教育部高等学校大学计算机课程教学指导委员会

中国大学生计算机设计大赛

软件开发类作品文档简要要求

作品编号：_____

作品名称：基于机器学习的验证码识别平台

作　　者：_____

版本编号：_____

填写日期：2020.5.20

填写说明：

1、本文档适用于**所有**涉及软件开发的作品，包括：软件应用与开发、大数据、人工智能、物联网应用；

2、正文一律用五号宋体，一级标题为二号黑体，其他级别标题如有需要，可根据需要设置；

3、本文档为简要文档，不宜长篇大论简明扼要为上；

4、提交文档时，以 PDF 格式提交本文档；

5、本文档内容是正式参赛内容组成部分，务必真实填写。如不属实，将导致奖项等级降低甚至终止本作品参加比赛。

目　录

第一章 需求分析

1.1 需求背景

随着人工智能技术的发展，信息技术行业与软件行业对人工智能的需求也越来越大，而对于爬虫或者数据挖掘软件来说，最头疼的莫过于程序卡在验证码上，这样既无法爬取到数据，也很容易降低效率。因此验证码识别对于数据分析师或者爬虫程序员来说是非常需要的。同时，通俗来说，验证码识别不仅仅只可以识别验证码。由于验证码噪点多，干扰线粗长，字符形状不固定，其识别难度是非常大的，而一旦能识别验证码，大部分可读性字符，例如交通摄像头拍摄车牌识别，文章书籍识别，场景标题监测，工厂流水线货物产品检测等等都可以实现。

并且人工智能技术融入当代互联网技术、自动化电子技术，信息技术这些已经是大势所趋，难以逆反，将其接受，容纳，转化为生产技术的一部分是至关重要的。验证码分析识别同样如此，在自动获取网络数据，分析信息数据等等过程中，都需要绕过验证码进行大量识别，因此验证码识别是生产过程中虽然毫不起眼当时也非常重要的一环。

1.2 功能性需求

1.2.1 验证码生成性功能与标准验证码库

鉴于目前并没有找到官方统一的验证码标准库，并且经过大量调查，不少网站验证码生成都基于 Captcha 库，因此将 Captcha 验证码生成算法（即第三方库）作为理论与实践的数据标准。由于在制作时考虑到日后神经网络训练所需配置高低的问题，所以验证码生成的数据只由数字组成，这能降低开发和训练所带来的高昂成本。

验证码生成不仅需要带有噪点、干扰线、以及形状变化等特点，同时在同一张图片中还需要具备不同的颜色，以保证其真实性和难度，做到能人眼识别的同时也能让机器识别其字符码，并且尽可能具有标准化和通用化。

1.2.2 图像处理

图像处理应当具有可展示性，并且能够具备不同的处理进度和算法展示，包括灰度化、模糊化、二值化等算法处理，在处理识别的过程中应当展示出来，以便供提供处理进展数据和方便人员进行分析，图像处理还应当在之后的识别同时在图形界面中逐张展示，提供即时性的处理和让人员进行分析。

1.2.3 神经网络训练和识别

神经网络应当符合现阶段的技术发展，在图像识别上应当使用有比较优良技术价值的算法，例如 CNN 内部的卷积层、池化层等，并在确保验证码数据足够庞大的同时能够让神经

网络具有长时间不间断的训练，以便能确保能达到良好的准确率。而神经网络的识别应体现在模型的识别上，具体为神经网络在训练后输出带有接口的模型，并且模型能够在后期进行函数调用。

1.2.4 图形界面—交互特性需求

图形界面是用户与程序交互的主要方式，这能有效降低用户操作程序的成本。并且能直观展现程序处理结果和过程。图形界面因具有打开数据，输入数据，处理数据，展现数据等核心功能以及一些别的辅助性功能。在用户输入图片，并开始测试之后，图形界面应当调用并显示处理过程中的图片和处理结果，如果作为网络接口，可以在不要求图形界面显示处理结果的同时要通过通信端返回结果。

1.3 非功能性需求

1.3.1 输出监控日志以及处理时间计算

程序处理以及模型识别过程中，可能需要逐模块分析其运作形式，因此需要具备监控日志的功能，在需要完成识别和测试的时候，通过查看日志可以了解程序运行的行为，运行方式，以及判断可能出现的错误等。

需要展示程序运行时间，以便能判断神经网络训练出来模型的优良好坏，模型的好坏不仅取决于其准确率，还取决于运行的效率，程序也同样如此。程序/模型处理时间应该展现在图形界面上，方便进行分析。作为程序，运行时间应当具备：响应时间快，能及时反馈结果和过程

1.3.2 通讯传输数据

通讯传输数据可以作为测试性需求进行开发，将程序应用到实际场合中需要程序将识别出来的结果进行 C/S 或者 B/S 传输。这样可以进行远程自主识别和结果返回。

1.4 设计定位

鉴于开发平台为微软 windows10 系统，因此运行平台也必须为 windows10 后者 windows7 系列平台，以便能展现最佳效果。程序编程语言为 python，并指定为主要编程语言，能在 python==3.7 的环境中运行，由于开发 IDE 为 PyCharm，因此推荐运行环境也为 PyCharm，并导入附录中的第三方库，即可有效运行。

运行功能定位为验证码自动识别，包括手动识别，挂载在服务器上识别以及作为测试功能的自动抓取识别和传输结果。

该程序平台主要通过极限编程的模式进行开发，主体或者说唯一编程语言为 python，主要编程平台框架为 Anaconda，人工智能训练框架为 Tensorflow2.1 版本，并采用 CNN 卷积神经网络算法进行搭建。

第二章 概要设计

程序主要采用结构化设计，通过仅有的参数传递尽可能降低每个模块之间的耦合，并尽可能将功能划分成模块以提高模块内聚性，各个模块以及其中的关系如图 2-1 所示：

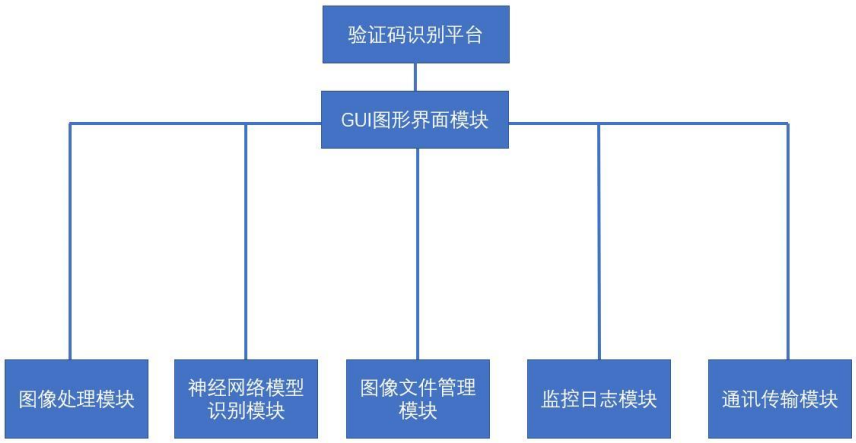


图 2-1 模块结构图

图像处理模块针对验证码图像进行各种处理，包括灰度化、模糊化、二值化，梯度化等，神经网络识别模块针对处理后的图像进行识别。图像文件管理模块针对用户打开验证码，和随机挑选验证码进行管理和调用，通讯传输模块是针对识别结果进行本地传送通讯的测试模块。

打开图形界面后，用户可以打开验证码图像导入到程序中，开始测试之后，用户通过打开主函数直接调用 GUI 图形界面，而图形界面本身也有三个模块组成，各自通过函数调用组成 GUI。进入 GUI 后可以在菜单栏打开图片、打开随机图片、进行测试、查询帮助和版本。然后再进行测试中，再主函数内部进行各个模块的调用，依次为处理图像，图像识别，生成结果，计算准确率以及计算运行时间。并且可以在生成结果之后查看监控日志，分析程序处理过程，模块关系功能如图 2-2 所示。

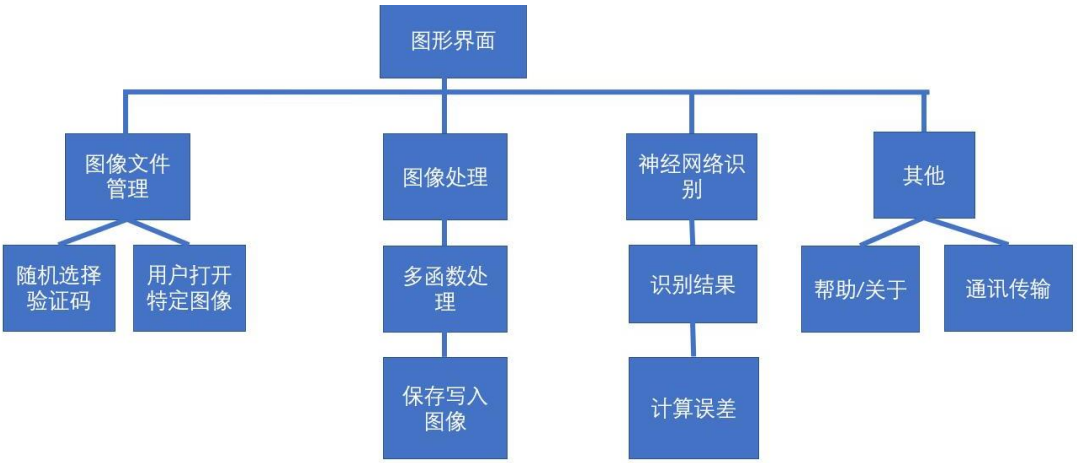


图 2-2 系统软件结构

第三章 详细设计

3.1 主要功能设计

3.1.1 验证码生成

验证码生成模块主要是生成验证码数据的，既是作为神经网络训练的标准数据源，也是测试神经网络模型的数据集。验证码的生成方式采用 Python 官方提供的 Captcha 验证码库，并在此基础上，调用其内部算法进行生成。具体编程流程如图 3-1 所示：

图 3-1 编程流程

在生成随机字符列表中，主要采用调用第三方随机库函数实现，在已经定义好的函数内调用随机库，让其在之前已经定义好的设定的验证码字符数据中进行随机选择，生成 4 位字符并以列表方式展现，而到了生成随机验证码函数的时候，函数就将列表转换成字符串，再将字符串传入验证码生成库进行图片化处理。主要两个函数的部分代码（包含可理解的逻辑注释）如图 3-2、图 3-3 所示：

```
# 获取四个随机字符
# def random_captcha_text(char_set=number,captcha_size=4):
def random_captcha_text(char_set=number, captcha_size=4):
    # char_set为预定的字符数组，captcha_size为字符数量
    captcha_text = [] #验证码列表
    for i in range(captcha_size):
        # 随机选择
        c = random.choice(char_set)
        # 加入验证码列表
        captcha_text.append(c)
    return captcha_text
```

