

# Project Report

518030910367 田亚博

2021 年 1 月 5 日

## 1 Introduction

本次实验主要包含三个部分，首先是熟悉给的示例代码，并装好环境，跑通代码，得到一个基础结果；其次就是根据参考文献自己搭建一个 CNN 网络，并使用 MINIST 和我们自己的手写中文数字数据集进行训练；最后写出数独的搜索求解算法，对不同难度级别的题目图像，进行处理识别，求解结果。

本次实验所用的环境较简单，本次实验所用的 python 版本为 python3.7，框架为 tensorflow2.3.1，其余具体环境在 requirements.txt 文件中。

## 2 Explanation

### 2.1 Party I

首先示例代码的代码结构很清晰，pyimagesearch 文件夹下为我们需要调用的文件，在 model 文件夹下为训练模型所需要的网络模型 SudokuNet，利用了 tensorflow 框架来搭建该模型；求解数独所需要的识别图片和提取问题的函数在 sudoku 文件夹下。最外层的 train\_digit\_classifier.py 用来训练并保存模型（模型保存在 output 文件夹下），里面调用了我们实现的模型 SudokuNet 并对其进行训练，训练的性能曲线如图，从 loss 上来看，虽然模型看起来还没有收敛，但是最终的效果还不错，validation accuracy 达到了 0.9906。

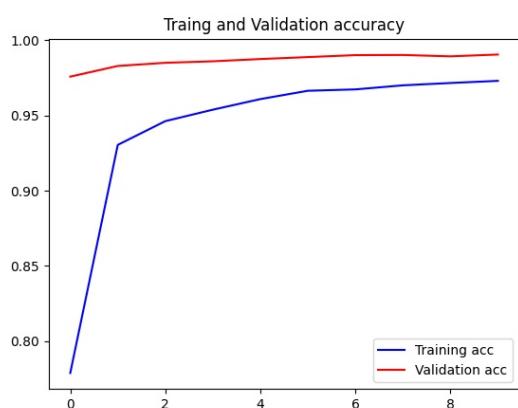


Figure 1: accuracy

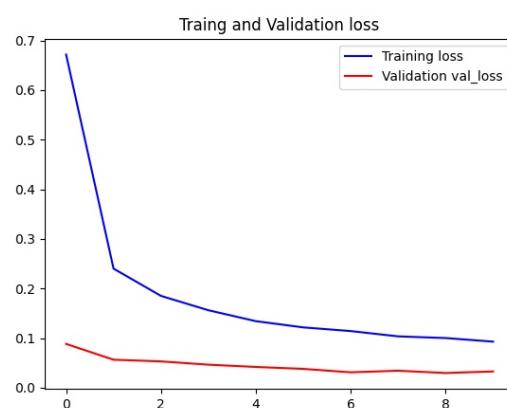


Figure 2: loss

然后运行 solve\_sudoku\_puzzle.py，其中先调用了 sudoku 文件夹下 puzzle.py 中的 find\_puzzle 函数对图片进行识别处理，找到图片中的问题，最后得到的图像如下：

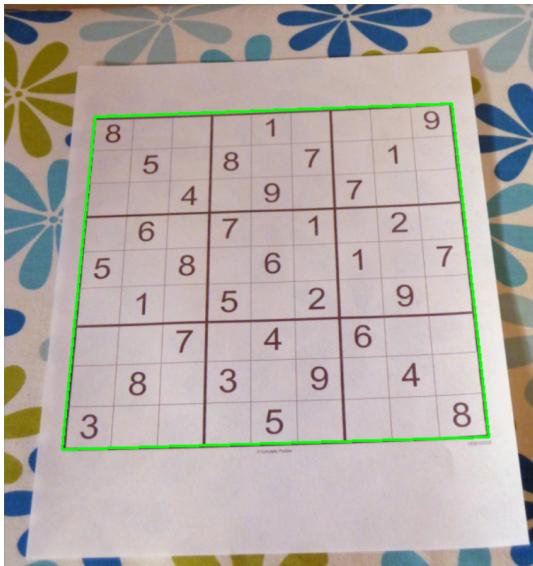


Figure 3: Puzzle Outline

8			1			9		
5		8	7		1			
4		9	7					
6		7	1	2				
5	8	6	1	7				
1		5	2	9				
8	7	4	6					
3		5		4				

