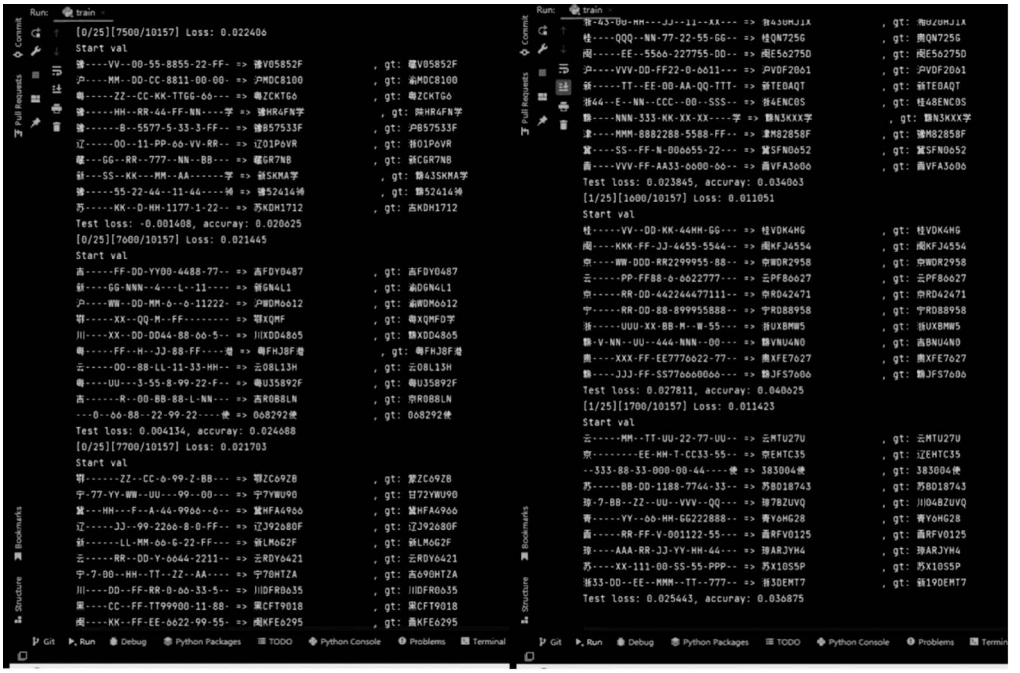

图 5-14　模型初步训练一段时间后在测试数据集上的评估

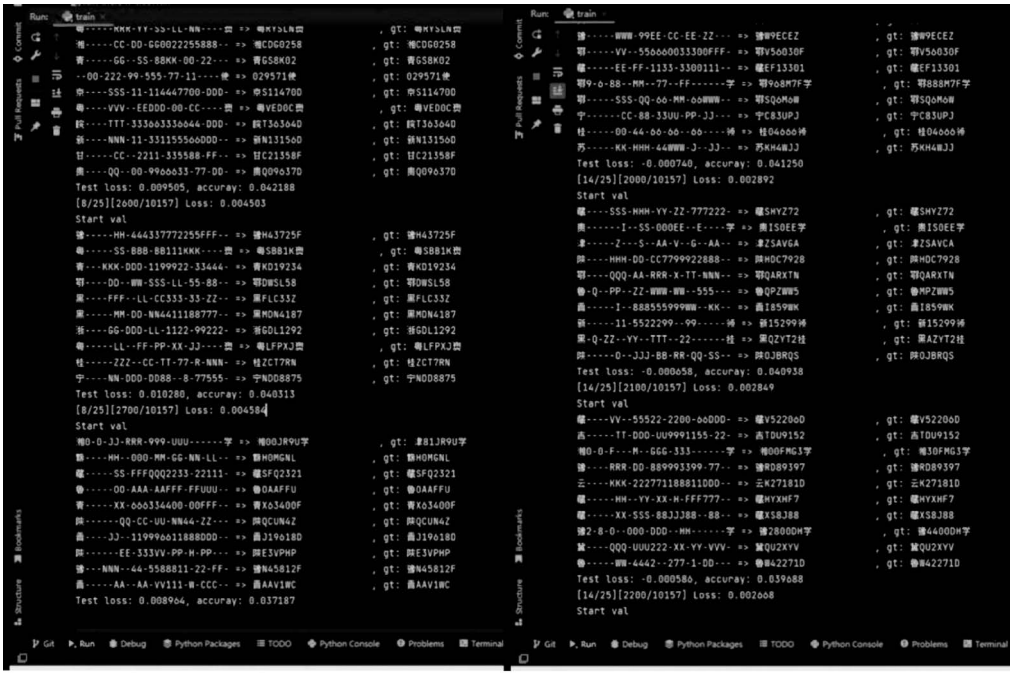

图 5-15　模型训练 10 个 epoch 左右时在测试数据集上的评估

在图中，“=>”左侧展示了模型的识别过程，而“=>”右侧则呈现了模型的识别结果。初期，模型无法准确识别车牌号中的字符，随着训练的进行，车牌号中的字符逐渐被正确识

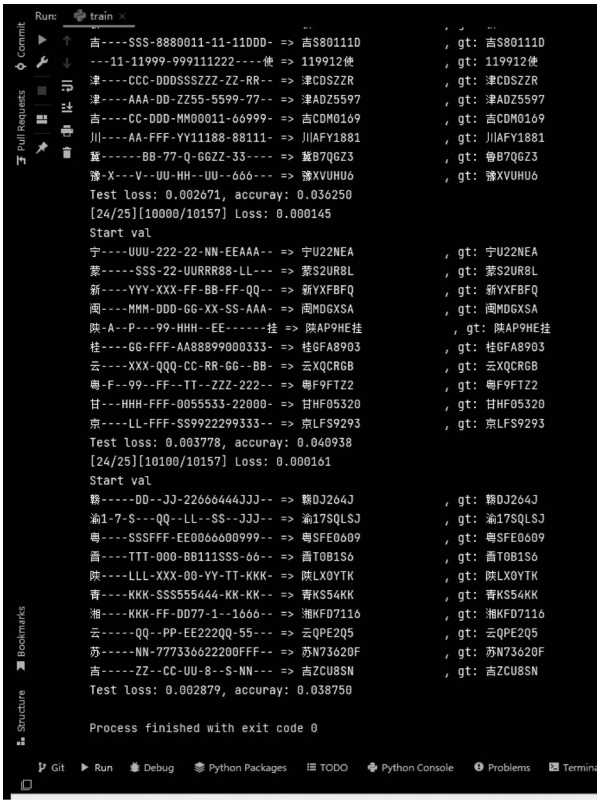

图 5-16　模型训练结束时在测试数据集上的评估

别，尽管每个字符的准确性仍然不够理想。最终，在训练的末尾阶段，观察到终端中显示的车牌识别结果与标签已经一一对应，呈现出更为准确的字符识别情况。

5.4　应用集成开发与界面设计

1. 用户界面的设计与交互

在车牌号识别应用中，一个好的用户界面应直观、易用，而良好的交互设计则能确保用户在使用过程中顺畅地完成操作。

首先，界面设计中的布局要直观明了。界面元素的排列应符合用户的思维逻辑，帮助用户快速理解和掌握应用功能，减少使用中的困扰，提升整体体验。其次，界面元素和标识应友好。按钮、文本框和图标等元素应具备友好的外观，并通过清晰的标识反映其功能，确保用户在每一步操作中都有明确的引导，避免混淆。

在交互设计方面，首先要考虑用户的操作习惯和预期行为。界面交互应符合用户在其他类似应用中的习惯，降低学习成本。操作流程要简洁且有层次感。用户不喜欢烦琐的流程，通过合理的层级结构，用户能迅速找到所需功能。及时的反馈和提示信息非常重要。无论操作成功与否，应用都应通过清晰的提示信息告知用户当前状态，帮助其理解应用的工作原理，减少困惑。

