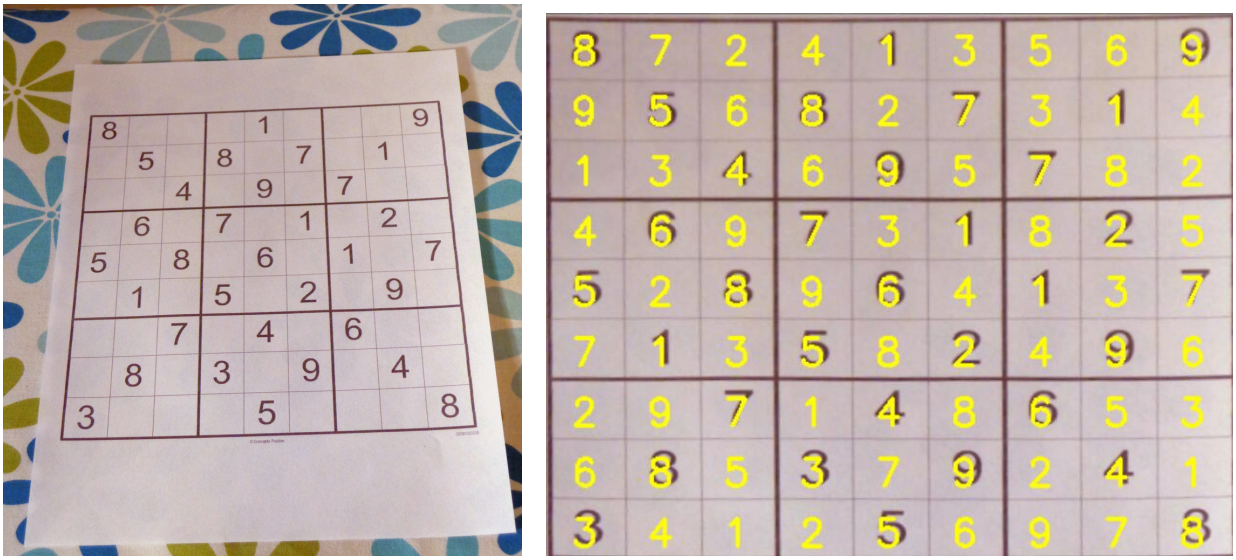
BE PROJECT REPORT

REPORT OF PROJECT THREE FOR EI339

Sudoku

Gong Chen 518030910301
Department of Electronics and Telecommunications
SHANGHAI JIAO TONG UNIVERSITY

Contents



(a) Original Image (b) Result Image

Figure 1: Result Images of the Reference Code

1 Successful Running the Reference Codes

The project requires us to run successfully the reference code given by our teacher. Figure 1(a) and figure 1(b) are the exhibition of the results:

2 Data Process and Optimization Methods

2.1 Introduction to Database

This project uses database which consists of two parts: MNIST and SJTU-EI339-CN.

- MNIST has 60000 data points for training and 10000 data points for testing. Each data point includes a 28×28 numpy array and a label. Each array represents a gray image of an English digit number and each label represents the class of the number.
- SJTU-EI339-CN has 4093 data points for training and 861 data points for testing. Most data points include a 28×28 numpy array and a label but some data points do not have standard form and size. Each array represents a gray image of an Chinese digit number and each label represents the class of the number.

2.2 Analysis of Database

The **SJTU-EI339-CN database** which is established by us students has several characteristics:

- The digit number is not in the central position of the image.
- The sizes of the images are different from one another.
- Some images have experienced image binarization but some are just gray scale images.
- Many images have noisy like dots or lines distributing in the image.

The **MNIST database** has already been processed, so it is really standard and has the following characteristics:

- Every digit has the normalized size 28×28.
- The digit is in the central 20×20 square of the image and the surroundings are black area.

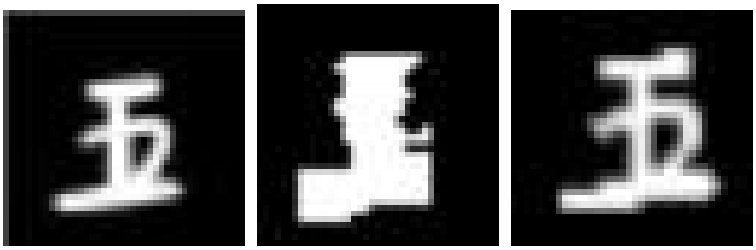
Therefore, the indispensable procedure is to process the data in SJTU-EI339-CN for **normalization of the database**.

2.3 Traditional Methods for Processing Data in SJTU-EI339-CN

According to the characteristics of the database listed in the previous section, we can come up with several ways to pre-process the database to make it become normalized.

1) Eliminate Noisy:
   The normal way to eliminate the noisy in the image is using **Gaussian Blur** function. However, we find that Gaussian Blur can not only eliminate the noisy distributing among the image, but can also **make the strokes of the characters thicker** and even unable for us to recognize. The change from Figure 2(a) to figure 2(b) exhibits this kind of situation vividly and experiments prove that this phenomenon largely hurts our model's performance.

2) Resize:
   To normalize the size of the picture, we can use the function $np.resize$ to make all pictures' size become 28×28.



(a) Original Digit (b) Gaussian Blur (c) My Process Operation

Figure 2: Digit Process

3) Image Binarization:
This can be easily implemented. We only need to define an appropriate threshold and thus divide all the pixels into two classes: lower-value class is changed to 0 and higher-value class is changed to 255.

4) Move Object to Centre:
To move the object to the centre of the image, we need to detect the object of the image first. Usually, we achieve this by counting the percentage of pixels with value higher than the threshold of each column as well as each row and then locate the object according to these percentages. However, the existence of noisy prevents us from doing this.

To sum up, existing methods I have known can not satisfy my data processing requirements. Therefore, I come up with a special data processing method which proves to have great effect.

2.4 My Optimized Methods for Processing Data

My optimized methods for data processing include three steps:

1) **Digit Location Detection based on Gaussian Blur Method**:
Use Gaussian Blur method to eliminate noisy and count the percentage of pixels with value larger than threshold of each row and each column of this blur image. Use these percentages to calculate the coordinates $(startX, endX, startY, endY)$ of the object in this image.

```
1 img_original = img.copy()
2 img_blur = cv2.GaussianBlur(img, (7, 7), 3)
3 img_blur = cv2.adaptiveThreshold(img_blur, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
      cv2.THRESH_BINARY,11, 2)
```

```
img_original = cv2.adaptiveThreshold(img_original, 255, cv2.
    ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,11, 2)
img_blur = cv2.bitwise_not(img_blur)
img_original = cv2.bitwise_not(img_original)
img = modify_img(img_blur, img_original)
```

2) **Cut the Object from Original Image**:

Take away the object from the original image according to the coordinates obtained in the previous step. The following function is used to detect the location of the object in the blur image and cut it from the original image. The input parameters are $thresh$ (image after Gaussian Blur) and $img\_original$ (the original image). The output is the minimum square area in the image that contains the object. This function's code can be seen as follows:

```
def modify_img(thresh, img_original):
    (h, w) = thresh.shape
    cnt_h = [0 for i in range(h)]
    cnt_w = [0 for i in range(w)]
    for i in range(0, h):
        for j in range(0, w):
            if thresh[i][j] != 0:
                cnt_h[i] += 1
                cnt_w[j] += 1

    h_min = h-1
    h_max = 0
    w_min = w-1
    w_max = 0
    for i in range(len(cnt_h)):
        if cnt_h[i] >= 1:
            h_min = min(h_min, i)
            h_max = max(h_max, i)
    for i in range(len(cnt_w)):
        if cnt_w[i] >= 1:
            w_min = min(w_min, i)
            w_max = max(w_max, i)
    length = max(h_max-h_min, w_max-w_min)

    h_max_new = (h_max + h_min + length + 1) // 2
    h_min_new = (h_max + h_min - length + 1) // 2
```

```
    w_max_new = (w_max + w_min + length + 1) // 2
    w_min_new = (w_max + w_min - length + 1) // 2

    new_thresh = img_original[h_min_new:h_max_new + 1, w_min_new:w_max_new + 1]
    if (h_min_new < 0 or h_max_new >= h):
        if (h_min_new < 0):
            new_thresh = img_original[0:h_max_new + 1, w_min_new:w_max_new + 1]

            new_thresh = np.pad(new_thresh, ((-h_min_new, 0), (0, 0)), 'constant
                ', constant_values=0)
        else:
            new_thresh = img_original[h_min_new:h, w_min_new:w_max_new + 1]
            new_thresh = np.pad(new_thresh, ((0, h_max_new-h+1), (0, 0)), '
                constant', constant_values=0)
    if (w_min_new < 0 or w_max_new >= w):
        if (w_min_new < 0):
            new_thresh = img_original[h_min_new:h_max_new + 1, 0:w_max_new + 1]
            new_thresh = np.pad(new_thresh, ((0, 0), (-w_min_new, 0)), 'constant
                ', constant_values=0)
        else:
            new_thresh = img_original[h_min_new:h_max_new + 1, w_min_new:w]
            new_thresh = np.pad(new_thresh, ((0, 0), (0, w_max_new-w+1)), '
                constant', constant_values=0)
    if (new_thresh.shape == (0,0)):
        print(h_min_new, h_max_new, w_min_new, w_max_new)

    print(new_thresh.shape)
    return new_thresh
```

3) **Normalize Size**:

Because MNIST database's data has the property that the object area has size 20×20 and the integral image has size 28×28, we firstly resize the object we extracted in the second step to size 20×20 and then add paddings to make the image becomes 28×28.