图 3-2 获取随机字符并导入到列表

```python
# 生成字符对应的验证码
def gen_captcha_text_and_image():
    image = ImageCaptcha()
    # 获取随机生成的验证码
    captcha_text = random_captcha_text()
    # 把验证码列表转换为字符串
    captcha_text = ''.join(captcha_text)
    # 生成验证码
    image.write(captcha_text, './image/' + captcha_text + '.png')
```

图 3-3 生成字符对应的验证码

3.1.2 图像处理

图像处理等同于数据预处理，是深度学习最重要的步骤之一，神经网络训练的成本、效率、结果很大程度上取决于前期数据预处理的效果。如果数据处理得好，就能有效去除大部分图像噪点和干扰线，从而让图片的特征更好的体现出来，因而能非常有效降低神经网络的训练成本，减少出错率，提高效率。图像处理流程如图 3-4 所示：

图 3-4 图像处理流程

图像处理包含几大处理算法：灰度化、模糊化、二值化、梯度化、计算轮廓、形态学处理的的膨胀化、计算列像素数量等。这几项算法根据滤波理论（均值、中值、各项异性、高斯）等依次进行排序处理，而在进行前期滤波灰度、高斯模糊之后，进行二值化的时候就能有效解决大部分噪点的问题，这非常能有效提取出验证码的特征。但为了保险起见，在二值化之后还是有可能存在非常大的、无用的噪点，这时候就进行膨胀，将图像特征细瘦化，这也能有效将其他大部分噪点给膨胀去掉。而上述几种，基本都是以滤波计算为基础，通过查

阅几十种 OpenCV 库内部的算法之后，确定下来效果比较显著的算法模型。部分重要代码以及模型调用算法如下图所示。

读取原图图像代码如图 3-5 到所示

```
#opencv读取原图片
def img_input(img_open_path):  #参数为图像路径

    img = cv2.imread(img_open_path,1)#cv读入图像路径的图像，1为读取全像素通道颜色
    print(img.shape)
    #cv2.imshow('image', img)
    #cv2.waitKey(5000)
    #显示原始图片
    return img
```

如图 3-5 读取原图像图

灰度与模糊处理代码如图 3-1-6 所示

```
#灰度与模糊处理
def gray_blurred(img):  #参数为原图像
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #cv读取原图像，采用cv灰度化算法
    blurred = cv2.GaussianBlur(gray, (9, 9),0) #将灰度化的图像进行模糊化,0为灰白像素通道

    cv2.imwrite('.\\dilateimage\\'+'gray.png', gray) #写入进路径文件
    return gray,blurred   #返回灰度化、模糊化图像
```

图 3-6 灰度化以及模糊化

模糊图像二值化代码如图 3-1-7 所示

```
#模糊图像二值化
def getbinary(blurred): #参数为模糊化图像
    ret, binary = cv2.threshold(blurred, 216, 225, cv2.THRESH_BINARY | cv2.THRESH_OTSU)
    #将模糊化的灰度图像进行二值化，216为判断阈值，225是更改像素，后面为cv2算法
    print("threshold value: %s" % ret)
    cv2.imwrite('.\\dilateimage\\'+'binary.png', binary)
    return binary
```

图 3-7 图像二值化

膨胀华代码如图 3-1-8 所示

```
#膨胀化
def getdilate(binary):
    kernel = np.ones((4, 4), dtype=np.uint8) #返回全是1的n维数组
    dilate = cv2.dilate(binary, kernel, 1) # 1:迭代次数，也就是执行几次膨胀操作
    cv2.imwrite('.\\dilateimage\\' + 'dilate.png', dilate)
    return dilate
```

图 3-8 图像形态学膨胀

对于定位图像分割位置，以及切割图象，由于代码过长，只截取部分重要代码以及分割后的结果图像。

判断图像分割位置：主体思想为以 X 轴（横轴）为基准先从轴最左边开始遍历垂直分割后图像的每一列的像素，如果该列有 0 个黑像素点，这判断该列为像素分割终点，然后计算从最左端的 0 点开始到结束分割点中间的像素总和，判断总和与噪点阈值的大小，如果小于阈值，则判断为噪点或空值，如果大于阈值，则判断为字符。

在判断位置之后，需要转换像素分割的左右点，届时，就把前一个像素分割终点作为新的像素起始点，而程序也将继续遍历下一行的像素，一共会返回 8 个像素分割点位置。

垂直投影像素代码如图 3-9 所示。

```
#垂直投影像素

def getVProjection(dilate):
    # 将image图像转为黑白二值图，ret接收当前的阈值，thresh1接收输出的二值图
    ret, thresh1 = cv2.threshold(dilate, 127, 255, cv2.THRESH_BINARY)
    (h, w) = thresh1.shape  # 返回高和宽
    a = [0 for z in range(0, w)]  # a = [0,0,0,...,0,0]初始化一个长度为w的数组，用于记录每一列的黑点个数
    # 记录每一列的波峰
    for j in range(0, w):  # 遍历一列
        for i in range(0, h):  # 遍历一行
            if thresh1[i, j] == 0:  # 如果该点为黑点
                a[j] += 1  # 该列的计数器加一计数
                thresh1[i, j] = 255  # 记录完后将其变为白色

    for j in range(0, w):  # 遍历每一列
        for i in range((h - a[j]), h):  # 从该列应该变黑的最顶部的点开始向最底部涂黑
            thresh1[i, j] = 0  # 涂黑
    return thresh1
```

图 3-9 垂直投影像素

记录每列像素量以及位置如图 3-10 所示

```
    (h, w) = dilate.shape  # 返回高和宽
    a = [0 for z in range(0, w)] # a = [0,0,0,...,0,0]初始化一个长度为w的数组，用于记录每一列的黑点个数'
    b = [0 for z in range(0,w)]  # a = [0,0,0,...,0,0]初始化一个长度为w的数组，用于记录每一列的黑点个数'
    pixelnum=0
    position= {'left': 0, 'right': 0,'value':0}
    location= {'left':0,'right':0}
    print('---------以下分割为像素位置的输出---------')
    # 记录每一列的波峰
    for j in range(0, w):  # 遍历一列
        for i in range(0, h):  # 遍历一行
            pixelnum=0
            if thresh1[i, j] == 0:  # 如果该点为黑点
                a[j] += 1  # 该列的计数器加一计数
        if a[j]==0: #如果该列像素点为黑点的数量为0，则判定为空列
            position['right']=j  #记录该空列的为右区间点
            location['right']=j
            for k in range(position['left'],position['right']):#从该两个列中进行像素量计算  #遍历一列
                for n in range(0,h):    #遍历一行
                    if thresh1[n,k]==0:   #如果该点为黑点
                        b[k]+=1       #该列的黑点数量+1
                        pixelnum = b[k]+pixelnum  #计算列区间内所有像素点的总和
                        position['value']=pixelnum
            if pixelnum<=10:
                continue
```

如图 3-10 记录每列像素量以及位置

分割图像：主要是想就是将 8 个分割点转换双值列表，每个列表有两个值，分别对应每

个区块的分割起始点和分割终点。然后进行循环分割。

裁剪图像代码如图 3-11 所示

```
#裁剪图像
def confirmpoint(dilate):
    #dilate=cv2.imread(.\\dilateimage\\+'dilate.png',0)
    print('------以下为分割图像的输出------')
    count = len(open(r"location.txt", 'r').readlines())  #读取txt文件中的数据有多少行
    print('原始行数:',count)
    #with open("location.txt", "r") as f:
    #    data = f.readlines()
    #    print(data)
    with open('location.txt') as jaf:
        listdata = jaf.readline()  # 读取数据行
        newlistdata = listdata.strip().split(' ')  # 将数据转换为字符列表
        print('字符型列表数据输出:',newlistdata)   #输出字符列表
    intdata = [int(x) for x in newlistdata ]  #转换字符列表为整数列表
    print('整数型列表数据输出:',intdata)   #输出整数列表
```

图 3-11 裁剪图像

但是毕竟会极有可能存在像素值大于阈值的噪点，或者字符连在一起，甚至重叠在一起的情况，这样的话会很容易出现切割错误。这也就是垂直投影以及切割图像、KNN 近邻算法会被放弃而转换成神经网络识别的原因。

而以上这些图像等，将会在测试报告章节那里做详细阐释以及会放出实际图像的切割情况。

在整体图像成功分割成单个字符图像后，后期还有一系列配套模块进行处理，包括扩充图像背景，统一图像尺寸，进一步二值化等。所示代码将刚分割出来的条状矩形图像填充白色背景并转换成正方形图像，如图 3-12 所示，并进行保存。所示代码将图像重新统一尺寸为 28 像素*28 像素如图 3-13 所示。

```
for i in range (file_number):
    image = cv2.imread('.\\cut_image\\%d.jpg'%i)  #读取图像
    top, bottom, left, right = (0, 0, 0, 0) #设定四个值
    # 获取图片尺寸：高和宽
    h, w, _ = image.shape
    # 对于长宽不等的图片，找到最长的一边
    longest_edge = max(h, w)
    # 计算短边需要增加多少像素宽度才能与长边等长(相当于padding，长边的padding为0，短边才会有padding)
    if h < longest_edge:
        dh = longest_edge - h #最长的边减去高
        top = dh // 2
        bottom = dh - top     #整个判断部分为确定该填充图像的大小
    elif w < longest_edge:
        dw = longest_edge - w #最长的边减去宽
        left = dw // 2
        right = dw - left
    else:
        pass  # pass是空语句，是为了保持程序结构的完整性。pass不做任何事情，一般用做占位语句。
    # RGB颜色
    BLACK = [225, 225, 225]
    # 给图片增加padding，使图片长、宽相等
    # top, bottom, left, right分别是各个边界的宽度，cv2.BORDER_CONSTANT是一种border type，表示用相同的颜色填充
    constant = cv2.copyMakeBorder(image, top, bottom, left, right, cv2.BORDER_CONSTANT, value=BLACK)
    # 调整图像大小并返回图像，目的是减少计算量和内存占用，提升训练速度
```