Figure 4: Puzzle Transform



Figure 5: Cell Thresh



Figure 6: digit

然后利用我们保存的训练好的模型进行数字识别，当所有小格处理完以后，再对数独进行求解，输出结果，得到的结果图如下

8	7	2	4	1	3	5	6	9
9	5	6	8	2	7	3	1	4
1	3	4	6	9	5	7	8	2
4	6	9	7	3	1	8	2	5
5	2	8	9	6	4	1	3	7
7	1	3	5	8	2	4	9	6
2	9	7	1	4	8	6	5	3
6	8	5	3	7	9	2	4	1
3	4	1	2	5	6	9	7	8

Figure 7: result

这个部分难度不大，装好环境后直接运行代码就可以跑通，示例中的参数也很合理，基本上不用调整就能得到一个很好的模型，参考问题也能顺利识别并解出来。

## 2.2 Party II

### 2.2.1 Model realization

该部分要求我们自己根据参考文献来实现一个 CNN 网络，这个网络就是大名鼎鼎的 Lenet-5，该网络的示意图如图所示，

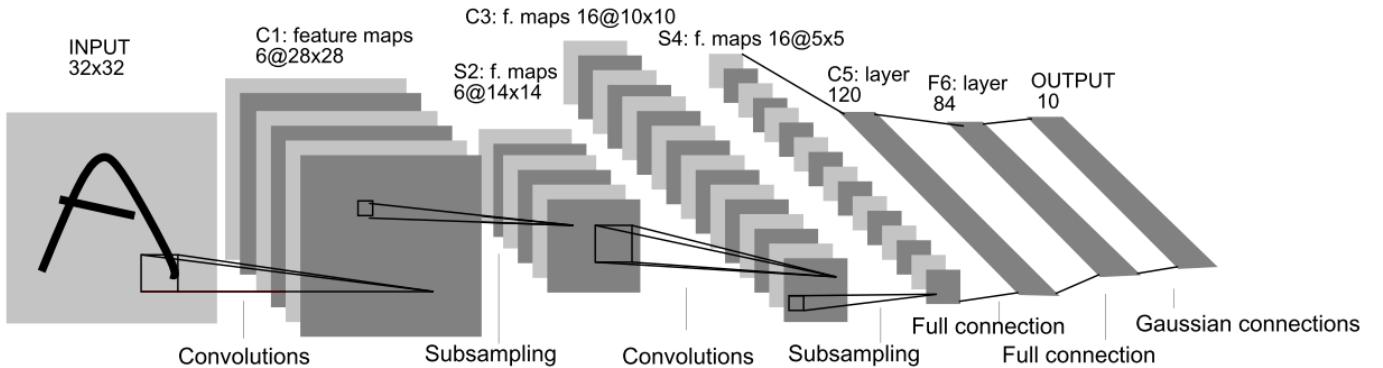


Figure 8: Architecture of Lenet-5

LeNet-5 由 7 层 CNN（不包含输入层）组成，上图中输入的原始图像大小是  $32 \times 32$  像素，卷积层用  $C_i$  表示，子采样层（pooling，池化）用  $S_i$  表示，全连接层用  $F_i$  表示。下面逐层介绍其作用和示意图上方的数字含义。

- C1 层（卷积层）: 6@28×28:** 该层使用了 6 个卷积核，每个卷积核的大小为  $5 \times 5$ ，这样就得到了 6 个 feature map（特征图）。卷积后的图像大小为  $28 \times 28$ 。
- S2 层（下采样层，也称池化层）: 6@14×14:** 这一层主要是做池化或者特征映射（特征降维），池化单元为  $2 \times 2$ ，因此，6 个特征图的大小经池化后即变为  $14 \times 14$ ，池化单元之间没有重叠，在池化区域内进行聚合统计后得到新的特征值，因此经  $2 \times 2$  池化后，每两行两列重新算出一个特征值出来，相当于图像大小减半，因此卷积后的  $28 \times 28$  图像经  $2 \times 2$  池化后就变为  $14 \times 14$ 。
- C3 层（卷积层）: 16@10×10:** C3 层有 16 个卷积核，卷积模板大小为  $5 \times 5$ 。但与之前不同的是，C3 与 S2 并不是全连接而是部分连接，有些是 C3 连接到 S2 三层、有些四层、甚至达到 6 层，通过这种方式提取更多特征，连接的规则如下图所示：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X			X	X	X			X	X	X	X		X	X	
1	X	X			X	X	X			X	X	X	X		X	
2	X	X	X			X	X	X			X		X	X	X	
3		X	X	X			X	X	X	X		X		X	X	
4			X	X	X			X	X	X	X		X	X	X	
5			X	X	X			X	X	X	X		X	X	X	

Figure 9: Map table

- S4 层（下采样层，也称池化层）: 16@5×5:** 与 S2 的分析类似，池化单元大小为  $2 \times 2$ ，因此，该层与 C3 一样共有 16 个特征图，每个特征图的大小为  $5 \times 5$ 。