为了实现以上设计和交互目标，选用 PySide6 来制作车牌字符识别应用的集成界面。PySide6 基于 Qt 技术，提供丰富的图形界面和应用开发功能，PySide6 中的常见模块包括 QtWidget、QtCore、QtGui、QtNetwork 和 QtSql，分别用于创建图形用户界面、提供 Qt 核心功能、图像处理、网络编程和数据库操作。PySide6 具有以下优点：跨平台性好，能在 Windows、Linux 和 macOS 等多个平台上运行，确保应用的可移植性；图形界面设计工具强大，如 Qt Designer，通过拖曳组件的方式可以轻松设计用户界面，加快开发速度；丰富的部件库能用于构建各种功能和交互式元素，如按钮、标签、列表框等，用于展示和操作图像及识别结果；信号与槽机制使得不同组件之间的交互简单而灵活，适用于异步操作，如选择图片、处理图像和显示识别结果。

2. 批量车牌识别功能

为增强应用的实用性，引入了批量车牌识别功能。该功能允许用户一次性输入多个车牌号图片进行识别，便于用户批量验证模型性能。在界面设计上，需要提供批量输入的接口，并展示每个车牌的识别结果。通过直观的图表或列表形式，用户能够清晰地了解整体评测结果，为模型性能的改进提供依据。

为了实现批量车牌字符识别集成应用，这里从功能描述、界面设计、与模型的集成、用户交互、界面布局等方面来进行考虑。

(1) 功能描述。

该应用程序提供了批量车牌字符识别功能，用户可以选择多张车牌图片进行字符识别。主要有以下 4 个特点。

① 提供图形用户界面，包含图片显示区域、选择按钮和识别结果列表；

② 支持一次性选择多张车牌图片进行批量字符识别；

③ 通过 PyTorch 加载 CRNN 模型，将选中的图片传递给模型进行识别；

④ 图片及其字符识别结果实时显示在界面上。

(2) 界面设计。

界面设计主要涉及以下 3 项。

① 使用 QMainWindow 作为主窗口，包含多个 QLabel 用于显示车牌图片，一个 QPushButton 用于触发图片选择，以及一个 QListWidget 用于显示识别结果。

② 图片显示区域采用 QGridLayout 进行布局，每行显示 3 张图片。

③ 通过 QListWidget 单独显示字符识别结果列表。

(3) 与模型的集成。

与算法模块的集成有如下两方面：

① 通过 PyTorch 加载 CRNN 模型，该模型用于字符识别；

② 用户选择图片后，应用程序将图片传递给模型，获取字符识别结果。

(4) 用户交互。

用户交互主要分为如下 3 个步骤：

① 用户通过单击“选择图片”按钮与应用程序进行交互；

② 选择完成图片后，应用程序自动对选中的车牌图片进行批量字符识别；

③ 识别结果以图像和字符识别字符串的形式显示在界面上。

(5) 界面布局。

界面布局分为如下 3 部分：

① GUI 界面采用 QGridLayout 进行布局，以网格状排列多个 QLabel 用于显示图片；

② 每行显示 3 张图片，用户可以根据实际需要进行调整；

③ QListWidget 用于单独显示字符识别结果列表。

通过 PySide6 和 PyTorch 的集成，该应用程序提供了一种便捷的方式，让用户通过图形界面轻松进行批量车牌字符识别，可以选择最多 9 张图片进行批量识别。批量选择图片功能，如图 5-17 所示；批量显示识别结果，如图 5-18 所示。