4) **Decentralize**:

This means that we decentralize the values of pixels in the image. We achieve this through updating the value of each position: $value_{new} = \frac{value_{old}-0.5}{0.5}$. I tried this operation in order to improve my model's performance. The specific operation can be seen clearly through the following code:

```
TrainData = (TrainData - 0.5) / 0.5
TestData = (TestData - 0.5) / 0.5
```

5) **Shuffle the Data**:

This is used to shuffle the data, which has shown great improvement in model's performance compared with directly using the data in order.

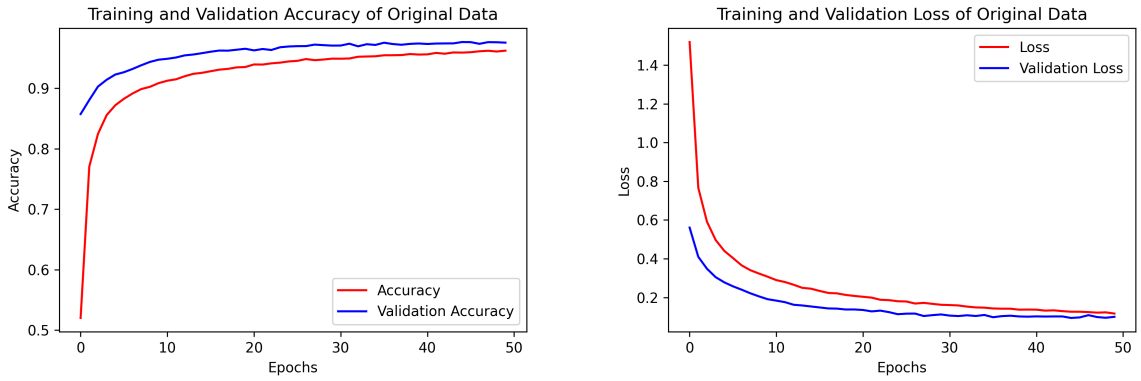
The results and comparison between different performance when using different data processing methods can be seen in the next section.

2.5 Results and Comparison

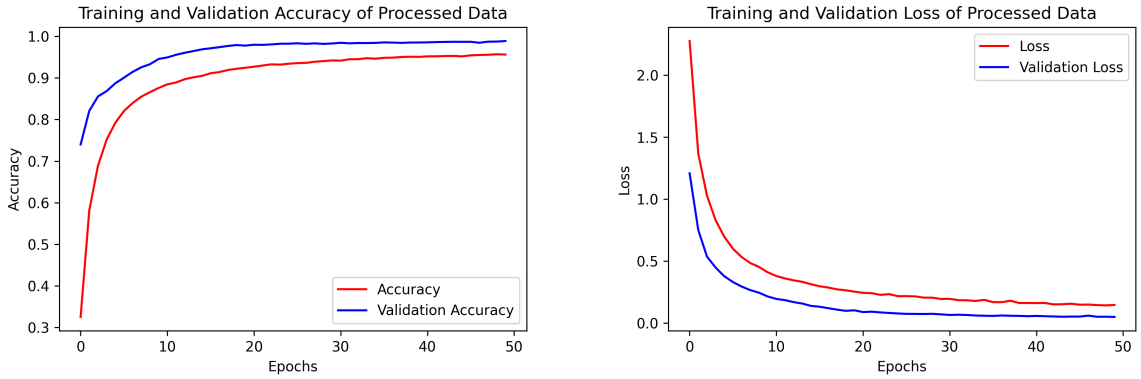
To control the variable and test the performance of the different data through different data processing methods, I do some experiments **based on the same Net**. The net I use is given by our teacher's reference code. It may not be the most appropriated model net but I use it in order to speed up the training process and see the performance of our data processing methods immediately. The **curve of testing accuracy and testing loss against epochs with different data processing methods** can be seen in figure 3(a), 3(b) and 3(c). The specific **f1-score** of different classes based on different data processing methods but the same model can be seen in table 1(C means Chinese digit and E means English digit).

Table 1
testing f1-score of different numbers when using different data processing methods

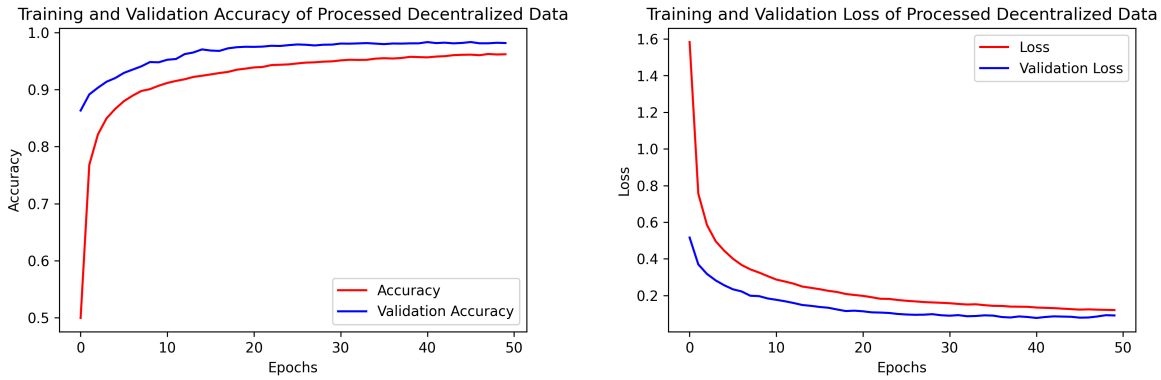
f1-score	Raw Data	Processed Data	Processed Decentralized Data
E0	0.99	0.99	0.99
E1	0.99	**1.00**	1.00
E2	0.98	**0.99**	0.99
E3	0.99	0.99	0.99
E4	0.99	0.99	0.99
E5	0.99	0.99	0.99
E6	0.99	0.99	0.99
E7	0.99	0.99	0.99
E8	0.99	0.99	0.99
E9	0.98	**0.99**	0.98
C1	0.87	0.98	**1.00**
C2	0.79	0.97	**0.98**
C3	0.79	**0.97**	0.94
C4	0.81	**0.98**	0.98
C5	0.79	**0.94**	0.86
C6	0.76	**0.94**	0.86
C7	0.81	**0.92**	0.60
C8	0.95	**0.98**	0.93
C9	0.76	**0.94**	0.69



(a) Original Data



(b) Processed Data



(c) Processed Decentralized Data

Figure 3: Performance of Different Data Processing Methods based on Same Model

From the table and curve, we can draw some conclusions:

1) If we use the raw data, the performance of identifying English digits is much better than identifying Chinese digits. The reason is that MNIST data has the same form like

position of digit number in the image but data images in our SJTU database are not normalized.

2) Our processing methods can largely improve the performance of identifying Chinese digit numbers. The f1-score can be averagely improved from 0.76+ to 0.94+

3) Decentralized operation only makes a little influence on the performance of identifying Chinese number 1 and 2 but will make identifying other Chinese numbers much worse. The reason may be that decentralizing leads to negative values of some positions, which can influence the model through activation function $Relu$.

4) The curves tell us that if using our data processing method, not only can the testing accuracy reach nearly 100%, but the loss can also decrease more quickly and more nearly to 0

3 Implement of LeNet-5 using only $Numpy$

In this section, I am going to introduce my implementation of $LeNet5$. In order to understand the mechanism of different layers deeply, I do not use any functions of $Tensorflow$ or $PyTorch$. Instead, I only use $numpy$ to realize different functions.

My work can be divided into three parts: realization of different layers, realization of different loss functions and realization of $LeNet-5$. They will be introduced later and the results of the net will also be exhibited in this section.

3.1 Layers

3.1.1 Convolution Layer

1) Parameters

The parameters and their explanation can be seen as follows:

```
class Convolution():
    def __init__(self, input_channels, filters, kernel=5, feature_mapping=None,
        activation='relu'):
        '''
        input_channels: number of channels of input
        filters: number of filters
        kernel: size of the filters
        feature_mapping: which channels of the input need to be convoluted
```

```
        activation: activation function
        '''
        self.input_channels = input_channels
        self.filters = filters
        self.kernel = kernel
        self.feature_mapping = feature_mapping
        if self.feature_mapping is None:
            self.feature_mapping = np.ones((self.filters, self.input_channels))
        self.activation = activation
        self.init_parameters()

    def init_parameters(self):
        """
        W: 4D (filters, input_channels, width, height)
        b: 1D (filters)
        """
        self.W = np.random.randn(self.filters, self.input_channels, self.kernel,
             self.kernel) / np.sqrt(self.input_channels)
        # self.w = np.ones((self.filters, self.input_channels, self.kernel, self
            .kernel))
        for i in range(self.filters):
            for j in range(self.input_channels):
                if not self.feature_mapping[i][j]:
                    self.W[i][j] = 0
        self.b = np.zeros(self.filters)
```

2) Pass Forward

- Firstly, we have to know the mechanism of passing forward which is shown through equation (1):

$$a^l = \sigma(z^l) = \sigma(a^{l-1} * W^l + b^l) \tag{1}$$

  where a^{l-1} denotes the input, W^l denotes the filter, b^l denotes the bias.