5. **C5 层（卷积层）: 120:** 该层有 120 个卷积核，每个卷积核的大小仍为  $5 \times 5$ ，因此有 120 个特征图。由于 S4 层的大小为  $5 \times 5$ ，而该层的卷积核大小也是  $5 \times 5$ ，因此特征图大小为  $(5-5+1) \times (5-5+1) = 1 \times 1$ ，这样该层就刚好变成了全连接。
6. **F6 层（全连接层）: 84:** F6 层有 84 个特征图，特征图大小与 C5 一样都是  $1 \times 1$ ，与 C5 层全连接。
7. **OUTPUT 层（输出层）: 10:** Output 层也是全连接层，共有 10 个节点。

为了方便起见，我在编写代码过程中进行了一些近似处理，例如下采样层我选择直接使用最大池化层，loss function 我直接使用的交叉熵函数，实验结果表明影响不大。

代码仍然是根据 tensorflow 框架来写的，这给模型编写带来了极大的便利，该模型的构建比较简单，实现了 Lenet-5 的模型结构。

### 2.2.2 Data preprocessing

由于我们需要在 MINIST 和我们自己的手写数据集上进行训练，所以我还对数据进行了预处理，将 MINIST 和我们自己的手写数据集进行混合，基本思路就是将我们的数据读进一个数组，将 MINIST 读进另一个数组，将两个数组合并然后 shuffle，label 也按照同样的方法合并并按同样的顺序 shuffle，然后将得到的新数据和 label 返回喂给模型即可。

### 2.2.3 Model training

训练自己手写的模型时遇到了比较多的问题，比如开始时 accuracy 上不去，训练出的模型无法识别数字等，这其中大部分都能通过调参来解决，下面是我实验过程中的一些结果。

#### 激活函数 –ReLU 与 Sigmoid

最开始的时候，我用的激活函数是 sigmoid 函数，但是得到的结果较差，accuracy 较低，之后将激活函数改为 relu 函数以后，就得到了更好的结果。所以之后的实验也都是用 relu 函数来作为激活函数进行的。至于原因，应该是因为 sigmoid 函数在模型训练后期梯度太小，导致模型收敛过慢，从而使得得到的效果并不理想，而 relu 函数则很好地解决了这个问题，它在模型训练的任何时期都能保证梯度下降有较好的效果。这是二者在 learning rate 为 0.001，batchsize 为 128，epoch 数为 10 时的训练曲线对比图