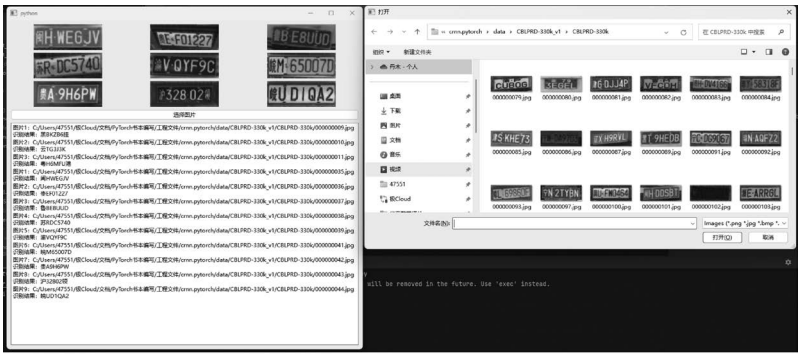

图 5-17　批量选择图片功能

图 5-18　批量显示识别结果

对车牌识别功能进行集成，利用 PySide 部署到用户界面，如代码 5-21 所示。

【代码 5-21】 批量车牌文本识别功能应用与用户界面集成。

```
#导入 PySide6 库中的一些模块和类
from PySide6.QtWidgets import QApplication, QMainWindow, QVBoxLayout, QLabel, QPushButton, QFileDialog, QWidget, \
    QListWidget, QGridLayout, QListWidgetItem
from PySide6.QtCore import Qt, QSize
from PySide6.QtGui import QPixmap
from PIL import Image
import torch
from torchvision import transforms
from models.crnn import CRNN      #导入名为 CRNN 的模型类(请确保有定义这个类)
import utils
import dataset
from torch.autograd import Variable

#定义一个名为 BatchRecognitionApp 的类,继承自 QMainWindow
class BatchRecognitionApp(QMainWindow):
    def __init__(self):
        super(BatchRecognitionApp, self).__init__()

        #模型文件路径
        model_path = '../trained_model/netCRNN_24_10100.pth'

        #创建一个 CRNN 模型实例,并加载预训练权重
        self.model = CRNN(32, 1, 77, 256)
        self.model = torch.nn.DataParallel(self.model)
        self.model.load_state_dict(torch.load(model_path))

        #存储图片 QLabel 和结果列表的列表
        self.image_labels = []
        self.result_list = QListWidget()

        #创建一个按钮用于选择图片
        select_button = QPushButton("选择图片", self)
        select_button.clicked.connect(self.select_images)

        #创建一个网格布局
        layout = QGridLayout()
        row = 0
        col = 0

        #循环创建用于显示图片的 QLabel,每行显示 3 个
        for _ in range(9):    #你可以根据需要调整这个数字,决定每行显示的图片数量
            image_label = QLabel()
            image_label.setAlignment(Qt.AlignCenter)
            self.image_labels.append(image_label)
            layout.addWidget(image_label, row, col)
            col += 1
            if col == 3:    #你也可以根据需要调整这个数字,决定每行显示的列数
```

```
48                  row += 1
49                  col = 0
50
51          #增加一个按钮和结果列表
52          row += 1
53          layout.addWidget(select_button, row, 0, 1, 3)
54          row += 1
55          layout.addWidget(self.result_list, row, 0, 1, 3)
56
57          #创建一个 QWidget 作为中心控件,并将布局设置为这个 QWidget 的布局
58          central_widget = QWidget()
59          central_widget.setLayout(layout)
60
61          #将中心控件设置为 QMainWindow 的中心控件
62          self.setCentralWidget(central_widget)
63
64      #处理选择图片按钮的单击事件
65      def select_images(self):
66          file_dialog = QFileDialog()
67          file_dialog.setFileMode(QFileDialog.ExistingFiles)
68          file_dialog.setNameFilter("Images (*.png *.jpg *.bmp *.tif)")
69
70          if file_dialog.exec_():
71              image_paths = file_dialog.selectedFiles()
72              self.process_images(image_paths)
73
74      #处理选择的图片,进行识别
75      def process_images(self, image_paths):
76          char_sets = '京沪津渝冀晋蒙辽吉黑苏浙皖闽赣鲁豫鄂湘粤桂琼川贵云藏陕甘青
#宁新港澳挂学领使临 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZIO'
77          converter = utils.strLabelConverter(char_sets)
78          transformer = dataset.resizeNormalize((100, 32))
79
80          #设置模型为评估模式
81          self.model.eval()
82
83          for index, image_path in enumerate(image_paths):
84              image = Image.open(image_path).convert('L')
85              image = transformer(image)
86
87              image = image.view(1, *image.size())
88              image = Variable(image)
89
90              with torch.no_grad():
91                  #对图像进行预测
92                  result = self.model(image)
93                  _, preds = result.max(2)
94                  preds = preds.transpose(1, 0).contiguous().view(-1)
95                  preds_size = Variable(torch.IntTensor([preds.size(0)]))
96                  raw_pred = converter.decode(preds.data, preds_size.data, raw=True)
97                  sim_pred = converter.decode(preds.data, preds_size.data, raw=False)
```

```
98
99              #显示识别结果
100                 self.display_result(index, image_path, sim_pred)
101
102         #显示识别结果
103         def display_result(self, index, image_path, sim_pred):
104             #加载图片并调整大小
105             pixmap = QPixmap(image_path)
106             pixmap = pixmap.scaledToWidth(150, Qt.SmoothTransformation)
107
108             #将图片设置到对应的 QLabel 中
109             self.image_labels[index].setPixmap(pixmap)
110
111             #创建一个结果项,并将其添加到结果列表中
112             result_item = QListWidgetItem(f"图片{index + 1}:{image_path}\n 识别结
果:{sim_pred}")
113             self.result_list.addItem(result_item)
114
115     #应用程序入口
116     if __name__ == '__main__':
117         app = QApplication([])
118         window = BatchRecognitionApp()
119         window.show()
120         app.exec()
```