- Secondly, we define the convolution function $convolution()$. This function is used to calculate the convolution of the input and one filter when given the feature mapping list (decide which channels of the input are involved).

```
def convolution(self, input, filter, mapping=None):
    """
    input: 3D (channels, width, height)
```

```
    filter: 3D (channels, width, height)
    output: 2D (width, height)
    """
    dim = np.subtract(input[0].shape, filter[0].shape) + 1 # dimension of the
        output
    if mapping is None: # fully connect
        mapping = np.ones(input.shape[0])
        output = np.zeros(dim)
    for i in range(dim[0]):
        for j in range(dim[1]):
            p = np.multiply(input[:, i:i+filter.shape[1], j:j+filter.shape[2]],
                filter).sum((1, 2))
            output[i][j] = np.sum(p * mapping)
    return output
```

- Thirdly, we define the $pass\_forward()$ function, which is used to calculate the output based on the net's parameters when given the input.

```
def pass_forward(self, input):
    """
    input: 3D (channels, width, height)
    output: 3D (filters, width, height)
    """
    self.input = input
    output = []
    for i in range(self.filters): # calculate the convolution of input and each
        filter
        self.z = self.convolution(input, self.W[i], self.feature_mapping[i]) +
            self.b[i]
        if self.activation == 'relu':
            output.append(np.maximum(0, self.z))
        else:
            output.append(self.z)
    self.output = np.array(output)
    return self.output
```

3) Pass Backward

This part is used to update the parameters of the net when given the gradient of the output(Δ) and the learning rate(η).

- The mechanism of the backward of input z^{l-1}, weight W^l, bias b^l when given gradient of the output J can be seen as follows:

$$\begin{aligned}
\frac{\partial J}{\partial z^{l-1}} &= \frac{\partial J}{\partial z^l}\frac{\partial z^l}{\partial z^{l-1}} = (W^l)^T\frac{\partial J}{\partial z^l}\sigma'(z^{l-1}) = \frac{\partial J}{\partial z^l} * rot180(W^l)\sigma'(z^{l-1}) \\
\frac{\partial J}{\partial W^l} &= \frac{\partial J}{\partial z^l}\frac{\partial z^l}{\partial W^l} = a^{l-1} * \frac{\partial J}{\partial z^l} \\
\frac{\partial J}{\partial b^l} &= \sum_{u,v}(\frac{\partial J}{\partial z^l})_{u,v}
\end{aligned} \tag{2}$$

- We calculate the $pass\_backward()$ according to the equation (2). The explanation and size of the input and output as well as the explanation of each step are note in the following code:

```
def pass_backward(self, delta, eta):
    """
    delta: 3D (filters, width, height)
    output: 3D (input_channel, width, height)
    """
    if self.activation == 'relu':
        delta = delta * (self.z >= 0)
    # Using padding makes it convenient to calculate backward of the input
        through convolution
    delta_new = np.pad(delta, ((0,), (self.kernel-1,), (self.kernel-1,)), mode='
        constant', constant_values=0)
    delta_input = []
    for i in range(self.input_channels):
        W_j_i_rotate = np.array([np.rot90(np.rot90(self.W[j][i])) for j in range
            (self.filters)])
        delta_input.append(self.convolution(delta_new, W_j_i_rotate))
        delta_input = np.array(delta_input)

    for i in range(self.filters):
        delta_i = np.array([delta[i]])
            self.b[i] -= eta * np.sum(delta_i)
        for j in range(self.input_channels):
            if not self.feature_mapping[i][j]:
                    continue
            input_j = np.array([self.input[j]])
```

```
            delta_w = self.convolution(input_j, delta_i, self.feature_mapping[i
                ][j])
            self.W[i][j] -= eta * delta_w

    return delta_input
```

3.1.2 Fully Connect Layer

1) Parameters

The size and explanation of the parameters are noted in the following codes:

```
class FullyConnect():
    def __init__(self, input_size, output_size, activation='relu'):
        self.input_size = input_size # size of the input
        self.flat_input_size = np.product(input_size) # flat size of input
        self.output_size = output_size # size of output
        self.activation = activation # activation function
        self.init_parameters()

    def init_parameters(self):
        '''
        W: 2D (flat size of input, size of output)
        b: 1D (size of output)
        '''
        self.W = np.random.randn(self.flat_input_size, self.output_size) / np.
            sqrt(self.flat_input_size)
        # self.w = np.ones((self.output_size, self.flat_input_size))
        self.b = np.zeros(self.output_size)
```

2) Pass Forward

The calculation of the output y when given input x is quite easy. We only need to calculate the dot product of the input and the weight matrix W and plus the bias b:

$$y = W \cdot x + b \tag{3}$$

The code can be seen as follows:

```
def pass_forward(self, input):
    self.input = input.reshape(self.flat_input_size)
    output = np.dot(self.input, self.W) + self.b
    if self.activation == 'relu':
        output = np.maximum(0, output)
    return output
```

3) Pass Backward

- The mechanism of the backward function of Fully Connect Layer is shown through equation (4).

$$\begin{aligned} \frac{\partial J}{\partial x} &= \frac{\partial J}{\partial y} * W^T \\ \frac{\partial J}{\partial W} &= x^T * \frac{\partial J}{\partial y} \\ \frac{\partial J}{\partial b} &= \frac{\partial J}{\partial y} \end{aligned} \tag{4}$$

- The corresponding code to calculate the gradient of x, W as well as b and update them can be seen as follows:

```
def pass_backward(self, delta, eta):
    delta_W = np.outer(self.input, delta)
    delta_b = delta
    delta_input = np.dot(delta, self.W.transpose())

    self.W -= eta * delta_W
    self.b -= eta * delta_b
    return delta_input.reshape(self.input_size)
```

3.1.3 MaxPooling Layer

1) Parameters

We only need three parameters: $input\_channels$ to denote the number of channels of the input, $size$ to denote the size of max pooling area and $stride$ to denote the stride of the max pooling area.

```
class MaxPooling():
    def __init__(self, input_channels, size=2, stride=2):
        self.channels = input_channels
        self.size = size
        self.stride = stride
```

2) Pass Forward

The mechanism of max pooling is to represent a certain area with the max value in it. In our environment, when the $size$ is 2 and $stride$ is 2, the output y can be calculated through:

$$y_{i,j} = \max(x_{2i,2j}, x_{2i+1,2j}, x_{2i,2j+1}, x_{2i+1,2j+1}) \tag{5}$$

We also need to store the position of the max value of a max pooling area which is used when passing backward. The corresponding code including notes can be seen as follows:

```
def pass_forward(self, input):
    '''
    input: 3D (channels, width, height)
    '''
    input_w, input_h = input.shape[1:]
    output_w, output_h = input_w // self.stride, input_h // self.stride
    output = np.zeros((self.channels, output_w, output_h))
    self.pos = np.zeros((self.channels, output_w, output_h), dtype=np.int)
    for i in range(self.channels):
        for j in range(0, input_w, self.stride):
            for k in range(0, input_h, self.stride):
                output[i, j//self.stride, k//self.stride] = np.max(input[i, j:j+
                    self.size, k:k+self.size])
                self.pos[i, j//self.stride, k//self.stride] = np.argmax(input[i,
                     j:j+self.size, k:k+self.size])
    return output
```

3) Pass Backward

The mechanism of passing backward when given the derivative of output y and input

x is:

$$\frac{\partial J}{\partial x_i} = \begin{cases} 0, & \delta(i,j) = false \\ \frac{\delta J}{\delta y_j}, & \delta(i,j) = true \end{cases} \tag{6}$$

where x_i denotes the input node i, y_j denotes the output node j, $\delta(i,j)$ denotes whether x_i is chosen by y_j as the max value. The code with explanation can be seen as follows:

```
def pass_backward(self, delta):
    '''
    delta: 3D (channels, width, height)
    '''
    input_w, input_h = delta.shape[1:]
    output_w, output_h = input_w * self.stride, input_h * self.stride
    delta_input = np.zeros((self.channels, output_w, output_h))
    for i in range(self.channels):
        for j in range(0, input_w):
            for k in range(0, input_h):
                delta_input[i, j*self.stride+self.pos[i, j, k]//self.size, k*
                    self.stride+self.pos[i, j, k]%self.size] = delta[i, j, k]
    return delta_input
```

3.1.4 Softmax Layer

1) Parameter

In this layer, we only need one parameter $size$ denoting the size of the input.

```
class Softmax():
    def __init__(self, size):
        self.size = size
```

2) Pass Forward The mechanism of passing forward when given input x is:

$$y_i = \frac{e^{y_i}}{\sum_{j=1}^{c} e^{x_j}} \tag{7}$$

Therefore, the code of $pass\_forward()$ can be seen as follows:

```
def pass_forward(self, input):
    self.input = input.reshape(self.size)
    e = np.exp(self.input - np.max(self.input))
    self.output = e / np.sum(e)
    return self.output
```

3) Pass Backward

- The mechanism of passing backward when given gradient of output y is:

$$
\begin{aligned}
i = j: \ & \frac{\partial y_i}{\partial x_j} = \frac{\partial y_i}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\frac{e^{x_i}}{\sum_k e^{x_k}}\right) = \frac{e^{x_i}\left(\sum_k e^{x_k}\right) - e^{x_i} e^{x_i}}{\left(\sum_k e^{x_k}\right)^2} = y_i(1 - y_i) \\
i \neq j: \ & \frac{\partial y_i}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_k e^{x_k}}\right) = \frac{-e^{x_i} e^{x_j}}{\left(\sum_k e^{x_k}\right)^2} = -\frac{e^{x_i}}{\sum_k e^{x_k}} \frac{x_j}{\sum_k e^{x_k}} = -y_i y_j \\
\Rightarrow & \frac{\partial Y}{\partial X} = diag(Y) - Y^T Y
\end{aligned}
\tag{8}
$$

- Therefore, the code of $pass\_backward()$ function can be seen as follows:

```
def pass_backward(self, delta):
    predict = np.argmax(self.output)
    m = np.zeros((self.size, self.size))
    m[:, predict] = 1
    m = np.eye(self.size) - m
    d = np.diag(self.output) - np.outer(self.output, self.output)
    d = np.dot(delta, d)
    d = np.dot(d, m)
    return d
```

3.2 Loss

I tried two kinds of loss functions: MSE and LogLikelihood. As these two loss functions are familiar to us, I will briefly introduce them and show the codes.