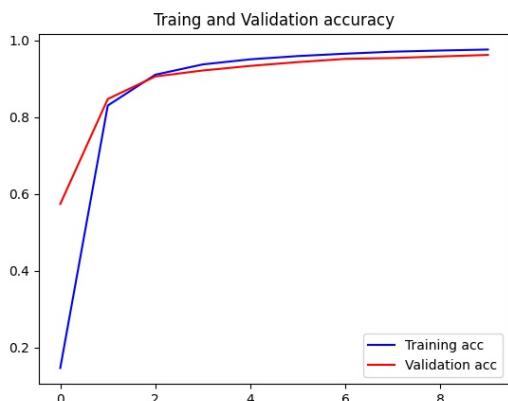


Figure 10: Sigmoid accuracy

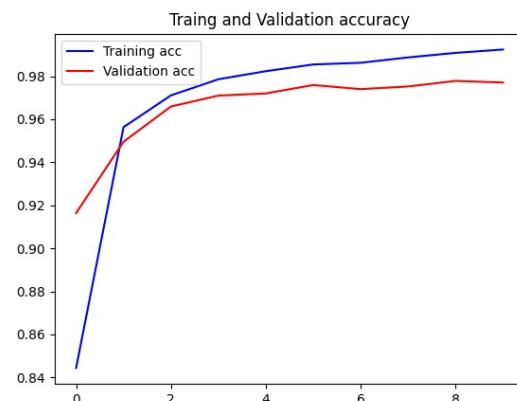


Figure 11: ReLU accuracy

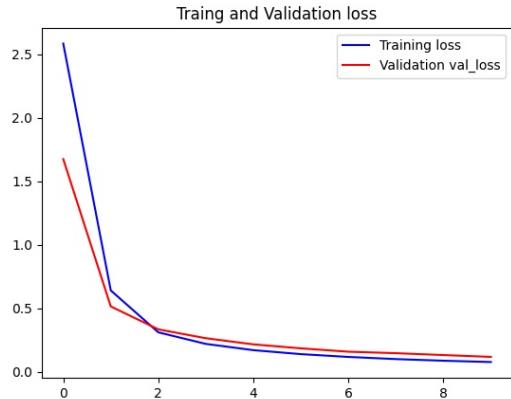


Figure 12: Sigmoid loss

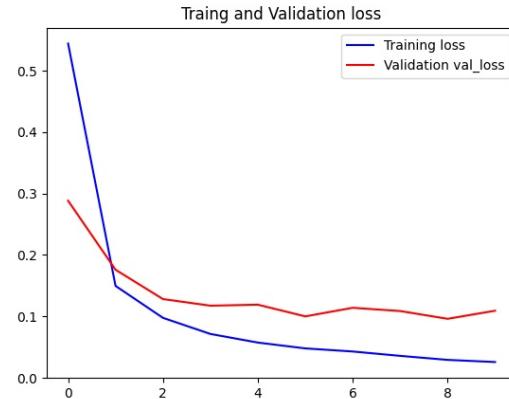


Figure 13: ReLU loss

从图中可以明显看出 ReLU 函数效果比 Sigmoid 函数效果要好，下表是在不同参数下训练的 accuracy(epoch 为参数时，learning rate 均为 0.001，batchsize 为 128；learning rate 为参数时，epoch 数为 10，batchsize 为 128)

Table 1: 不同参数下 ReLU 和 Sigmoid 的训练效果

parameter	Epoch			learning rate	
	10	20	30	0.001	0.002
ReLU val_accuracy	0.9771	0.9760	0.9823	0.9771	0.9755
Sigmoid val_accuracy	0.9624	0.9728	0.9740	0.9624	0.9712

从表格中可以更加清晰地看出，ReLU 激活函数效果好于 Sigmoid 激活函数，而且原因同我上述的分析吻合，Sigmoid 函数在训练后期梯度太小，导致训练缓慢，不管是增加训练轮数还是提高学习率，对识别正确率都有一个正向的效果。同时，epoch 轮数地提高也有助于提高识别正确率，所以，后续的实验均采用 20 个 epoch。

## Dropout

Dropout 是指每个训练批次中，通过忽略一些特征检测器（让一些隐层节点值为 0），可以明显地减少过拟合现象。这种方式可以减少特征检测器（隐层节点）间的相互作用，检测器相互作用是指某些检测器依赖其他检测器才能发挥作用。说的简单一点就是：我们在前向传播的时候，让某个神经元的激活值以一定的概率  $p$  停止工作，这样可以使模型泛化性更强，因为它不会太依赖某些局部的特征。

原本的模型中是不包含 Dropout 部分的，为了探究一下 Dropout 是否对该模型的训练有影响，我做了一些实验来观察不同 Dropout rate 的影响，以下为实验结果（激活函数为 ReLU，learning rate 为 0.001，batchsize 为 128，共训练 20 个 epoch）

Table 2: 不同 dropout rate 对模型训练的影响

Dropout rate	0	20%	50%	80%
val_accuracy	0.9760	0.9803	0.9802	0.9768

其中 dropout rate 为 50% 的性能曲线如图所示：

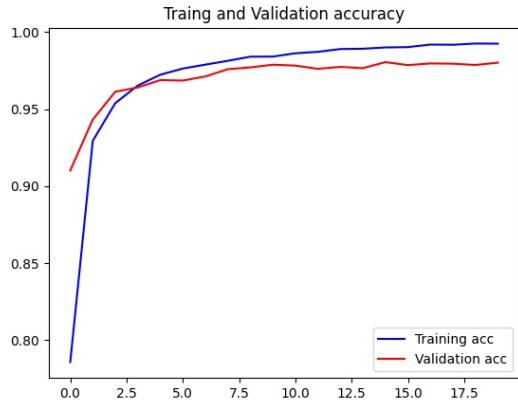


Figure 14: Dropout 50% accuracy

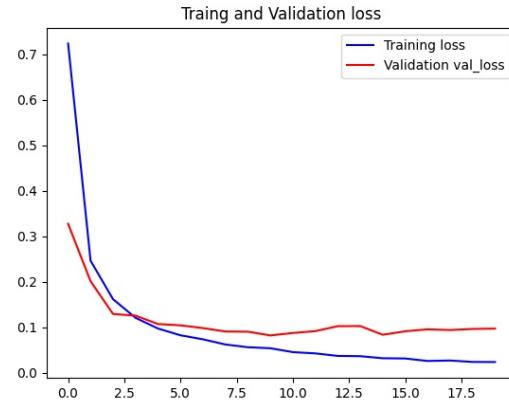


Figure 15: Dropout 50% loss

## 最终模型结构

经过一系列测试与验证，最终模型采用了 ReLU 作为激活函数，并且全连接层采用了 50% 的 dropout，进行接下来的实验。

## Learning rate

尝试了不同的 learning rate，并对比了不同 learning rate 的识别效果，尝试了很多不同的 learning rate，也做过比较细粒度的测试，比如  $1.1e-3$ ,  $1.2e-3$ , ..., 这些细粒度的实验只是为了测试与验证，所以这里就不展示，最终的部分结果对比图表如下：(batchsize 为 128, 共训练 20 个 epoch)