图 3-12 填充并修改尺寸

```
def convertjpg(jpgfile,outdir,width=28,height=28):
    #函数参数类型：图像文件名，图像路径，将图像转换成28*28大小的统一格式
    img=Image.open(jpgfile)  #第三方库打开图像文件
    try:
        new_img=img.resize((width,height),Image.BILINEAR)#重新统一图像尺寸
        new_img.save(os.path.join(outdir,os.path.basename(jpgfile)))#保存图像
    except Exception as e:
        print(e)
```

图 3-13 重新修改尺寸

3.1.3 神经网络训练以及识别

神经网络模型为程序的最核心部分，该部分的最主要功能是识别判断验证码字符，并输出结果，作为当前比较新的技术，程序采用 tensorflow2.0 版本内部的继承模型，并参照官方模板进行修改和测试。由于是图像识别，因此将着重采用卷积神经网络进行模型搭建，并且将进行多种激励函数的测试。

在进行神经网络训练的时候，程序将数据进行批次化传送训练，按照每批（batch）128 张，每个轮回（epoch）训练 4 批，一次（time）为 4 个轮回。并且在内部进行原生的验证码生成而不是传入之前生成的验证码，即每次训练都会拥有不同的验证码，验证码重复率比较低。以下为神经网络搭建以及训练的核心代码部分，包含注释。

（1）封装与修改批次

将验证码进行批次化封装，并返回文本标签与修改后的图像。每张图像都要使用 NumPy 进行数值矩阵化，然后进行修改尺寸和图像处理，并传出封装好的图像和标签，如图 3-14 所示。

```
def get_next_batch(batch_size=128): #每一批次包含128张图像
    #batch_x为numpy数值化候的图像，batch_y为该图像的字符标签
    batch_x = np.zeros([batch_size, IMAGE_HEIGHT, IMAGE_WIDTH, 1])
    batch_y = np.zeros([batch_size, MAX_CAPTCHA, CHAR_SET_LEN])

    def wrap_gen_captcha_text_and_image():
        while True:
            #随机生成验证码，与前面生成验证码部分相同
            text, image = gen_captcha_text_and_image(char_set=CHAR_SET)
            #规定验证码尺寸：高60，宽160，像素通道3，即为彩色
            if image.shape == (60, 160, 3):
                return text, image
```

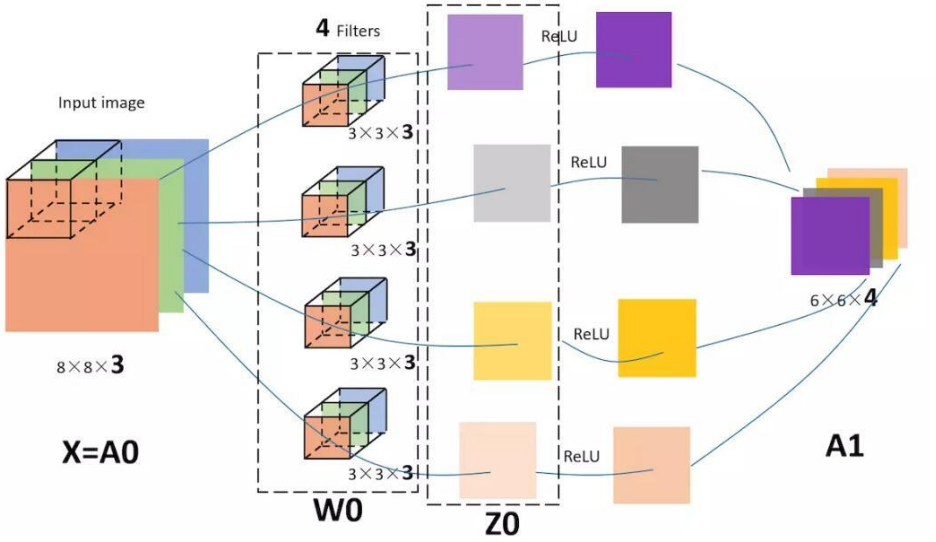
3-14 封装与修改批次

（2）构建神经网络模型

首先要初始化一个继承于官方内部模型的类，并在最开始的时候进行卷积化、线性矫正、池化等操作。这个卷积神经网络模型由卷积核来提取特征，通过池化层对显著特征进行提取，经过多次的堆叠，得到比较高级的特征，最后可以用分类器来分类。这是 CNN 模型的一个大概流程，其具体实现的结构是丰富多样的，但总的思想是统一的。

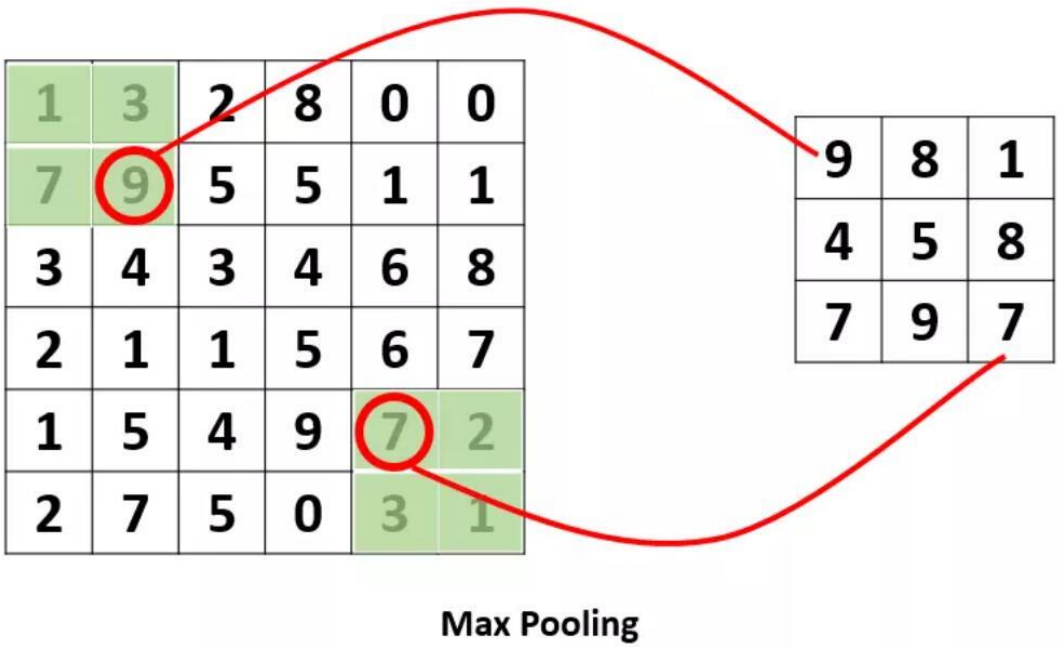
第一个卷积层，可能只是从原始图像像素中学习到一些边缘特征，但是到了第二个卷积层可以从这些边缘特征中探测到简单的形状特征，然后接下来的卷积层，就可以用这些简单

的形状特征探测到更高级的特征。以下为该作品模型内卷积层提取特征的过程，将三通道图像经过 3*3 的滤波器从而提取出新的特征层，如图 3-15 所示。



3-15 卷积层工作原理

之所以要在每个卷积层后面加入池化层，是因为池化层可以提取显著特征的同时降低模型的参数，从而降低模型的过拟合。因为只是提取了显著特征，而舍弃了不显著的信息，是的模型的参数减少了，从而一定程度上可以缓解过拟合的产生。在每层卷积层后面加入池化层这也就能尽可能防止卷积层过多而造成的过拟合问题。比如下面的 MaxPooling，采用了一个 2×2 的窗口，并取 stride=2，如图 3-16 所示。



3-16 池化层工作原理

池化层还具有局部不变性。所谓局部不变性，比如图像，就是图像经过简单的平移、旋转、尺度放缩，池化层在相同的位置依旧可以提取到相同的特征。包括平移不变性，旋转不变性，尺度不变性。这对类似验证码的字符旋转，形状变化等有很大的纠正性，能有效在每一层中提取出更正确的特征。但是局部不变性只能保证小幅度的变换，如果变换幅度较大，

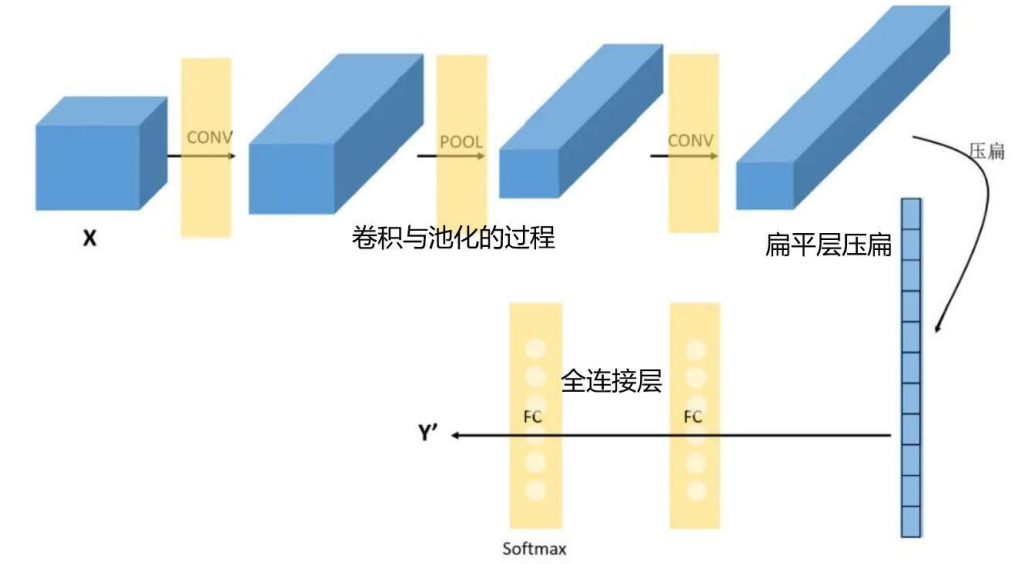
池化层并不能保证这种局部不变性。图 3-17 为模型的神经网络的核心代码。

```
def crack_captcha_cnn():
    model = tf.keras.Sequential()#实例化一个Sequential类，该类是继承于tensorflow内部的Model类；

    model.add(tf.keras.layers.Conv2D(32, (3, 3)))#具有32个卷积核的卷积层，过滤器大小3*3
    #普通二维卷积，常用于图像。参数个数 = 输入通道数× 卷积核尺寸(如3乘3)× 卷积核个数
    model.add(tf.keras.layers.PReLU()) #参数矫正线性单元，继承自tensorflow的层，与轴形状有关
    model.add(tf.keras.layers.MaxPool2D((2, 2), strides=2)) #2D图像池化层，池大小2*2，步长值为2

    model.add(tf.keras.layers.Conv2D(64, (5, 5)))#卷积核与过滤器持续增大，
    model.add(tf.keras.layers.PReLU())#同上
    model.add(tf.keras.layers.MaxPool2D((2, 2), strides=2))#同上

    model.add(tf.keras.layers.Conv2D(128, (5, 5)))#卷积核与过滤器持续增大，
    model.add(tf.keras.layers.PReLU())#同上
    model.add(tf.keras.layers.MaxPool2D((2, 2), strides=2))#同上

    model.add(tf.keras.layers.Flatten())  #压平层，用于将多维张量压成一维。即降维
    model.add(tf.keras.layers.Dense(MAX_CAPTCHA * CHAR_SET_LEN))
    #上面为密集连接层。参数个数 = 输入层特征数× 输出层特征数(weight)+ 输出层特征数(bias)
    model.add(tf.keras.layers.Reshape([MAX_CAPTCHA, CHAR_SET_LEN]))
    #上面为形状重塑层，改变输入张量的形状。
    model.add(tf.keras.layers.Softmax())#激活函数层，采用Softmax激活函数
    return model
```