3.2.1 Mean Square Error

The calculating equation is:

$$MSE = \frac{1}{n}\sum_{i}(y_i - \hat{y}_i)^2 \tag{9}$$

wherr y_i is the ground truth adn $\hat{y}_i$ is the output of our net. The corresponding codes can be seen as follows:

```
class MSE():
    def loss(y_true, y_pred):
        return np.sum(np.square(y_true - y_pred)) / 2.

    def derivative(y_true, y_pred):
        return y_pred - y_true
```

3.2.2 Log Likelihood

```
class LogLikelihood():
    def loss(y_true, y_pred):
        loss = np.sum(y_true * y_pred)
        loss = -np.log(loss) if loss != 0 else 500
        return loss

    def derivative(y_true, y_pred):
        d = y_pred.copy()
        d[np.argmax(y_true)] -= 1
        return d
```

3.3 LeNet-5

1) Introduction

This part is to construct LeNet-5 based on the layers we defined before. The architecture of LeNet-5 is shown in figure 4.

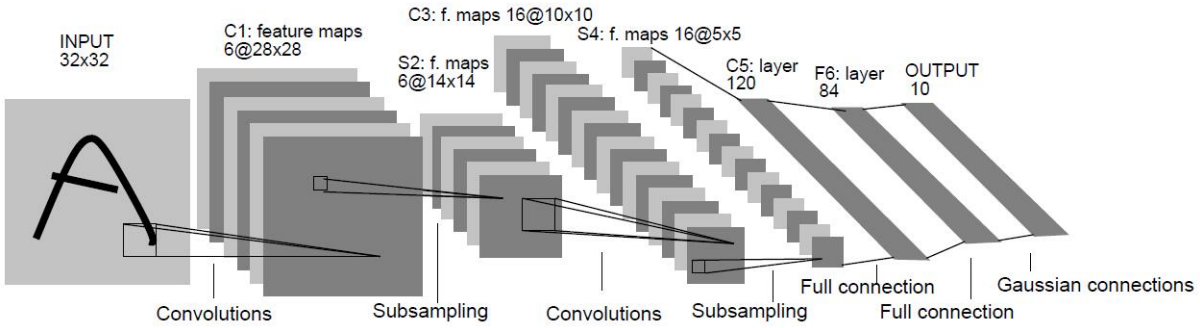


Figure 4: Architecture of LeNet-5

We can directly see that, LeNet consists of two Convolution layers, two MaxPooling layers, three Full Connection layers and one Softmax layer. We should notice that the first convolution layer is based on all the features produced by the previous layer and the second convolution layer is based on different features produced by the previous layers, which can be shown in figure 5. Therefore, we need to import its feature maps when constructing the layer.

	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 5: Feature Maps of the Second Convolution Layer

2) Initialization

The corresponding codes of constructing LeNet-5 are shown as follows:

```
class LeNet5():
    def __init__(self, input_size):
        '''
        input_size: 3D (channels, width, height)
        '''
        self.input_size = input_size
```

```
        self.conv1 = Convolution(input_size[0], 6)
        self.maxpool1 = MaxPooling(6)
        self.conv2 = Convolution(6, 16, feature_mapping=[
            [1, 1, 1, 0, 0, 0], [0, 1, 1, 1, 0, 0], [0, 0, 1, 1, 1, 0],
            [0, 0, 0, 1, 1, 1], [1, 0, 0, 0, 1, 1], [1, 1, 0, 0, 0, 1],
            [1, 1, 1, 1, 0, 0], [0, 1, 1, 1, 1, 0], [0, 0, 1, 1, 1, 1],
            [1, 0, 0, 1, 1, 1], [1, 1, 0, 0, 1, 1], [1, 1, 1, 0, 0, 1],
            [1, 1, 0, 1, 1, 0], [0, 1, 1, 0, 1, 1], [1, 0, 1, 1, 0, 1],
            [1, 1, 1, 1, 1, 1]
        ])
        self.maxpool2 = MaxPooling(16)
        self.fc1 = FullyConnect((16, 4, 4), 120)
        self.fc2 = FullyConnect(120, 84)
        self.fc3 = FullyConnect(84, 10)
        self.softmax = Softmax(10)
```

3) Pass Forward

The $forward()$ function of LeNet-5 is simply calling $pass\_forward()$ function of each layer in constructing order when given the input.

```
def forward(self, x):
    out = self.conv1.pass_forward(x)
    out = self.maxpool1.pass_forward(out)
    out = self.conv2.pass_forward(out)
    out = self.maxpool2.pass_forward(out)
    out = self.fc1.pass_forward(out)
    out = self.fc2.pass_forward(out)
    out = self.fc3.pass_forward(out)
    out = self.softmax.pass_forward(out)
    return out
```

4) Pass Backward

The $back\_ward()$ function of LeNet-5 is simply calling $pass\_backward()$ function of each layer in the reverse constructing order when given the derivative of the output. The codes can be seen as follows:

```
def backward(self, delta, eta):
    out = self.softmax.pass_backward(delta)
```

```
        out = self.fc3.pass_backward(out, eta)
        out = self.fc2.pass_backward(out, eta)
        out = self.fc1.pass_backward(out, eta)
        out = self.maxpool2.pass_backward(out)
        out = self.conv2.pass_backward(out, eta)
        out = self.maxpool1.pass_backward(out)
        out = self.conv1.pass_backward(out, eta)
        return out
```

3.4 Train Function

The main function to train the LeNet-5 can be seen as follows:

```
def main():
    # train_input, train_label, val_input, val_label, test_input, test_label =
        load_mnist()
    train_input = np.load('D:/3/AI/opencv-sudoku-solver/Data/English_train_data.npy')
    train_label = np.load('D:/3/AI/opencv-sudoku-solver/Data/English_train_labels.npy
        ')
    test_input = np.load('D:/3/AI/opencv-sudoku-solver/Data/English_test_data.npy')
    test_label = np.load('D:/3/AI/opencv-sudoku-solver/Data/English_test_labels.npy')
    train_label = np.array([vectorize(test_label[i], 10) for i in train_label])
    test_label = np.array([vectorize(test_label[i], 10) for i in test_label])
    train_input = np.reshape(train_input, (60000, 1, 28, 28))
    test_input = np.reshape(test_input, (10000, 1, 28, 28))
    print(train_input.shape)
    print(test_input.shape)
    seq = np.arange(len(train_input))

    net = LeNet5(train_input[0].shape)
    for epoch in range(epochs):
        if shuffle: np.random.shuffle(seq)
        for step in range(len(train_input)):
            i = seq[step]
            x = train_input[i]
            y_true = train_label[i]
            y = net.forward(x)
            loss = LogLikelihood.loss(y_true, y)
            dloss = LogLikelihood.derivative(y_true, y)

```

```
            # print('Epoch %d step %d loss %f' % (epoch, step, loss))
            d = net.backward(dloss, lr)

            if step > 0 and step % 1000 == 0:
                net.save()
                correct = 0
                loss = 0
                #for i in range(len(test_input)):
                for i in range(200):
                    x = test_input[i]
                    y_true = test_label[i]
                    y = net.forward(x)
                    loss += LogLikelihood.loss(y_true, y)
                    if np.argmax(y) == np.argmax(y_true): correct += 1
                print('Validation accuracy: %.2f%%, average loss: %f' % (correct
                    /200*100, loss/len(test_input)))
        correct = 0
        for i in range(len(test_input)):
            x = test_input[i]
            y_true = test_label[i]
            y = net.forward(x)
            if np.argmax(y) == np.argmax(y_true):
                correct += 1
        print('Test accuracy: %.2f%%' % (correct/len(test_input)*100))
```

3.5 Results and Conclusions

In the training process, I test the model on testing database for every 3000 steps and record the testing loss and testing accuracy where one step means training on a data point and updating the model. The curves of testing accuracy and testing loss against training steps can be seen through figure 6:

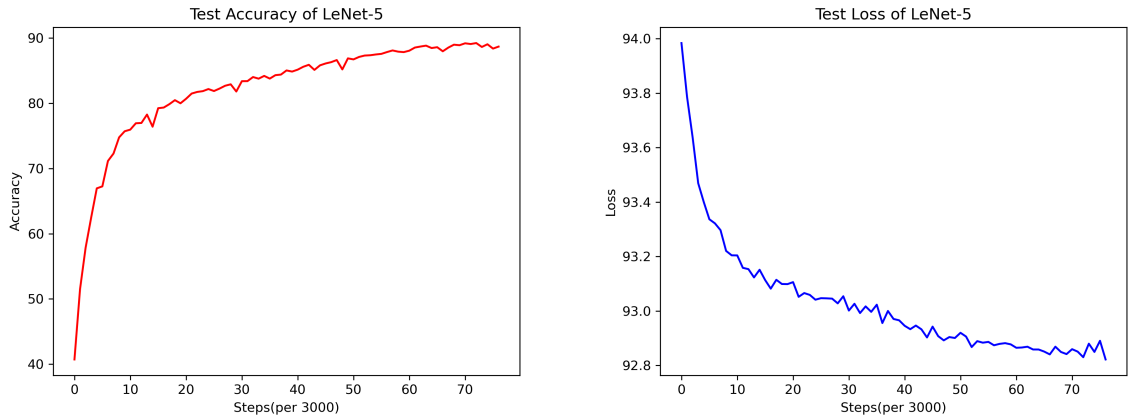


Figure 6: Accuracy and Average Loss on Training Data of LeNet-5

We can draw some conclusions from the curves and results of LeNet-5 wrriten by myself:

- It can be seen clearly that I successfully construct LeNet-5. The net is trainable and can be trained to detect digital numbers with higher accuracy with the passing of time.
- The testing accuracy of the LeNet-5 converges to 90% after about 70×3000 steps but the testing loss is keep decreasing which means that the model could still be improved. However, the training process is too slow that I have to run two or three days to make the loss converge. The reason is analyzed in the next item. Therefore, I only train several epochs, but the result and performance can still be seen directly and obviously.
- My own implementation of LeNet has really low speed. The reasons are: I only use $numpy$ to calculate without any parallel tools to accelerate the calculating process; Instead of inputting a batch of data points at a time, the Net calculates the corresponding gradient and updates the parameters for every data point(will make the model more accurate but also will take more time to converge).

4 Different Nets for Detecting Digit numbers

According to General Function Approximation Theory, the neural net can simulate nearly all the functions as long as the number of parameters is large enough and the number of hyper-parameters we have tried is large enough.

Therefore, instead of only using LeNet-5, I try another four different net models in order to simulate the real classification function more perfectly. These nets are also defined and

built by myself. However, I use *Keras* to implement the layers in these net models in order to speed up the training process using parallel calculating tools. Then, I choose the net with the best performance to detect digit when solving Sudoku problem.

I will separately introduce these four nets: Original Net, Shallow Net, Deep Net and ALex Net and also compare their performance as well as results in this section.

4.1 Original Net

This net is the model given by our teacher in the reference code. The architecture of this net can be seen in figure 7. It has two Convolution layers, two Max-Pooling layers and three Fully-Connected layers and all the layers are activated by function *Relu*. The definition of this net can be seen as follows:

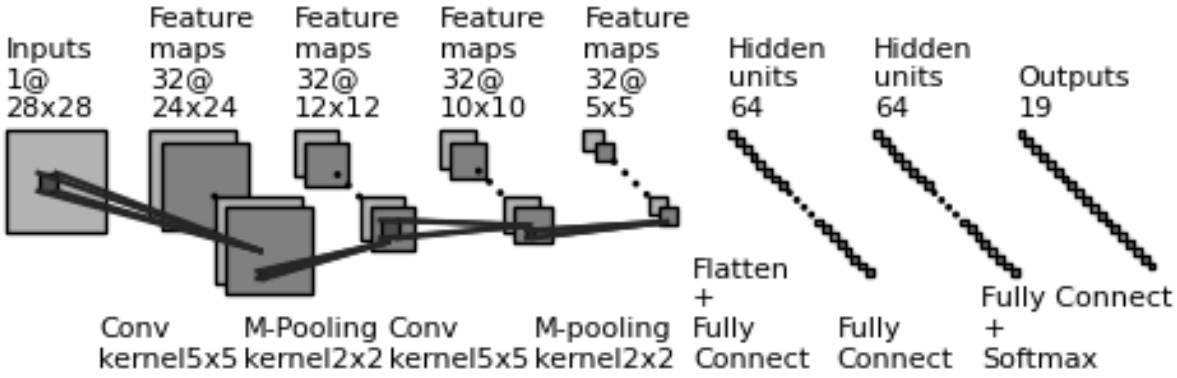


Figure 7: Architecture of Original Net

```
def build(width, height, depth, classes):
    model = Sequential()
  inputShape = (height, width, depth)
  model.add(Conv2D(32, (5, 5), padding="same", input_shape=inputShape))
  model.add(Activation("relu"))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  model.add(Conv2D(32, (3, 3), padding="same"))
  model.add(Activation("relu"))
  model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
  model.add(Dense(64)) #
  model.add(Activation("relu"))
  model.add(Dropout(0.5))
```

```
16
17    model.add(Dense(64))
18    model.add(Activation("relu"))
19    model.add(Dropout(0.5)) # Fully-connected layer set with 50% dropout
20
21    model.add(Dense(classes))
22    model.add(Activation("softmax"))
23
24    return model
```

4.2 Shallow Net

This is a shallow net with less parameters. I try this net to avoid the appearance of overfitting because I initially regard digit classification as an easy task and it only needs a small number of parameters. However, the result compared with other net models proves I am wrong which can be seen in the next section.

The architecture of Shallow Net can be seen in figure 8. It has two Convolution layers, two Max-Pooling layers, three Fully-Connected layers and all the layers are activated by function $Relu$. The definition of this Net can be seen as follows:

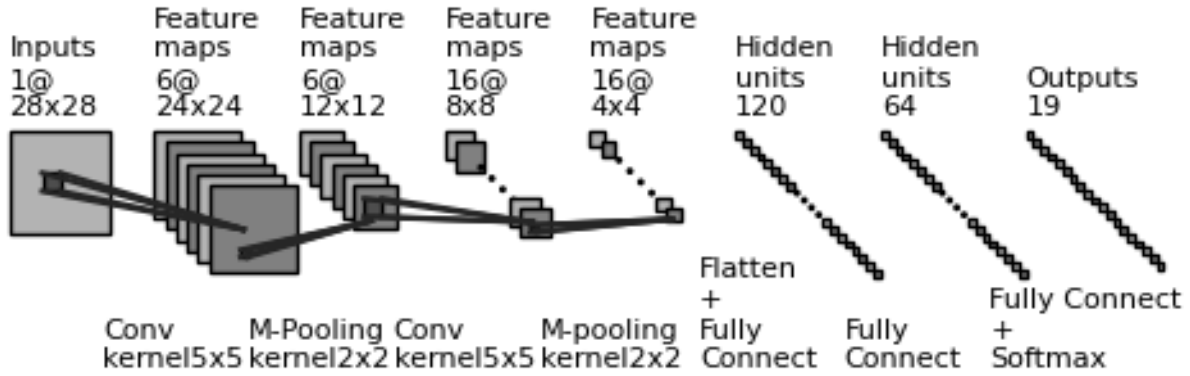


Figure 8: Architecture of Shallow Net

```
1  def build(width, height, depth, classes):
2      model = Sequential()
3      inputShape = (height, width, depth)
4
5      model.add(Conv2D(6, (5, 5), padding="valid", use_bias=True, input_shape=
           inputShape))
```

```
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(16, (5, 5), padding="valid", use_bias=True))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(120))
    model.add(Activation("relu"))
    model.add(Dropout(0.5))

    model.add(Dense(84))
    model.add(Activation("relu"))
    model.add(Dropout(0.5))

    model.add(Dense(classes))
    model.add(Activation("softmax"))

    return model
```

4.3 Deep Net

Deep Net means that this model has more layers and parameters which has better performance when the classification task is really difficult.

The architecture of Deep Net can be seen in figure 9. It has three Convolution layers, three Max-Pooling layers, five Fully-Connected layers and all the layers are activated by function $Relu$.

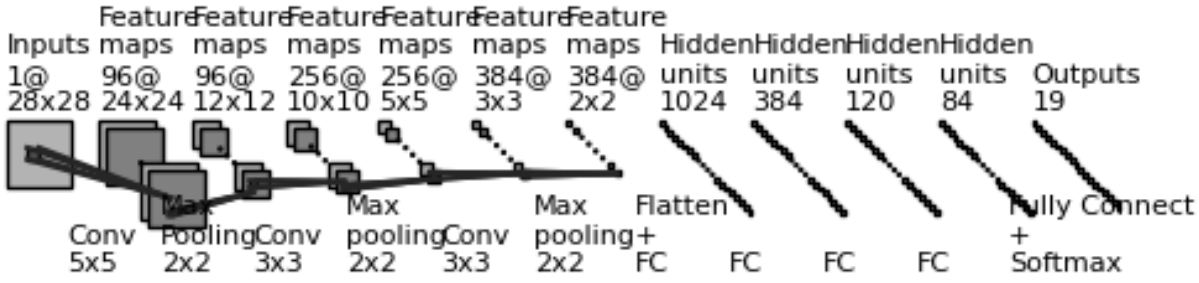


Figure 9: Architecture of Deep Net

The definition of Deep Net can be seen as follows:

```
def build(width, height, depth, classes):
    model = Sequential()
    inputShape = (height, width, depth)

    model.add(Conv2D(96, (5, 5), padding="valid", use_bias=True, input_shape=
        inputShape))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(256, (3, 3), padding="valid", use_bias=True))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(384, (3, 3), padding="valid", use_bias=True))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(1024))
    model.add(Activation("relu"))
    model.add(Dropout(0.5))

    model.add(Dense(384))
    model.add(Activation("relu"))
    model.add(Dropout(0.5))

    model.add(Dense(120))
    model.add(Activation("relu"))
    model.add(Dropout(0.5))

    model.add(Dense(84))
    model.add(Activation("relu"))
    model.add(Dropout(0.5))

    model.add(Dense(classes))
    model.add(Activation("softmax"))

    return model
```

4.4 Alex Net

Alex Net is a famous net model. In this project, I transform the original Alex Net to a simpler model which can be implemented more easily and directly.