Table 3: 不同 learning rate 对模型训练的影响

learning rate	5e-4	1e-3	1.5e-3	2e-3	3e-3
val_accuracy	0.9788	0.9807	0.9807	0.9799	0.9793

其中效果较好的 learning rate 为  $1e-3$ ,  $1.5e-3$ ,  $2e-3$  的训练性能曲线如下：

- learning rate =  $1e-3$

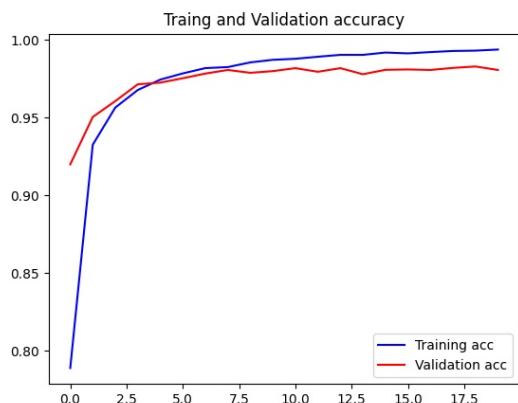


Figure 16: learning rate 0.001 accuracy

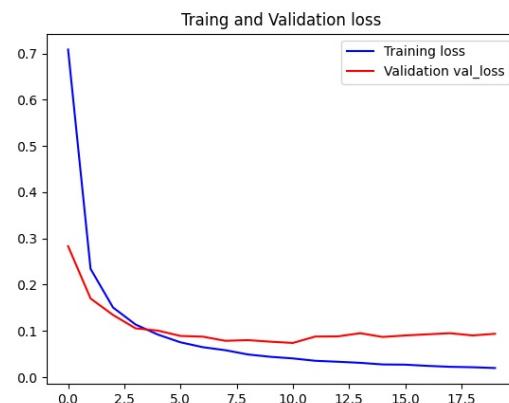


Figure 17: learning rate 0.001 loss

- learning rate = 1.5e-3

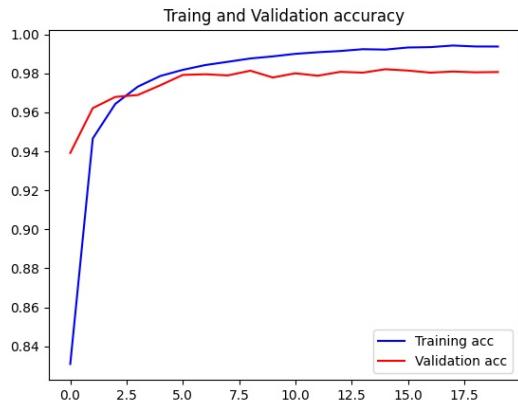


Figure 18: learning rate 0.0015 accuracy

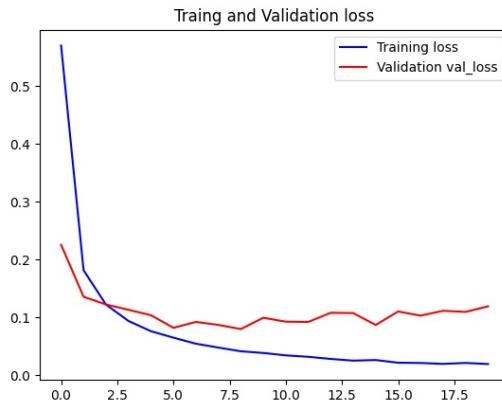


Figure 19: learning rate 0.0015 loss

- learning rate = 2e-3

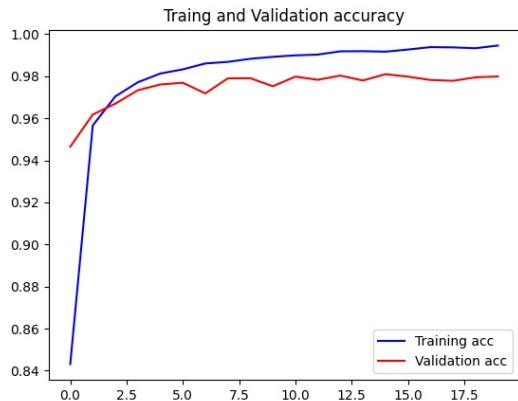


Figure 20: learning rate 0.002 accuracy

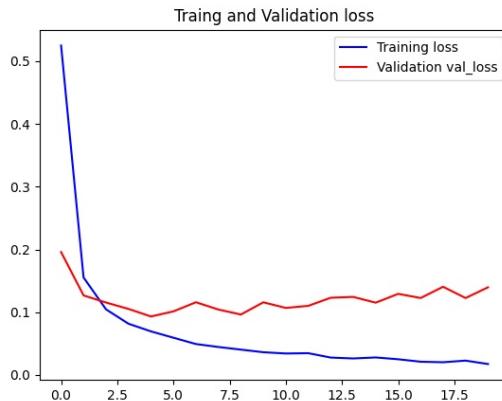


Figure 21: learning rate 0.002 loss

从上图中可以看出，validation accuracy 在后期会有震荡，而我统计的数据是最后一个 epoch 的 accuracy，所以跟最高 accuracy 可能会有一些差别。

### 2.3 Party III

该部分要求我们写出数独的搜索求解算法，这部分思想比较简单，我使用了一个简单的回溯算法，将判断允许的数字填入空处，接着填下一个空，再下一个...（这是一个递归过程）直到某一个空发生跟限制条件冲突的情况，会 pass 掉；直到所有的空都被填满且满足条件，则返回填满的 data 矩阵。其中判断某一处是否允许填入某个数字的函数 judge 如下

```

def judge(data, x, y, num): #关键函数一，判断数字是否重复，是否允许填入
    if data[y].count(num) > 0: #行判断
        #print('error1')
        return False

    for col in range(9): #列判断
        if data[col][x] == num:
            #print('error2')
            return False

    for a in range(3): #九宫格判断
        for b in range(3):
            if data[a+3*(y//3)][b+3*(x//3)] == num:
                #print('error3')
                return False
    return True

```

Figure 22: judge function

其中还有一个初始化函数，build\_data\_list，它的作用是为每个空位建立备选数字列表，具体函数如下

```