图 3-17 神经网络核心代码

由于验证码图像的像素只有 160*60，本身是比较小的图像和数量级比较低的像素，因此，刚开始的卷积核不能太大，根据官方推荐的阶级，采用 32 做开始的卷积核，并且在后面几层不断增大卷积层数量，以方便提取特征。并且对于卷积神经网络而言，卷积核越小越有利于提取出特征，“小而深”是卷积神经网络有效发挥的特点。

在经过数次卷积和池化之后，我们 最后会先将多维的数据进行“扁平化”，也就是把 (图像高，图像宽，图像像素通道)的数据压缩成长度为 height × width × channel 的一维数组，这有利于与全连接层进行对接，降低后面神经网络的训练成本。而整体模型结构图也如下图 3-18 神经网络结构图所示。



3-18 神经网络结构

在数据进入全连接层之后，需要对其进行传统的激活函数处理，以保证权重和偏值的正确。该作品模型的激活函数采用 softmax 进行处理。该函数又称归一化指数函数。它是二分类函数 sigmoid 在多分类上的推广，目的是将多分类的结果以概率的形式展现出来。该类函数主要作用就是通过概率化将数据分类，即主要用于多分类问题，并且该函数在卷积神经网络中经常使用。

模型经过扁平化，标准神经网络全连接层之后，就能获得结果输出，模型的搭建等也基本结束。

（3）模型训练

在神经网络模型完成搭建之后，就可以开始着重进行模型训练，而模型的训练需要调用之前已经搭建好的神经网络模型以及数据封装模块，在训练过程中，则需要根据训练次数来保存模型，以方便在后期调用最新的训练模型，如图 3-19 模型训练所示。

```
def train():
    try:
        model = tf.keras.models.load_model(SAVE_PATH + 'model')#尝试载入上一个模型
    except Exception as e:
        print('#######Exception', e)
        model = crack_captcha_cnn()#如果没有模型（即第一次），就调用神经网络创建一个新的模型
    model.compile(optimizer='Adam', #优化器--adam
                  metrics=['accuracy'], #metrics: 列表，包含评估模型在训练和测试时的性能的指标，
                  loss='categorical_crossentropy')  #交叉熵损失函数
    for times in range(500000):  #设定训练次数500000次
        batch_x, batch_y = get_next_batch(512)  #传入上述的batch批次类，一共为128*4=512张
        print('times=', times, ' batch_x.shape=', batch_x.shape, ' batch_y.shape=', batch_y.shape)
        model.fit(batch_x, batch_y, epochs=4) #把两项数据传入模型，使其开始运作，epoch为类似batch的批次
        print("y预测=\n", np.argmax(model.predict(batch_x), axis=2))#模型对图像进行预测，沿着第2轴展开
        print("y实际=\n", np.argmax(batch_y, axis=2))#输出图像实际标签的值
        if 0 == times % 10:
            print("save model at times=", times)
            model.save(SAVE_PATH + '%d'%(times)) #根据训练次数保存模型
```

3-19 模型训练

通过使用 model.compile 函数定义编译的优化器、损失函数以及数据列表等，而 numpy.argmax()函数则作为索引函数找到最有可能的值，而内部的变量则为模型的预测函数。训练次数设定为 50 万次，本来预想进行为期差不多一个月的模型训练，但是其实在连续训练到第 2 天左右，平均准确率已经基本达到 75%以上，因此使用该模型做识别已经足够。每次在进行一次完整的模型训练后都会进行保存，下一轮训练的时候就调用上一次模型继续进行。

3.1.4 图形界面设计

图形界面是用户与程序交互的关键链接，作为图像识别平台，用户需要借助图形界面进行分析图像像素和图像。该程序通过调用 Tkinter 库模块进行搭建，以下为最初打开后以及经过处理的的识别的平台。

在启动程序里面，主要通过是先生成显示的图形界面，然后用户再通过调用各个模块进行程序交互，并且在图形界面里面，还具有各种不同的显示信息如图 3-20 所示。

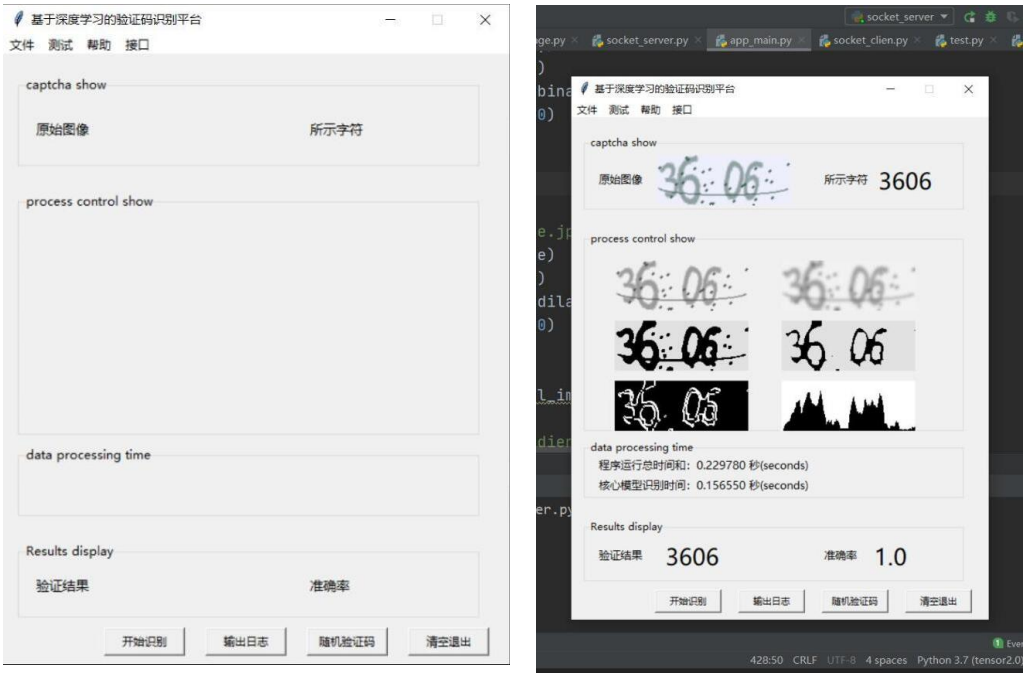


图 3-20 识别平台示例图

平台分有菜单栏、原始图像显示区、图像处理过程显示区、数据处理时间区以及结果显示区，在菜单栏部分，设计了与一般软件一样的文件打开栏，可以打开选定的验证码图片，在测试区，具备和界面下方按钮一样的功能：“开始识别”、“输出日志”、“随机验证码”。如果想查看版本或者帮助还可以在帮助界面进行查询，而接口部分可以进行测试功能，包括目前开发的 socket 通讯传输。

3.2 辅助功能设计

3.2.1 程序时间计算

计算程序运行时间是用于分析程序响应速度以及模型优化程度的功能，在前中期的模型搭建的优劣和模块耦合关系的复杂度等问题具有客观指导作用。设定起始时间和截止时间，并嵌入进总体运行过程和核心模型运行过程，并将四个参数传入时间计算函数，计算各自的差值，并输出到 GUI 图形界面（可在 GUI 图形界面的数据处理时间区域中看到输出结果）。时间计算大体流程架构如图 3-21 所示：

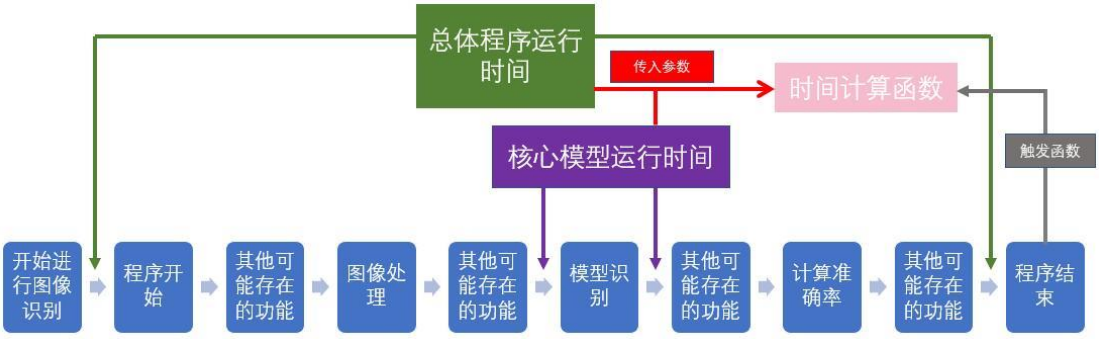


图 3-21 时间计算架构

3.2.2 传输测试

该程序中目前采用 socket 库进行通讯传输，目前也仅通过本地端口进行端口双程序传输。由于是单线程编程，发送端与接收端不能同时进行，因此，很多情况下在成功进行结果传输之后发送端进程会处于挂起（睡眠）状态，出现“未响应”的情况，而解决办法就是关闭接收端窗口，这样发送端就能重新被唤醒并继续执行新的图像识别任务。图传送端与接收端为传送结果如图 3-22 所示。