The architecture of Alex Net can be seen in figure 10. It has three Convolution layers, three Max-Pooling layers, three Fully-connected layers and all the layers are activated by function $Relu$.

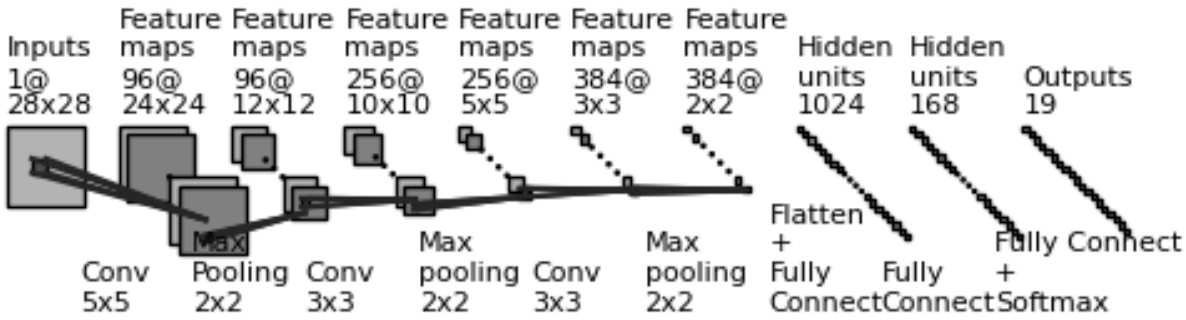


Figure 10: Architecture of Alex Net

The definition of Alex Net can be seen as follows:

```
def build(width, height, depth, classes):
    model = Sequential()
    inputShape = (height, width, depth)

    model.add(Conv2D(96, (5, 5), padding="valid", use_bias=True, input_shape=
        inputShape))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Activation(tf.nn.local_response_normalization))
    model.add(Dropout(0.5))

    model.add(Conv2D(256, (3, 3), padding="valid", use_bias=True))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Activation(tf.nn.local_response_normalization))
    model.add(Dropout(0.5))

    model.add(Conv2D(384, (3, 3), padding="valid", use_bias=True))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
```

```
22      model.add(Dense(1024))
23      model.add(Activation("relu"))
24      model.add(Dropout(0.5))
25
26      model.add(Dense(168))
27      model.add(Activation("relu"))
28      model.add(Dropout(0.5))
29
30      model.add(Dense(classes))
31      model.add(Activation("softmax"))
32
33      return model
```

4.5 Training Model

It is simple to train the model defined by ourselves using *Keras* and *Tensorflow*. The corresponding codes can be seen as follows:

```
1  if __name__ == '__main__':
2      INIT_LR = 1e-4
3      EPOCHS = 30
4      BS = 32
5
6      trainData = np.load("mix_train_data_28.npy")
7      testData = np.load("mix_test_data_28.npy")
8      trainLabels = np.load("mix_train_labels_28.npy")
9      testLabels = np.load("mix_test_labels_28.npy")
10
11     le = LabelBinarizer()
12     trainLabels = le.fit_transform(trainLabels)
13     testLabels = le.transform(testLabels)
14
15     print("[INFO] compiling model...")
16     opt = Adam(lr=INIT_LR)
17     model = SudokuNet.build(width=28, height=28, depth=1, classes=19)
18     model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"
           ])
19
20     print("[INFO] training network...")
21     H = model.fit(
```

```
22          trainData, trainLabels,
23          validation_data=(testData, testLabels),
24          batch_size=BS,
25          epochs=EPOCHS,
26          verbose=1)
27
28      print("[INFO] evaluating network...")
29      predictions = model.predict(testData)
30      print(classification_report(
31          testLabels.argmax(axis=1),
32          predictions.argmax(axis=1),
33          target_names=[str(x) for x in le.classes_]))
34
35      print("[INFO] serializing digit model...")
36      model.save("model.h5", save_format="h5")
```

4.6 Results and Comparison between Different Nets

4.6.1 Comparison on F1-score between Different Nets

The performances of different Net models can be seen in table 2 in the form of f1-score of different classes when testing. Because f1-score considers both precision and recall, it is reasonable to use this metric to judge the performance.

Table 2
f1-score of Different Numbers based on Different Nets(E: English, C: Chinese)

f1-score	Original Net	Shallow Net	Deep Net	Alex Net
E0	0.99	0.99	**1.00**	0.99
E1	0.99	0.99	1.00	**1.00**
E2	0.99	0.98	0.99	**0.99**
E3	0.99	0.98	0.99	**1.00**
E4	0.99	0.98	0.99	**0.99**
E5	0.99	0.98	0.99	**0.99**
E6	0.99	0.99	0.99	**0.99**
E7	0.98	0.98	0.99	**0.99**
E8	0.99	0.98	0.99	**0.99**
E9	0.98	0.98	0.99	**0.99**
C1	0.97	0.98	0.99	**0.99**
C2	0.95	0.96	0.98	**0.98**
C3	0.91	0.93	0.97	**0.98**
C4	0.99	0.97	0.99	**0.99**
C5	0.90	0.87	0.95	**0.97**
C6	0.95	0.87	0.95	**0.96**
C7	0.90	0.84	0.96	**0.97**
C8	0.97	0.97	**0.99**	0.98
C9	0.96	0.91	0.97	**0.98**

From the table, we can draw some conclusions:

1) The performance of identifying English digits is much better than identifying Chinese digits. However, using **Alex Net can largely narrows the gap** between them.

2) **Alex Net gets the highest f1-score** in most classes when testing. Especially when classifying **Chinese digits 5, 6, 7**, Alex Net's f1-score can still reach **over 0.96** but the other net models perform really poorly.

3) Alex Net is more suitable for this classification task and thus we will use it to detect numbers when solving Sudoku.

4.6.2 Comparison on Loss and Accuracy between Different Nets

The training and testing accuracy against epochs of different net models can be seen in figure 11. From this figure, it seems that all the nets' testing(validation) accuracy can reach nearly 100% and thus they have the same performance. However, this is just an illusion. There are over 60000 images of English digits but only 4000 images of Chinese digits and high total accuracy does not mean the high accuracy on every class.