def build_data_list(data): #初始化，未每个空位建立备选数字列表
    data_list = []
    for y in range(9):
        for x in range(9):
            if data[y][x] == 0:
                data_list.append([(x, y), [1, 2, 3, 4, 5, 6, 7, 8, 9]])
    return data_list

```

Figure 23: 为每个空处建立备选数字列表

然后是最重要的求解函数，即刚刚所说的递归函数，具体代码如下

```

def fill_num(data, data_list, start): #关键函数二，对有多个备选数
    始下一位置的猜测；若某位置为False，则忽略。
    if start < len(data_list):
        one = data_list[start]
        for num in one[1]:
            if judge(data, one[0][0], one[0][1], num):
                data[one[0][1]][one[0][0]] = num
                tem_data = fill_num(data, data_list, start+1)
                if tem_data != None:
                    return tem_data
            data[one[0][1]][one[0][0]] = 0 #有可能再往后猜了好几步后
        else:
            return data

```

Figure 24: 递归求解

但是这部分代码跟上述的递归求解相比多了两个部分。第一个部分为判断返回值是否为None，只有当不为None时才会返回，这是因为当第n+1阶的for loops执行完都没找到可填的数字时，此时的第n+1阶的递归函数也会返回值（只不过这个值为None）。一旦某一阶函数返回了值，其外部的函数依次返回，后续的猜测就终止了，无法找到正确值（正确值非None）。

因此需要增加这部分代码，避免某函数 for loops 结束，返回值为 None 时，引起外层函数连续 return，中断操作。

第二个部分为在 for loop 全部执行完以后，对过程中的所有赋值操作清零。这是因为尽管对返回值设了检查，return 有了保障，但是毕竟某一阶或多阶的 for loops 已经错误执行，导致 data 被多次错误赋值，而且一旦 for loops 能完整执行，表明前面一定填错了，因此要将每一个错误的填数清零。

最后是一个简单的示例，证明该算法可以正确地解出数独。

0	2	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
0	0	0	3	0	0	0	0	4
0	6	0	0	0	0	5	0	0
0	0	0	0	7	0	0	0	0
0	0	0	0	0	0	2	0	0
7	0	9	0	0	0	0	0	0
0	0	0	0	2	0	0	8	0
0	0	0	0	0	0	0	0	0
3	2	4	5	6	1	7	9	8
5	1	8	4	9	7	3	2	6
6	9	7	3	8	2	1	5	4
1	6	2	8	3	4	5	7	9
9	4	3	2	7	5	8	6	1
8	7	5	6	1	9	2	4	3
7	5	9	1	4	8	6	3	2
4	3	1	7	2	6	9	8	5
2	8	6	9	5	3	4	1	7