图 3-22 传送段与接收端示例图

传输过程中需要保持端口不被其他程序占用（程序代码固定端口为 27060），在进行测试的时候，先运行 socket_server.py 文件，这是模拟接收端的部分，在没有接收到结果的时候是没有任何反应的，同时在验证码识别的发送端，执行完图像识别显示结果后，在菜单栏上的“接口中”点击“测试 3”就会开始进行传输通讯，然后接收端就会弹出界面，显示接收的结果。具体架构流程图 3-23 所示：

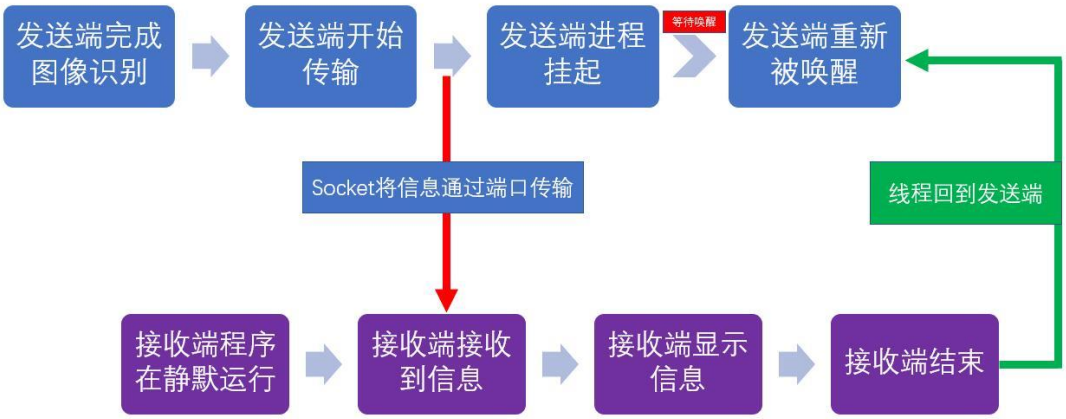


图 3-23 传输流程示意图

3.2.3 监控日志

监控日志可以作为分析程序运行的过程，并且可以判断程序出错的位置和可能出现错误的原因。查看监控日志可以在 GUI 图形界面中点击“输出日志”按钮，以消息框的方式查看，或者在程序目录中直接打开“log.txt”以 txt 形式查看。监控日志示例图为查看监控日志的示例以及界面如图 3-24 所示。

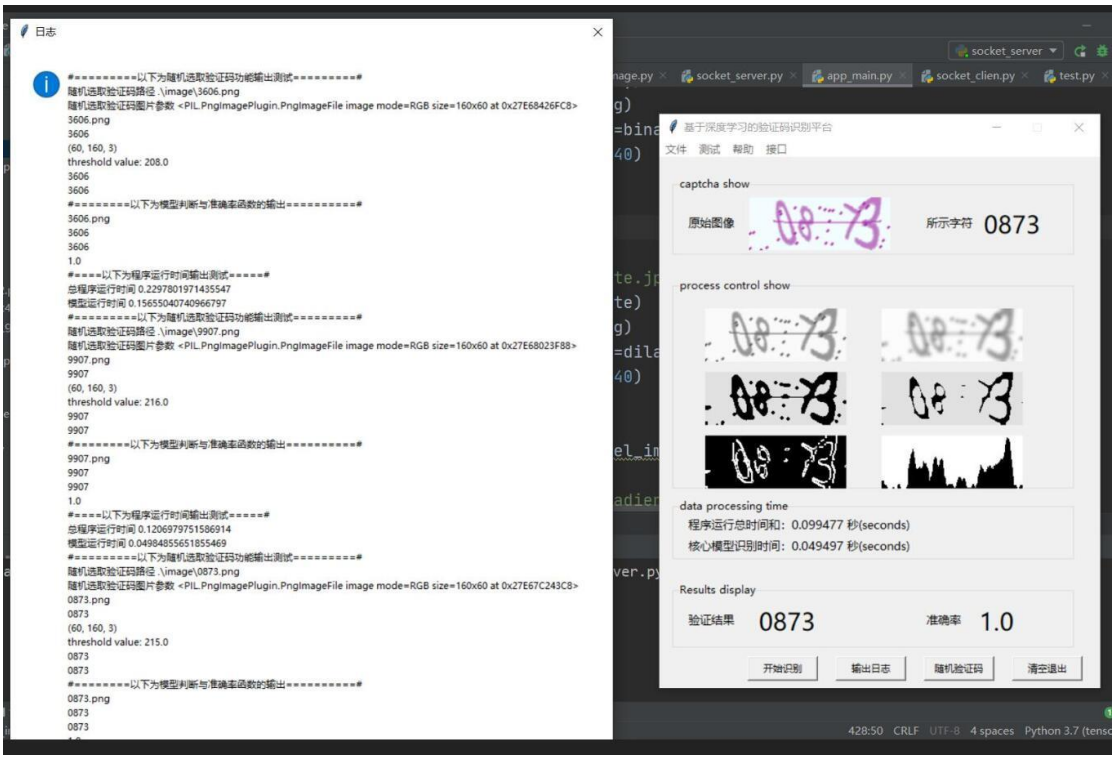
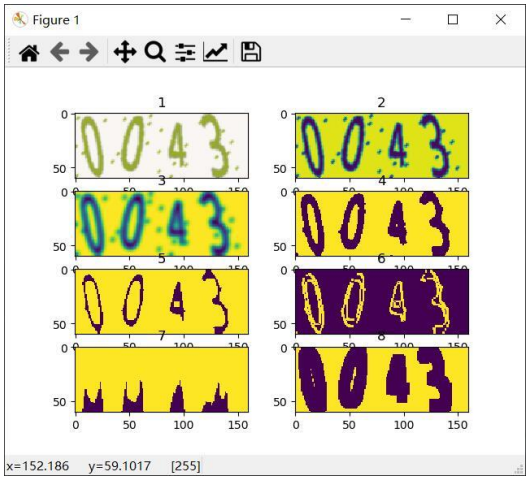


图 3-24 监控日志示例图

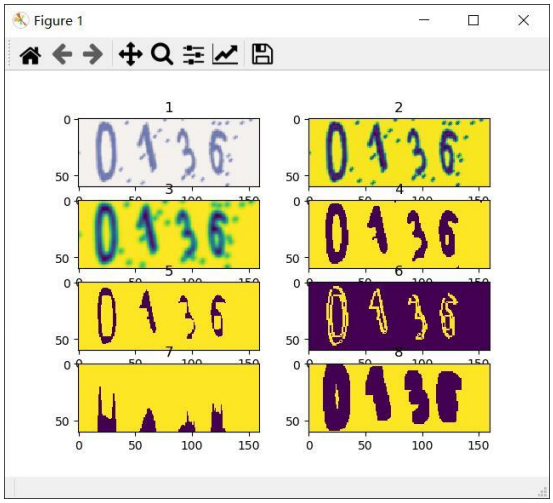
第四章 测试报告

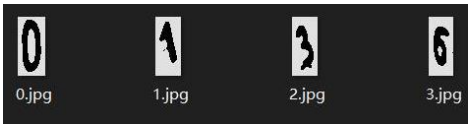
4.1 各项图像处理测试

测试所显示的图片采用 matplotlib 模块显示，matplotlib 模块与 OpenCV 模块不同，它所计算的灰白、模糊等一系列图像均不以灰、黑、白的形式展示，而是以黄、紫色调为主的色调代替。在以下的图片测试结果中，左图依次展示原图像、灰度化、模糊化、二值化、形态学膨胀化、梯度轮廓、垂直像素、腐蚀化，右图显示垂直投影后的调用自己写的图像切割函数所切割出的图像， 如下图所示。

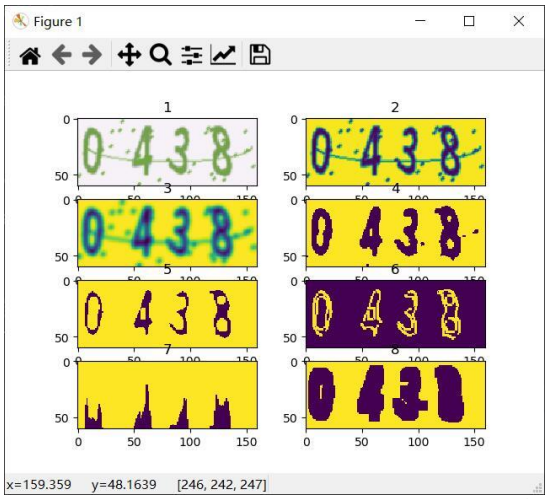


验证码为 0043 的图像

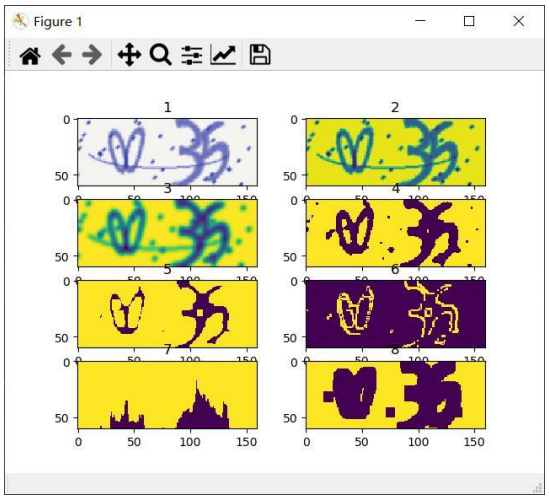




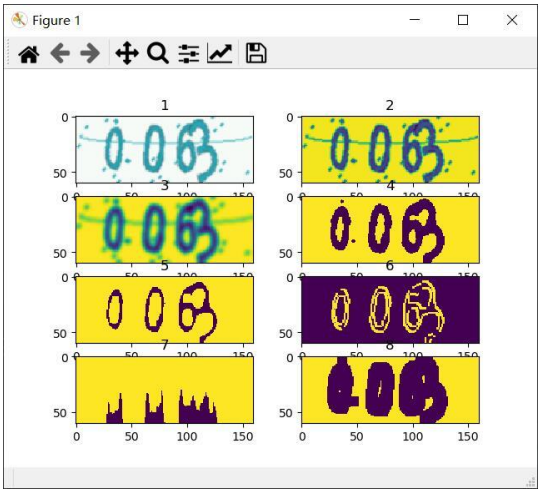
验证码为 0136 的图像



验证码为 0438 的图像



验证码为 0035 的图像



验证码为 0063 的图像

在上面的测试结果中，可以在前 3 张找到很明显的规律，就是每个字符都是分开的，具有独立性，并且形状还比较端正，即便是旋转，幅度也不大。这就非常有利于垂直像素投影和后期程序分割图像。