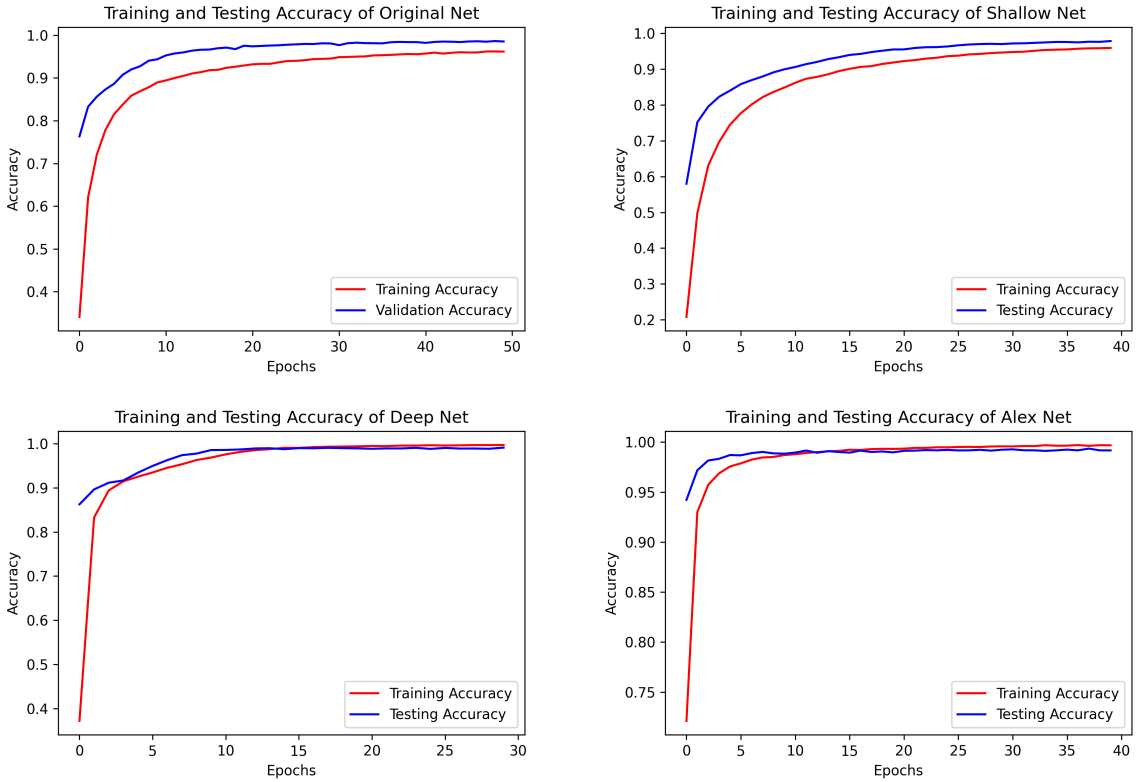


Figure 11: Training and Testing Accuracy of Different Nets

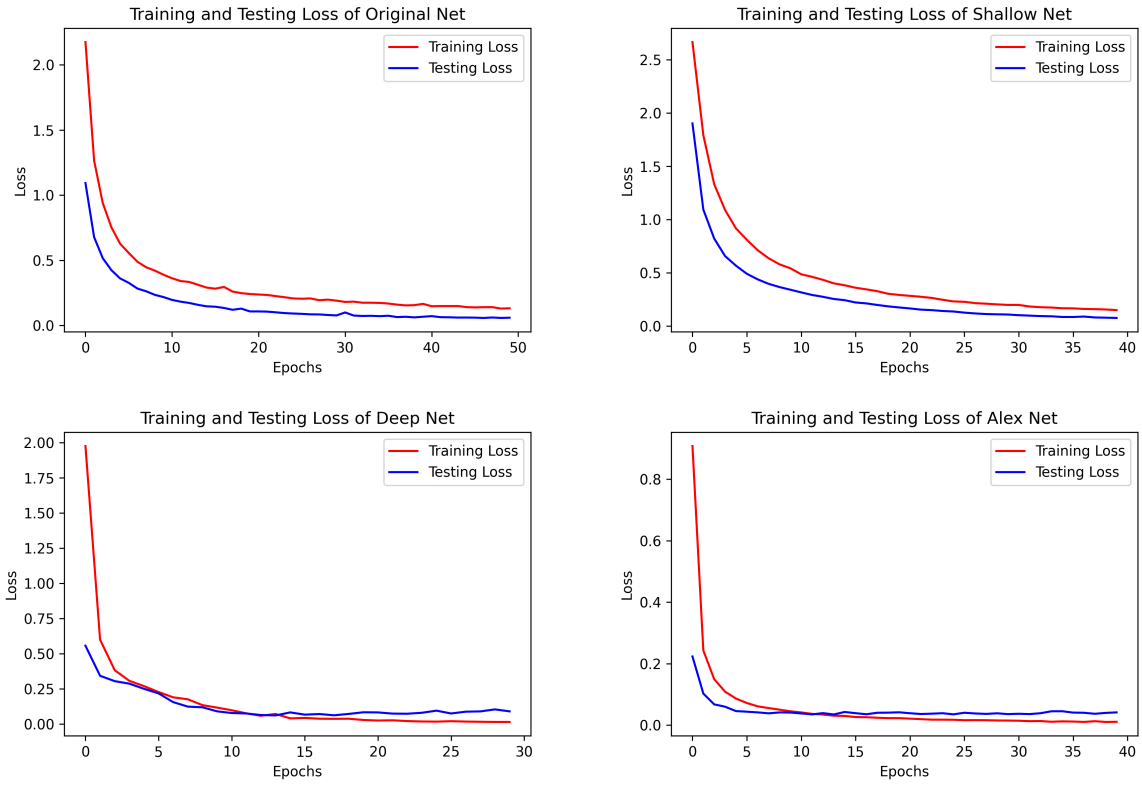


Figure 12: Training and Testing Loss of Different Nets

From the curves, we can draw some conclusions:

1) Alex Net's testing accuracy against epochs is increasing more quickly than the other three nets and converges to nearly 1 in the shortest time. This means that, compared with other nets, Alex Net needs less time to achieve a good performance.

2) Alex Net's testing loss is decreasing more quickly than the other three nets. This means that Alex Net can train the model more effectively as well as efficiently.

3) The Shallow Net's testing accuracy is increasing the most slowly and its training loss is decreasing the most slowly, too. This means that our classification task is essentially complex as well as difficult which needs a lot of layers and parameters to simulate the real classification function. Shallow Net can not fit this task.

5 Sudoku Class

In this section, I will show my definition of Sudoku Class including its initialization, difficulty calculation, solution and how to correct a wrong Sudoku.

5.1 Initialization and Difficulty Calculation

There are three elements in the Sudoku class:

- $board$: the original 9×9 board
- $board\_correct$: if there exists mistakes in the board, $board\_correct$ is used to store the board after our correction
- $difficulty$: the difficulty of a Sudoku problem is defined as the percentage of blank positions among all positions. The more blanks means that we need to try more situations and the problem is more difficult.

To initialize a Sudoku, we need to input a 9×9 matrix. Then, we initialize both the $board$ and $board\_correct$ as the matrix input and calculate the difficulty:

```
class Sudoku:
    def __init__(self, board):
        self.board = board
        self.blank_num = 0
        for i in board:
            for j in i:
                if j == 0:
                    self.blank_num += 1
        self.difficulty = int(float(self.blank_num) / 81 * 10)
        self.board_correct = copy.deepcopy(self.board)
```

5.2 Solution

The Sudoku is essentially an constrained satisfied problem and we can use depth first search to find a solution. The steps can be seen as follows:

(1) **Create 27 recording sets and 1 blank list**:
Set $row[i], i \in [0,8]$ is used to record the numbers appearing in row i. Set $column[i], i \in [0,8]$ is used to record the numbers appearing in column i. Set $palace[i], i \in [0,8]$ is used to record the numbers appearing in palace i which is the certain 3×3 area in a board. List $blank$ is used to record all the blank positions.

(2) **Initialize recording sets and Blank list**:
Check every position of the input 9×9 matrix. If the value is 0 which means a blank,

add the position to the blank list. Else, record the value in the corresponding row, column, palace recording sets.

(3) **Depth-First-Search-1**:
If the blank list is empty meaning that all the positions have been assigned legal values, return True. Else, traverse the current blank list to find the blank position with the least number of optional values. If the blank we found has no optional values, return False.

(4) **Depth-First-Search-2**:
For every optional number for the blank position we found in the previous step, try to assign the value to this blank. Update the corresponding row, column, palace recording set and the blank list. Return to step (3).

- **Analyze**: There are mainly two methods to solve Sudoku: dancing list and depth first search. The first methods is faster theoretically but is also difficult to comprehend and implement. Therefore, I only realize the second method to solve Sudoku. In order to speed up the solving process, I use sets to store the appearance of digits in different rows, columns and palaces. To check the existence of one element, set is faster than lists, dictionaries.

The corresponding implemented codes can be seen as follows:

```
def solve(self):
    self.res_board = copy.deepcopy(self.board_correct)
    self.nums = {1, 2, 3, 4, 5, 6, 7, 8, 9}
    self.row = [set() for _ in range(9)]
    self.col = [set() for _ in range(9)]
    self.palace = [[set() for _ in range(3)] for _ in range(3)]
    self.blank = []

    for i in range(9):
        for j in range(9):
            num = self.board_correct[i][j]
            if num == 0:
                self.blank.append((i, j))
            else:
                self.row[i].add(num)
                self.col[j].add(num)
                self.palace[i//3][j//3].add(num)
    flag = self.dfs(0)
    if flag:
```

```
20          print('Solve Successfully!')
21          return True
22      else:
23          print('Wrong Input Board!')
24          return False
25
26  def dfs(self, n):
27      if len(self.blank) == 0:
28          return True
29      min_count = 10
30      for blank_pos in self.blank:
31          rest = self.nums - self.row[blank_pos[0]] - self.col[blank_pos[1]] - self.
                palace[blank_pos[0]//3][blank_pos[1]//3]
32          if len(rest) > min_count:
33              continue
34          i, j = blank_pos
35          min_count = len(rest)
36
37      rest = self.nums - self.row[i] - self.col[j] - self.palace[i//3][j//3]
38
39      if not rest:
40          return False
41      for num in rest:
42          self.res_board[i][j] = num
43          self.row[i].add(num)
44          self.col[j].add(num)
45          self.palace[i//3][j//3].add(num)
46          self.blank.remove((i, j))
47          if self.dfs(n+1):
48              return True
49          self.row[i].remove(num)
50          self.col[j].remove(num)
51          self.palace[i//3][j//3].remove(num)
52          self.blank.append((i, j))
53      return False
```

5.3 Correction

If there exists mistakes in the input 9×9 matrix, we need to correct them in order to make the Sudoku problem solvable and thus provide more enjoyable experiences for play-

ers.

I divide the mistakes into two type:

- **Format Mistake**: One number appears twice in the same row or column or palace.
- **Calculation Mistake**: There does not exist Format Mistakes in the Sudoku problem. However, we can not find a solution for this problem.

For the **Format Mistake**, we can simply delete one of the number that appears twice in the same row or column or palace which means that this position (i, j) becomes a blank and we try to solve the new Sudoku problem. Then there are only two outcomes:

1) We successfully find a solution and then we can put the value of position (i, j) in the solution to the position (i, j) in the original board. Therefore, the input board matrix is solvable.

2) We can not find a solution which means that there exists a Calculation Mistake in the current new board.

For the **Calculation Mistake**, because there are many possibilities and situations leading to this mistake, the way we can do is to modify the least number of values of positions of the current board to make it solvable. I achieve through the following ways:

(1) Traverse all the positions which are not blank currently to see that whether deleting one of them and making it a blank can lead to a successful solution. If yes, fill the correct value in this position of the original board and let the player to solve. If not, go to step (2).

(2) Traverse all the positions which are not blank currently to see that whether deleting two of them and making the two positions blanks can lead to a successful solution. If not, go to step (3).

(3) $\cdots$

The results of my correction can be seen in the last section. The mechanism can be seen more vividly and obviously through figure 13.