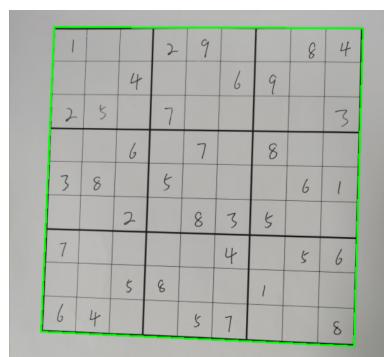
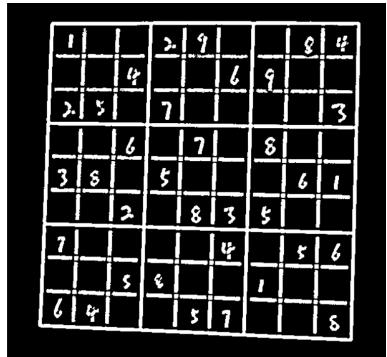
Figure 25: 数独求解示例

### 3 Result

我对每一张测试图片都做了识别测试，最终结果发现汉字识别效果整体较好，比较满意，但是阿拉伯数字识别准确率较低，下面是具体每一张图片的分步处理结果。

#### 1-1

经 find\_puzzle 函数处理的图片



1	2	9		8	4			
	4			6	9			
2	5	7			3			
	6	7		8				
3	8	5			6	1		
	2	8	3	5				
7			4		5	6		
	5	8		1				
6	4	5	7		8			

Figure 26: 1-1 Puzzle Thresh    Figure 27: 1-1 Puzzle Outline    Figure 28: 1-1 Puzzle Transform

经 extract\_digit 函数处理的图片

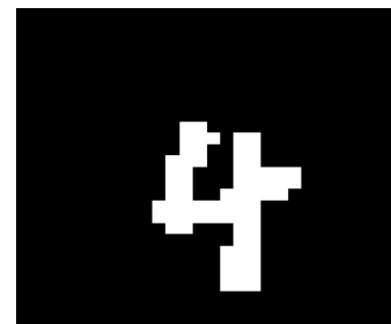
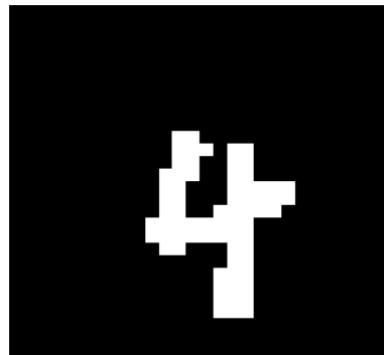


Figure 29: 1-1 Original

Figure 30: 1-1 Cleared

Figure 31: 1-1 Digit

其中 Original 为传入 extract\_digit 函数原本的图片，Cleared 为去除边框之后的图片，Digit 为经过 extract\_digit 函数处理后的图片。

最终识别结果

0	0	0	5	9	0	0	8	4
0	0	4	0	0	6	9	0	0
1	5	0	3	0	0	0	0	3
0	0	9	0	7	0	8	0	0
3	8	0	5	0	0	0	6	1
0	0	5	0	8	3	8	0	0
9	0	0	0	0	9	0	3	9
0	0	3	6	0	0	10	0	0
6	4	0	0	3	1	0	0	5

Figure 32: 1-1 识别结果

由于 `find_puzzle` 函数处理和 `extract_digit` 函数处理的结果大同小异，所以接下来就不再展示这部分图片，只展示最终结果。

## 1-2

图片 1-2 最终识别结果：

0	0	10	0	0	0	4	0	10
0	7	8	0	0	0	6	0	3
0	0	0	0	0	5	0	1	0
1	3	0	0	0	10	0	7	0
0	0	0	5	0	9	0	0	0
0	4	0	10	0	2	0	3	6
0	7	0	4	0	0	0	0	0
3	0	7	0	0	0	5	1	0
5	0	4	0	0	0	2	0	0

Figure 33: 1-2 识别结果

### 1-3

图片 1-3 最终识别结果:

6	0	0	0	0	0	6	0	0
4	0	0	4	2	0	0	0	0
0	10	5	0	3	0	0	0	0
5	0	0	8	0	9	0	0	0
3	0	4	0	0	0	3	0	3
0	0	0	8	0	3	0	0	1
0	0	0	0	7	0	7	9	0
0	0	0	0	9	1	0	0	6
0	0	3	0	0	0	0	0	4

Figure 34: 1-3 识别结果

### 1-4

图片 1-4 最终识别结果:

0	4	0	0	0	10	0	0	0
0	0	0	4	0	0	7	6	0
0	0	0	0	0	1	9	0	9
0	0	5	8	0	7	0	9	0
0	0	0	0	0	0	0	0	0
0	3	0	3	0	6	0	9	0
2	0	1	3	0	0	0	0	0
0	6	8	0	0	9	0	0	0
0	0	0	6	0	0	0	3	0

Figure 35: 1-4 识别结果

## 1-5

图片 1-5 最终识别结果:

0	9	0	7	1	0	0	9	0
0	10	4	0	0	8	5	0	0
0	0	8	9	0	0	8	8	0
7	0	0	0	0	10	0	0	8
8	0	5	0	8	9	10	0	8
0	0	0	0	6	0	0	0	4
0	8	6	0	0	10	7	0	0
0	0	3	16	0	0	7	1	0
0	8	0	0	3	4	0	8	0

Figure 36: 1-5 识别结果

接下来是汉字图片识别结果

## 2-1

图片 2-1 最终识别结果:

4	5	8	7	9	2	1	3	6
1	3	2	4	5	6	7	9	8
6	7	9	8	1	3	5	4	2
9	1	7	5	3	8	2	6	4
3	2	5	6	4	9	8	7	1
8	6	4	2	7	1	9	5	3
5	4	6	1	8	7	3	2	9
7	8	3	9	2	4	6	1	5
2	9	1	3	6	5	4	8	7

Figure 37: 2-1 识别结果

图片 2-1 成功识别出来，并且求出了该数独的解，整体很成功。

## 2-2

图片 2-2 最终识别结果:

0	0	5	7	0	0	0	0	6
0	0	9	0	8	6	0	0	4
0	0	2	0	3	0	8	0	0
0	6	0	0	0	5	0	7	0
9	0	0	5	0	0	0	0	2
0	2	0	1	5	0	0	6	0
0	0	3	0	5	0	9	0	0
4	0	0	8	7	0	9	0	0
7	0	0	0	0	2	6	0	0

Figure 38: 2-2 识别结果

图片 2-2 成功识别出来大部分手写数字，经过对比，错误识别了三个手写数字，整体比较成功。

## 2-3

图片 2-3 最终识别结果:

2	0	0	1	9	0	0	5	0
0	1	0	6	0	0	0	7	0
5	0	0	0	0	9	9	0	0
0	0	0	0	0	5	7	0	0
8	0	0	0	0	2	0	0	6
0	0	6	0	0	0	0	0	0
0	0	2	3	0	0	0	0	9
0	3	0	0	0	8	0	2	0
0	7	0	0	1	9	0	0	5

Figure 39: 2-3 识别结果

图片 2-3 和图片 2-2 类似，也是成功识别出来大部分手写数字，经过对比，只将一个“七”错误识别为了“九”，整体比较成功。

## 2-4

图片 2-4 最终识别结果:

4	6	7	2	3	5	9	8	1
2	3	1	8	4	9	5	7	6
8	9	5	1	6	7	2	3	4
1	5	2	3	9	4	8	6	7
9	7	8	5	1	6	3	4	2
3	4	6	7	8	2	1	5	9
6	8	4	9	2	3	7	1	5
5	1	9	6	7	8	4	2	3
7	2	3	4	5	1	6	9	8

Figure 40: 2-4 识别结果

图片 2-4 也是成功识别出来大部分手写数字，经过对比，只将一个“五”错误识别为了“九”，但是由于这个识别错误比较特殊，所以最终求出来了一组解。

## 2-5

图片 2-5 最终识别结果:

0	0	9	3	0	2	1	0	5
0	0	5	7	0	4	0	0	2
0	7	1	0	2	0	0	0	0
9	8	0	0	0	3	0	0	0
0	0	0	6	0	8	0	0	0
0	0	0	9	0	0	0	6	2
0	0	0	0	5	0	3	3	0
2	0	0	4	0	7	9	0	0
3	0	4	5	0	9	2	0	0

Figure 41: 2-5 识别结果

图片 2-5 成功识别出来大部分手写数字，经过对比，错误识别了三个手写数字，整体比较成功。

## 4 Conclusion

本次实验整体比较成功，要求基本完成，并且熟悉了 CNN 的构建与调试，积累了不少调试经验，通过解决调试过程中遇到的问题也对卷积神经网络有了一些直觉。另外，原本计划同时也将 pytorch 版本完成，但是由于最后一段时间比较繁忙，任务较多，虽然代码基本已经写好，但是还没有来得及调试，等到最后考试周结束，我会将 pytorch 版本调试完成，积累更多的经验。