3. 单个车牌文本识别功能

除了批量识别外,卷积神经网络模型还具备对单个车牌文本识别功能。通过简洁的界面,用户可以方便地输入单个车牌图片并获取模型的识别结果。

在单个车牌文本识别功能中,界面设计应注重用户友好性,通过直观的操作引导用户完成车牌图片的输入和识别。同时,为了提高用户信任度,可以考虑自行在结果展示中增加置信度或其他相关信息,使用户更加信任模型的识别结果。单个车牌文本识别界面如图 5-19 所示;单个车牌文本识别结果如图 5-20 所示。

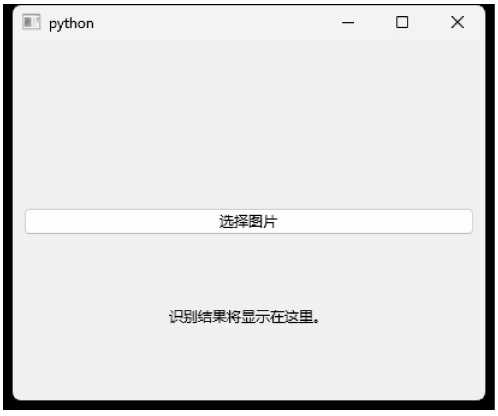

图 5-19　单个车牌文本识别界面

图 5-20　单个车牌文本识别结果

单个车牌文本识别功能应用与用户界面集成，如代码 5-22 所示。

【代码 5-22】 单个车牌文本识别功能应用与用户界面集成。

```
#引入必要的库
from PySide6.QtWidgets import QApplication, QMainWindow, QVBoxLayout, QLabel, QPushButton, QFileDialog, QWidget
from PySide6.QtCore import Qt
from PySide6.QtGui import QPixmap
from PIL import Image
import torch
from torchvision import transforms
from models.crnn import CRNN            #导入你的 CRNN 模型类
import utils
import dataset
from torch.autograd import Variable

#创建主窗口类
class BatchRecognitionApp(QMainWindow):
    def __init__(self):
        super(BatchRecognitionApp, self).__init__()

        #加载 CRNN 模型
        model_path = '../trained_model/netCRNN_24_10100.pth'

        self.model = CRNN(32, 1, 77, 256)
        self.model = torch.nn.DataParallel(self.model)
        self.model.load_state_dict(torch.load(model_path))

        #创建界面元素
        self.image_label = QLabel()                    #用于显示图片的 QLabel
        self.image_label.setAlignment(Qt.AlignCenter)
        self.result_label = QLabel("识别结果将显示在这里。")    #显示识别结果的 QLabel
        self.result_label.setAlignment(Qt.AlignCenter)
        select_button = QPushButton("选择图片", self)    #选择图片的按钮
        select_button.clicked.connect(self.select_images)    #连接按钮单击事件