但是后面两张有很明显的不同，后面两张：0035、0063 的字符有粘连现象，所以在垂直投影像素算法中，像素块很难被识别到底哪一块属于那个字符，甚至是到底有多少个字符。而又由于每个字符的高度，位置基本一样，因此也无法采取水平像素投影和分割，这就导致了垂直投影像素算法陷入瓶颈。一般而言，这类算法都是适用于不粘连，形态端正的切割对象，例如：车牌号，门牌号，书面打印的字符等，但是在验证码中该类算法的发挥就非常局限，并且也无法让分割算法正确运行，进行分割。因此在验证码识别中，必须得放弃这类算法，转而使用比较通用的，具有黑盒计算的神经网络模型。这也就是为什么上述所说该类算法和函数仅作保留的原因。

由于这些图像处理算法本身具有独特性和唯一性，因此在图像处理的测试过程中并不存在太多的修正过程，而只存在舍弃过程。处理本身就是根据去噪的程度和方式进行，而且一般情况下处理出来的结果本身也比较好。具有各自的独特性。在去噪方面，一般在灰度、模糊化，并进行二值化之后，就已经去除大部分噪点和干扰线，如果还存在像素占比比较大的噪点，通过膨胀化也能有效去除相当一部分。

倘若进行非神经网络的机器学习，我们还要将切割出来的图像进行重塑和统一化，这就需要程序将图像进行背景填白和修改尺寸，以下为切割后的图像进行填白和修改尺寸后的测试结果。但需要注意，这部分步骤，并不是这个程序必须要做的过程，因为该程序神经网络模型是处理整张验证码图像而不是切割后的图像，因此该部分步骤只是作为一种备用方案进行保留和分析。

以验证码图像 0136 为测试例子，进行填白，左图为刚切割出来的原图，右图为填白后的图像（该函数为程序目录“resize_img_get3.py”文件）：

可见，尺寸不一的切割图象，经过填白，已经变成了统一的 60*60 的 jpg 图像

但是有时候也会因为一些机器学习算法要求进行图像的尺寸修改，因此还很可能需要多一道工序，即再次修改图像尺寸，继续以 0136 为例子，将 60*60 的图像修改为 28*28 的图像，以便能符合部分机器学习算法（如 KNN 或者支持向量机）的要求（该函数为程序目录“image_packing_get4.py”文件）。图 4-1-1 为刚切割出来的原图，图 4-1-2 位填白后的图像，图 4-1-3 位将 60*60 的图像修改为 28*28 的图像。

图 4-1 刚切割出来的原图

图 4-2 填白后的图像

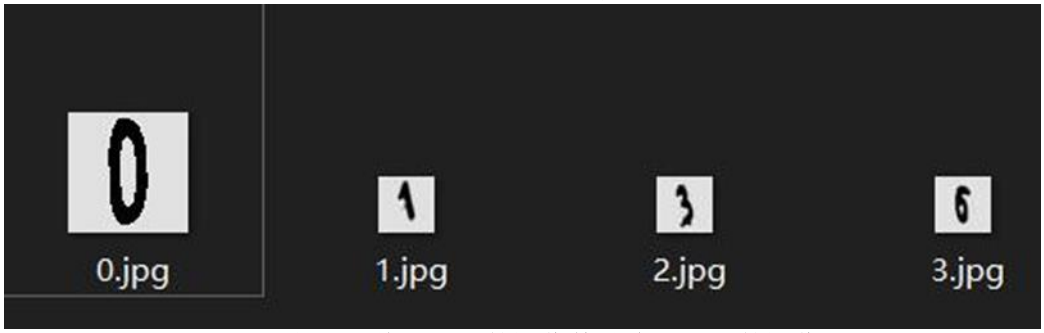


图 4-3 将 60*60 的图像修改为 28*28 的图像

4.2 神经网络模型测试

目前的神经网络模型仅保留了最后一部分模型的测试，因此，训练和测试均围绕这最新的神经网络模型进行展开。根据控制台的输出结果，通过截取了几部分进行分析，同时也将大部分的控制台输出数据进行整合，并进行总结，下图 4-4 到 4-7。

```
save model at times= 1230
times= 1231  batch_x.shape= (512, 60, 160, 1)  batch_y.shape= (512, 4, 62)
Train on 512 samples
Epoch 1/4

 32/512 [>.............................] - ETA: 18s - loss: 1.5175 - accuracy: 0.7344
 64/512 [==>...........................] - ETA: 18s - loss: 1.1185 - accuracy: 0.7969
 96/512 [====>.........................] - ETA: 16s - loss: 1.1241 - accuracy: 0.7917
128/512 [======>.......................] - ETA: 15s - loss: 1.1541 - accuracy: 0.7715
160/512 [========>.....................] - ETA: 14s - loss: 1.1349 - accuracy: 0.7625
192/512 [==========>...................] - ETA: 13s - loss: 1.1078 - accuracy: 0.7669
224/512 [============>.................] - ETA: 13s - loss: 1.1107 - accuracy: 0.7667
256/512 [==============>...............] - ETA: 11s - loss: 1.0983 - accuracy: 0.7725
288/512 [================>.............] - ETA: 10s - loss: 1.0751 - accuracy: 0.7778
320/512 [==================>...........] - ETA: 8s - loss: 1.0877 - accuracy: 0.7727
352/512 [====================>.........] - ETA: 7s - loss: 1.0996 - accuracy: 0.7692
384/512 [======================>.......] - ETA: 5s - loss: 1.0890 - accuracy: 0.7689
416/512 [========================>.....] - ETA: 4s - loss: 1.0660 - accuracy: 0.7710
448/512 [==========================>...] - ETA: 2s - loss: 1.1022 - accuracy: 0.7645
480/512 [============================>.] - ETA: 1s - loss: 1.0733 - accuracy: 0.7651
512/512 [==============================] - 22s 44ms/sample - loss: 1.0854 - accuracy: 0.7617
Epoch 2/4
```

4-4 第一部分实际测试

第一张图为训练到第 1231 次的时候，第一轮的批次的训练情况，在前 32 张时，损失函数的值为 1.51，而到了第 512 张后，损失函数的值为 1.07，可见在第一批的时候损失函数的值已经下降了近 0.5，这也是前 1000 多次训练得到的急速收敛的能力。

```
times= 1327  batch_x.shape= (512, 60, 160, 1)  batch_y.shape= (512, 4, 62)
Train on 512 samples
Epoch 1/4

 32/512 [>.............................] - ETA: 19s - loss: 1.2639 - accuracy: 0.8438
 64/512 [==>...........................] - ETA: 18s - loss: 1.1383 - accuracy: 0.8320
 96/512 [====>.........................] - ETA: 17s - loss: 1.0968 - accuracy: 0.8099
128/512 [======>.......................] - ETA: 16s - loss: 1.1343 - accuracy: 0.7676
160/512 [========>.....................] - ETA: 16s - loss: 1.0600 - accuracy: 0.7781
192/512 [==========>...................] - ETA: 14s - loss: 1.0015 - accuracy: 0.7865
224/512 [============>.................] - ETA: 13s - loss: 1.0560 - accuracy: 0.7958
256/512 [==============>...............] - ETA: 11s - loss: 1.0628 - accuracy: 0.7979
288/512 [================>.............] - ETA: 9s - loss: 1.1012 - accuracy: 0.7951
320/512 [==================>...........] - ETA: 8s - loss: 1.0550 - accuracy: 0.8000
352/512 [====================>.........] - ETA: 7s - loss: 1.0318 - accuracy: 0.8011
384/512 [======================>.......] - ETA: 5s - loss: 1.0290 - accuracy: 0.7949
416/512 [========================>.....] - ETA: 4s - loss: 1.0050 - accuracy: 0.7963
448/512 [==========================>...] - ETA: 2s - loss: 0.9790 - accuracy: 0.7997
480/512 [============================>.] - ETA: 1s - loss: 0.9765 - accuracy: 0.7958
512/512 [==============================] - 23s 44ms/sample - loss: 0.9807 - accuracy: 0.7959
Epoch 2/4
```

4-5 第二部分实际测试

第二张图为训练到第 1327 次的时候，第一轮的批次的训练情况，在前 32 张中，损失函数的值为 1.26，直到第一轮结束的时候，损失函数的值为 0.97。

```
times= 1409  batch_x.shape= (512, 60, 160, 1)  batch_y.shape= (512, 4, 62)
Train on 512 samples
Epoch 1/4

 32/512 [>.............................] - ETA: 21s - loss: 1.3603 - accuracy: 0.7500
 64/512 [==>...........................] - ETA: 20s - loss: 1.3617 - accuracy: 0.7734
 96/512 [====>.........................] - ETA: 18s - loss: 1.2284 - accuracy: 0.7917
128/512 [======>.......................] - ETA: 17s - loss: 1.1261 - accuracy: 0.7969
160/512 [========>.....................] - ETA: 16s - loss: 1.1505 - accuracy: 0.7719
192/512 [==========>...................] - ETA: 16s - loss: 1.0575 - accuracy: 0.7852
224/512 [============>.................] - ETA: 14s - loss: 1.0701 - accuracy: 0.7824
256/512 [==============>...............] - ETA: 12s - loss: 1.0276 - accuracy: 0.7861
288/512 [===============>..............] - ETA: 10s - loss: 1.0022 - accuracy: 0.7856
320/512 [=================>............] - ETA: 9s - loss: 0.9700 - accuracy: 0.7883
352/512 [===================>..........] - ETA: 7s - loss: 0.9544 - accuracy: 0.7912
384/512 [=====================>........] - ETA: 6s - loss: 0.9368 - accuracy: 0.7917
416/512 [=======================>......] - ETA: 4s - loss: 0.9441 - accuracy: 0.7891
448/512 [=========================>....] - ETA: 2s - loss: 0.9280 - accuracy: 0.7907
480/512 [===========================>..] - ETA: 1s - loss: 0.9279 - accuracy: 0.7885
512/512 [==============================] - 24s 47ms/sample - loss: 0.9006 - accuracy: 0.7896
Epoch 2/4
```

4-6 第三部分实际测试

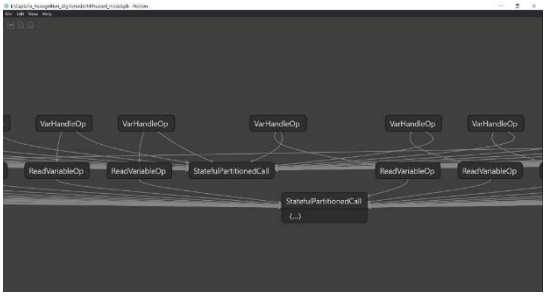
第三张图为训练到第 1409 次的时候，第一轮的批次的训练情况，前 32 张中，损失函数的值为 1.36，直到第一轮结束的时候，损失函数的值为 0.92。

第 1409 次的时候，可以看出，前 32 张的损失函数比第 1327 的时候要大，看似好像是训练到了极限或者说模型问题。其实原因有很多，由于是采用分阶段模型保存和调用的方案，每次保存的模型只进行了四个四批（512*4）的数据训练，因此权重、偏值变化等都有不同程度大小的过拟合，导致后面有些新的验证码就会出现较多的失误。从横向比较来看，其实越往后面，模型的正确收敛速度就越快，损失函数的数值在整体上也在不断减少，从每轮结束后的第一张的 1.07，第二张的 0.97 到第三张的 0.92，间接证实了每个阶段的新模型都在前一个阶段模型的基础上不断收敛。即使其中有一些次数的训练的损失函数值都比前面次数的大，也可以说是误差导向或者偶然性，这些并不能决定整体收敛的走向。

在前中期的时候损失函数先急速增长，然后急速下降，这是神经网络试错的结果，试错结束后，权重和偏值都在不断接近收敛状态。

到后期，基本处于波动性下降的状态，由于整个模型的收敛也逐渐到达极限，因此依旧会持续不断的产生大量波动，这些波动就是在全局最优化之间的来回，处于未拟合与过拟合的状态。在训练基本完成后，该神经网络模型的网络结构图如下图所示，由于结构为横向对称展开，因此分开截图，按照顺序，下图 4-7，4-8 依次为模型的边缘结构，中间结构。

4-7 边缘结构

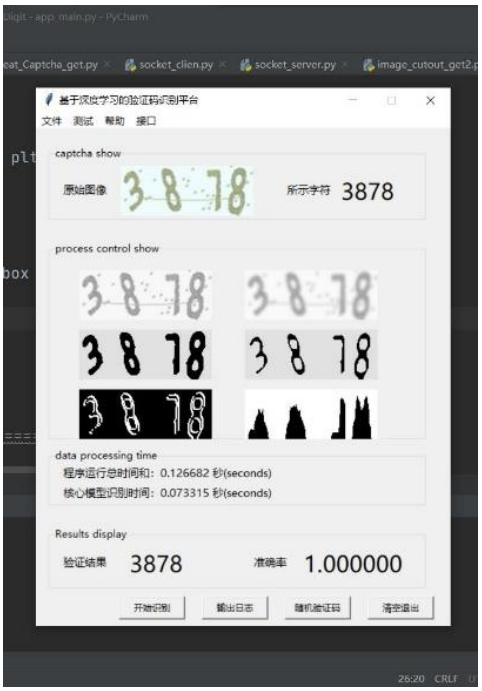


4-8 中间结构

4.3 整体程序测试

以下 4-9 到 4-14 为实际整体程序运行的测试图片：

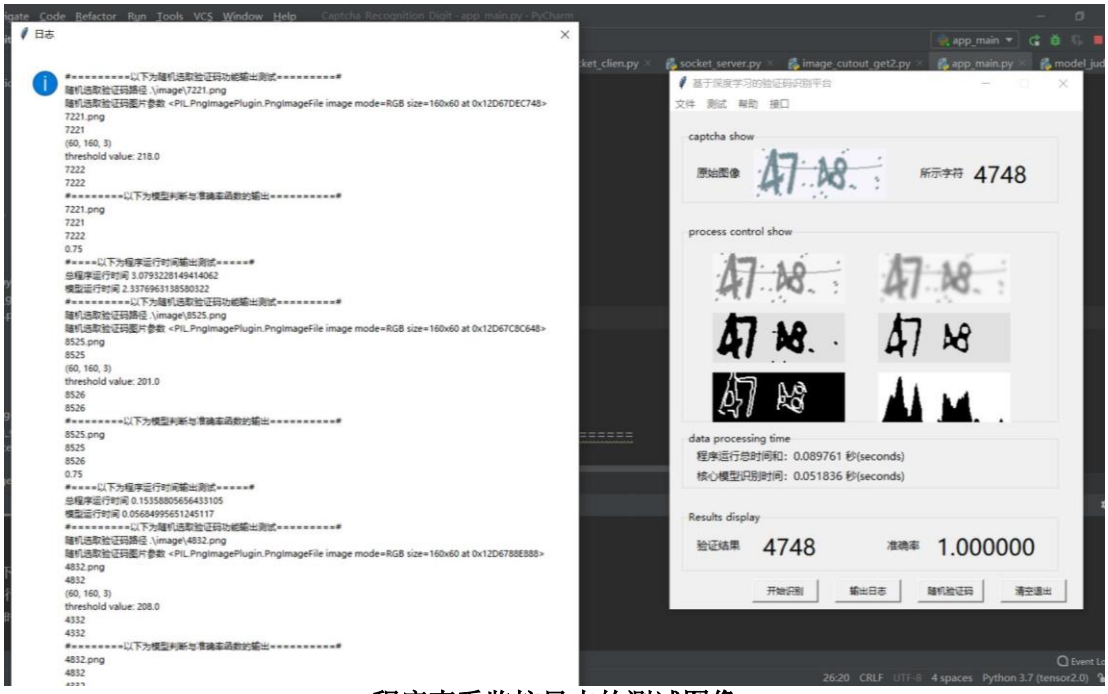
4-9 程序识别的测试图像



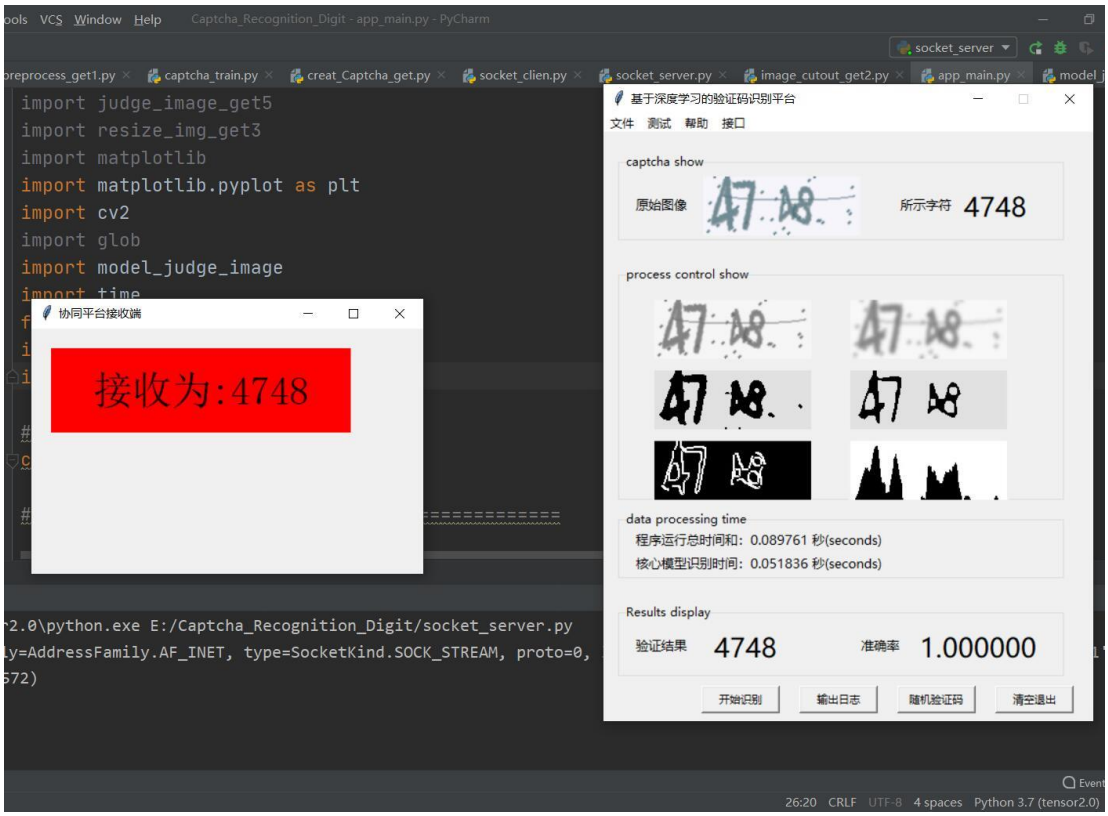
4-10 程序识别的测试图像

4-11 程序识别的测试图像

4-12 程序识别的测试图像



4-13 程序查看监控日志的测试图像



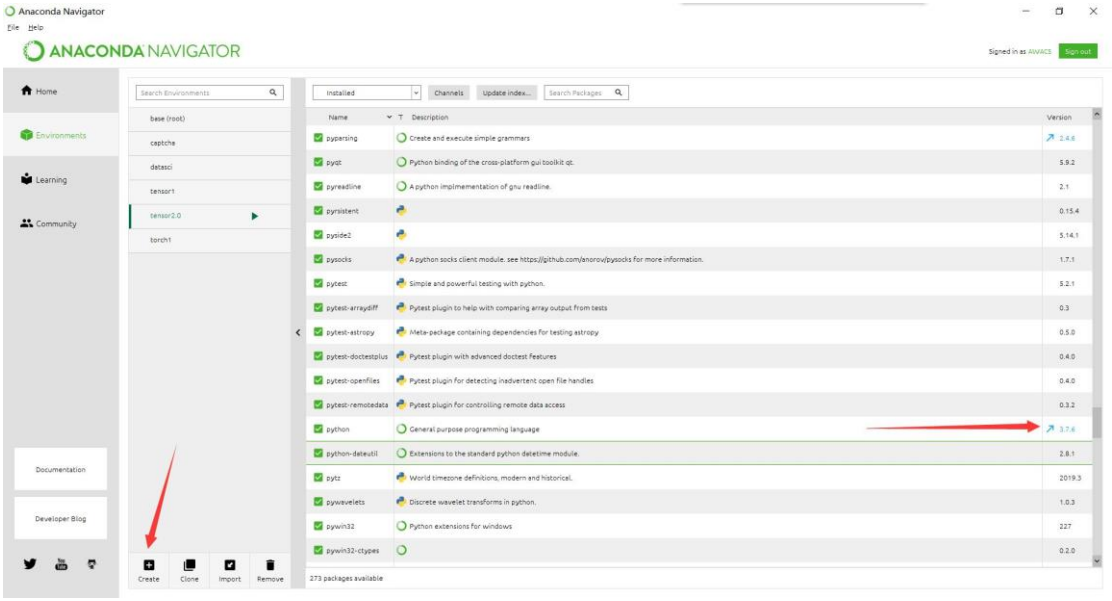
4-14 程序进行通讯传输的测试图像

第五章 安装及使用

5.1 开发环境安装

（1）安装 Anaconda(下载地址：https://www.anaconda.com/products/individual)。

（2）在 Anaconda 内部创建以 python==3.7 为基础的新环境，如图 5-0 安装 Anaconda。



5-0 安装 Anaconda

（3）在 anaconda 建立的新环境中安装以下第三方库：Tensorflow==2.0 以及以上、pillow、matplotlib、captcha、numpy、glob 等。