```

```
        #设置布局
        layout = QVBoxLayout()
        layout.addWidget(self.image_label)
        layout.addWidget(select_button)
        layout.addWidget(self.result_label)
        central_widget = QWidget()
        central_widget.setLayout(layout)
        self.setCentralWidget(central_widget)

    def select_images(self):
        file_dialog = QFileDialog()                          #创建文件选择对话框
        file_dialog.setFileMode(QFileDialog.ExistingFiles)   #设置文件选择模式为选
#择多个文件
        file_dialog.setNameFilter("Images (*.png *.jpg *.bmp *.tif)")  #设置
#文件过滤器

        if file_dialog.exec():          #如果文件对话框执行成功
            image_paths = file_dialog.selectedFiles()    #获取选中的文件路径
            self.process_images(image_paths)             #调用处理图像的方法

    def process_images(self, image_paths):
        #图像预处理
        alphabet = '京沪津渝冀晋蒙辽吉黑苏浙皖闽赣鲁豫鄂湘粤桂琼川贵云藏陕甘青
#宁新港澳挂学领使临 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZIO'
        converter = utils.strLabelConverter(alphabet)    #创建字符转换器
        transformer = dataset.resizeNormalize((100, 32))    #创建图像转换器
        self.model.eval()                                    #设置模型为评估模式

        results = []

        for image_path in image_paths:
            image = Image.open(image_path).convert('L')  #以灰度模式打开图像
            image = transformer(image)                   #对图像进行预处理
            image = image.view(1, *image.size())
            image = Variable(image)

            with torch.no_grad():
                result = self.model(image)
                _, preds = result.max(2)
                preds = preds.transpose(1, 0).contiguous().view(-1)
                preds_size = Variable(torch.IntTensor([preds.size(0)]))
                raw_pred = converter.decode(preds.data, preds_size.data, raw=True)
                sim_pred = converter.decode(preds.data, preds_size.data, raw=False)

            results.append((image_path, sim_pred))

        self.display_results(results)

    def display_results(self, results):
        for image_path, sim_pred in results:
            pixmap = QPixmap(image_path)
```

```
            pixmap = pixmap.scaledToWidth(200, Qt.SmoothTransformation)
            self.image_label.setPixmap(pixmap)
            self.result_label.setText(f"图片:{image_path}\n识别结果:{sim_pred}")

if __name__ == '__main__':
    app = QApplication([])                  #创建应用程序对象
    window = BatchRecognitionApp()          #创建主窗口对象
    window.show()                           #显示主窗口
    app.exec()                              #运行应用程序
```

5.5　本章小结

本章深入探讨了分类识别技术的应用，涵盖了多个关键方面。首先，通过5.1节介绍了分类识别的应用背景与动机，阐述了该技术在实际场景中的重要性和推动力。

5.2节聚焦于卷积循环神经网络(CRNN)结构的设计与构建。解释了卷积核的作用与选择、特征图提取与表示、池化操作的降维效果以及网络结构的定义与构建等关键概念，为后续训练与评测奠定了基础。

5.3节着重介绍了卷积神经网络的训练与评测过程。通过数据集准备与预处理，确保了输入数据的质量与一致性。接着深入探讨了数据集的解析与样本分析，帮助读者更好地理解所使用的数据。最后详细阐述了网络模型的训练过程和性能测试与评估的关键步骤，使读者能够全面了解模型的训练效果。

5.4节聚焦于应用集成开发与界面设计。详细介绍了用户界面的设计与交互，强调了用户体验的重要性。然后展示了批量和单个车牌文本识别功能的实现，为用户提供了方便快捷的批量识别服务。

通过对本章内容的深入学习与实践，读者不仅可以掌握分类识别技术的理论知识，还可以了解其在实际应用中的具体操作和开发过程。希望本章内容能够为读者在分类识别领域的学习与应用提供一定的参考。