OpenCV 安装包已在目录里面，具体安装详情：

https://blog.csdn.net/iracer/article/details/80498732。

注意关键点：

使用 ctrl+r 打开 cmd 命令行窗口，使用 cd 命令进入上一步下载好 opencv_xxx.whl 文件所在目录

输入如下代码安装该文件：

```
pip install opencv_python-3.4.1+contrib-cp35-cp35m-win_amd64.whl
```

（4）安装 pycharm：https://www.jetbrains.com/pycharm/download/#section=windows 作为 IDE

（5）Pycharm 调用编译器：具体操作界面过程如下：在左上角点击 fiel 打开 settings 后找到“project”，然后再进入“project interpertre”,点击右上角齿轮的图标，显示 add，点击进去在“conda environment”中选择下方的 existing environment ，在路径找到 anaconda 的 python 执行文件，确定后即可。导入 conda 环境如图 5-1。

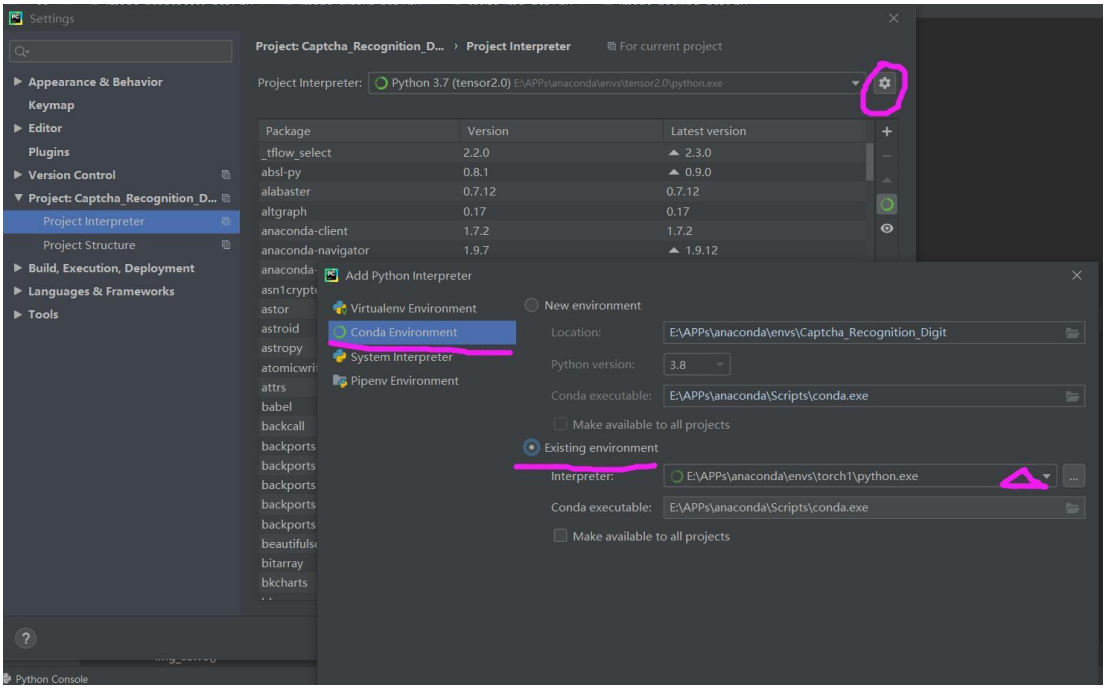


图 5-1 导入 conda 环境

5.2 程序安装以及使用

5.2.1 代码模式

安装完上述开发环境之后，在保持文件目录、文件名等不变的情况下，进入 IDE 内通过代码可以行 app_main.py，即可编译生成程序。

5.2.2 已编译模式

直接运行目录下的 app_main.exe 可执行文件，直接进行使用，不过由于 python 本身的局限性，完全由 python 编译出来的单线程程序启动的速度非常慢。正常情况下双击后会显示没有任何输出的控制台窗口，并且需要等待三分钟左右的时间才正式启动完毕，GUI 图形界面窗口才能显示出来。控制台窗口同时也是监控日子的输出窗口，可以实时查看程序的处理输出。不过需要注意，路径尽量不要有中文!! 不然可能会出错!

如果要进行通讯传输的测试，需要先运行“socket_server”文件，无论是代码模式下还是已编译模式下，若为代码模式下，需要先运行“socket_server.py”待其静默运行后，再进行“app_main.py”的运行，进行识别后再从接口菜单中点击“测试 3”触发 socket 传输。

如果是再已编译模式下，则先在同目录内运行“socket_server.exe”文件，待其弹出控制台，如果有有询问是否允许联网，请点击允许公网和私网下两项允许。届时则表示静默运行完毕，然后就可以打开“app_main.exe”执行文件，等待数分钟后弹出 GUI 界面， 然后进行验证码识别，进行识别后再从接口菜单中点击“测试 3”触发 socket 传输。

需要注意的是，前面说过，由于没有调用多线程，因此这两个程序是同时占用同一个进程，当接收端顺利接收结果，并展示的时候，识别平台作为发送端是处于“未响应”状态，这时候只需要关闭接收端的图形界面，识别平台就能被重新唤醒，能继续进行工作。图 5-2、图 5-3 分别为可执行程序运行图

图 5-2

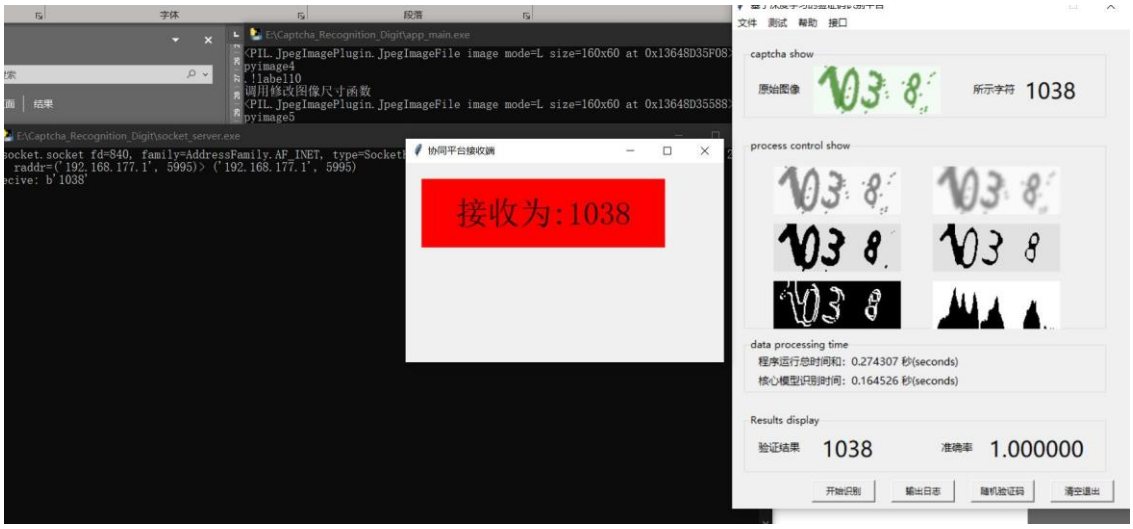


图 5-3

图 5-3 中先执行 socket_server.exe 文件后，再执行 app_main.exe 文件，进行点击接口-测试 3 后所展示的最终程序运行情况。

第六章 项目总结

6.1 面对的困难

刚开始是决定运用 KNN（K 近邻）算法实想验证码识别，后来实在在像素垂直投影以及分割上遇到很大的困难，才决定转型去深度学习神经网络做训练模型，尽管神经网络在制作上比 KNN 容易得太多，而且在识别上准确率也比 KNN 精确得多。只是当时想尝试传统的机器学习算法去进行验证码识别

6.2 水平提升

在进行神经网络模型搭建的时候，起初觉得将前面的卷积层和池化层做得越多越好，但是在后期的训练发现，过多的卷积层和池化层并不会对训练准确度有很大的提升，反而极大降低了模型训练的效率，消耗很多服务器系统资源。经过多次协调权衡之后才决定制作 3~5 层的卷积层和池化层作为特征提取。

6.3 升级演进

其实在制作该程序时，就已经考虑其升级版本，奈何时间不充足，来不及做，而升级版本就是通过完善的网络 socket 技术，将该验证码程序挂靠在服务器端，并执行自动获取验证码数据，自动识别，自动返回结果到本地或爬虫等程序上，并且作为验证码识别本身就拥有比较高的准确率，所以可以将模型和程序识别对象进行转化，例如车辆牌照识别、文字识别、甚至是人脸识别等。

6.4 商业推广

鉴于升级演进方面所做的描述，当前可以将程序作为服务器接口使用，后期完善后通过 socket 连接爬虫和本地应用程序，进行自动化操作，或者，将接口转售出去，获得利润。

6.5 心得体会

编程是软件开发过程中最基本、最底层的技艺，然而也是最重要的技艺。任何一个领域的专家都需要花费大量的时间来进行基本技艺的锻炼，木匠需要花费大量的时间来锻炼他们对各种工具的掌握，厨师则需要练习刀工和火候。程序也是一样的，对我来说，语言的各种特性必须要了然于胸。而对软件的开发也需要从代码做起。

经过许久的那么久的开发，我体会到了开发过程所带来的困难和成功开发后所带来的成就感，并且从中了解到了不少开发方法，如极限编程，瀑布开发，敏捷开发等，以及架构和模块间的关系等等，这些都是不断实践出来的收获，希望以后的技术能越发熟练，共勉！