In most situations, only deleting one of the position can lead to a solvable Sudoku problem, so I only try to implement this kind of correction. The corresponding codes can be seen as follows:

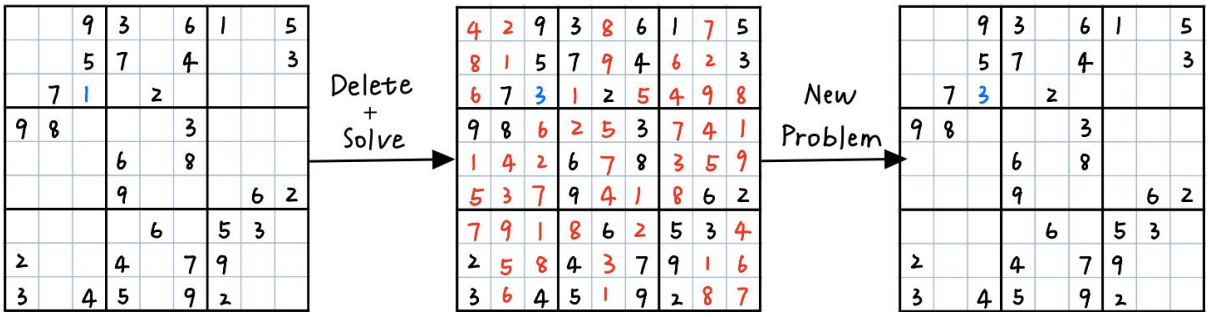


Figure 13: Example of Correction

```
def correct(self):
    self.nums_correct = {1, 2, 3, 4, 5, 6, 7, 8, 9}
    self.row_correct = [set() for _ in range(9)]
    self.col_correct = [set() for _ in range(9)]
    self.palace_correct = [[set() for _ in range(3)] for _ in range(3)]  # 3*3
    self.board_correct = np.zeros((9, 9), dtype=int)
    self.blank_correct = []
    self.pos_correct = None

    for i in range(9):
        for j in range(9):
            num = self.board[i][j]
            if num != 0:
                if (num in self.row_correct[i]) or (num in self.col_correct[j]) or (
                    num in self.palace_correct[i//3][j//3]):
                    self.blank.append((i, j))
                    self.pos_correct = (i, j)
                else:
                    self.row[i].add(num)
                    self.col[j].add(num)
                    self.palace[i // 3][j // 3].add(num)
                    self.board_correct[i][j] = num
    if self.solve():
        return True
    for i in range(9):
        for j in range(9):
            num = self.board_correct[i][j]
            if num != 0:
                self.board_correct[i][j] = 0
                flag = self.solve()
                if flag:
```

```
                    self.pos_correct = (i, j)
                    self.board_correct[i][j] = self.res_board[i][j]
                    return True
                else:
                    self.board_corret[i][j] = num
    return False
```

6 Comprehensive Model: Combination of Digit Detection and Sudoku Solution

6.1 Find Board in Image

To find the board area in a image, we have to do the following steps:

1) **Eliminate noisy**:

We use Gaussian Blur function to eliminate noisy in the image in order to find the right contours:

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (7, 7), 3)

thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2
    .THRESH_BINARY, 11, 2)
thresh = cv2.bitwise_not(thresh)
```

2) **Find Board**:

Find all the contours in the image and check whether there exists a big contour with four sides. If yes, this contour is just the board area we want to find.

```
cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE) #find all the contours
cnts = imutils.grab_contours(cnts)
cnts = sorted(cnts, key=cv2.contourArea, reverse=True) # sort then find the
    larger countour first
puzzleCnt = None
for c in cnts:
```

```
  # approximate the contour
  peri = cv2.arcLength(c, True) # calculate the length
  approx = cv2.approxPolyDP(c, 0.02 * peri, True)

  if len(approx) == 4: # it is a rectangle
    puzzleCnt = approx
    break
if puzzleCnt is None: #can not find a rectangle meaning no board found, return error
  raise Exception(("Could not find sudoku puzzle outline. Try debugging your thresholding and contour steps."))
```

3) **Normalize the Board**:

Transform the board area we obtained to top-down birds eye view in order to identify the digits more accurately:

```
puzzle = four_point_transform(image, puzzleCnt.reshape(4, 2))
warped = four_point_transform(gray, puzzleCnt.reshape(4, 2))
```

6.2 Identify Digits from Board and Form Sudoku Class

To Identify all the digits in the board image and finally form a Sudoku class, we need to do the following steps:

1) **Divide the board image** into 9×9 areas and try to extract digits from these areas through the next steps:

```
stepX = warped.shape[1] // 9 # x-length of a cell
stepY = warped.shape[0] // 9 # y-length of a cell
cellLocs = [] # locations of all cells
add = []
for y in range(0, 9):
  row = []
  for x in range(0, 9):
    # the starting and ending (x, y)-coordinates of the current cell
    startX = x * stepX
    startY = y * stepY
    endX = (x + 1) * stepX
```

```
12      endY = (y + 1) * stepY
13
14      cell = warped[startY+8:endY-4, startX+8:endX-4]
15      digit = extract_digit(cell, whether_debug) # extract digit from the area
```

2) **Detect whether the area contains a digit** by detecting the existence of contours and calculating its percentage of white pixels among all pixels. If the percentage is less than an threshold, we regard this cell as empty and return *None*. The corresponding codes can be seen as follows:

```
thresh1 = cell
thresh = cv2.threshold(thresh1, 140, 255, cv2.THRESH_BINARY_INV)[1] # change to
    binary image
cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
if len(cnts) == 0: # no contour means no digit
  return None
mask = np.zeros(thresh.shape, dtype="uint8")
cv2.drawContours(mask, cnts, -1, 255, -1)
(h, w) = thresh.shape
percentFilled = cv2.countNonZero(mask) / float(w * h)
if percentFilled < 0.005: # percentage is too low meaning noisy instead of digit
  return None
```

3) **Process and normalize the area to the format of our training database** by calling function *modify_img*. This function is used to process and normalize raw data images which is introduced and explained in the first section. Its purpose is to find the smallest square in the image that can contain the total digit and then we can resize it and add paddings to make it become standard form of out database. The corresponding code of the function can be seen as follows:

```
def modify_img(thresh):
  (h, w) = thresh.shape
  cnt_h = [0 for i in range(h)]
  cnt_w = [0 for i in range(w)]
  for i in range(0, h):
    for j in range(0, w):
```

```
      if thresh[i][j] != 0:
        cnt_h[i] += 1
        cnt_w[j] += 1

  h_min = h-1
  h_max = 0
  w_min = w-1
  w_max = 0
  for i in range(len(cnt_h)):
    if cnt_h[i] >= 1:
      h_min = min(h_min, i)
      h_max = max(h_max, i)
  for i in range(len(cnt_w)):
    if cnt_w[i] >= 1:
      w_min = min(w_min, i)
      w_max = max(w_max, i)
  length = max(h_max-h_min, w_max-w_min)

  h_max_new = (h_max + h_min + length + 1) // 2
  h_min_new = (h_max + h_min - length + 1) // 2
  w_max_new = (w_max + w_min + length + 1) // 2
  w_min_new = (w_max + w_min - length + 1) // 2

  new_thresh = thresh[h_min_new:h_max_new + 1, w_min_new:w_max_new + 1]
  if (h_min_new < 0 or h_max_new >= h):
    if (h_min_new < 0):
      new_thresh = thresh[0:h_max_new + 1, w_min_new:w_max_new + 1]
      new_thresh = np.pad(new_thresh, ((-h_min_new, 0), (0, 0)), 'constant',
          constant_values=0)
    else:
      new_thresh = thresh[h_min_new:h, w_min_new:w_max_new + 1]
      new_thresh = np.pad(new_thresh, ((0, h_max_new-h+1), (0, 0)), 'constant',
          constant_values=0)
  if (w_min_new < 0 or w_max_new >= w):
    if (w_min_new < 0):
      new_thresh = thresh[h_min_new:h_max_new + 1, 0:w_max_new + 1]
      new_thresh = np.pad(new_thresh, ((0, 0), (-w_min_new, 0)), 'constant',
          constant_values=0)
    else:
      new_thresh = thresh[h_min_new:h_max_new + 1, w_min_new:w]
      new_thresh = np.pad(new_thresh, ((0, 0), (0, w_max_new-w+1)), 'constant',
          constant_values=0)
```

```
45    if (new_thresh.shape == (0,0)):
46      return thresh
47    new_thresh = cv2.resize(new_thresh, (28, 28))
48    return new_thresh
```

3) **Input the processed image into our Net Model** and obtain the digit class of the input image.

```
model_path = './model_parameters/Alex_28_new.h5'
model  = load_model(model_path, custom_objects={"lrn": tf.nn.
    local_response_normalization})
pred = model.predict(roi).argmax(axis=1)[0]
if pred >= 10: # It is Chinese character and its class needs to minus 9
  pred = pred - 9
```

4) **Form** the 9×9 matrix and **create** the Sudoku object

```
puzzle = Sudoku(board=board.tolist())
```

6.3 Solve and Correct Sudoku

Sudoku is solved by calling *solve*() function in Sudoku class and is corrected through calling *correct*() function in Sudoku class. These two functions and their codes have been introduced and explained in the previous sections.

6.4 Result Exhibition

I use the black number to denote digits given by the input matrix, blue number to denote the digit after correction and red number to denote digits that are filled by the solution. The codes can be seen as follows:

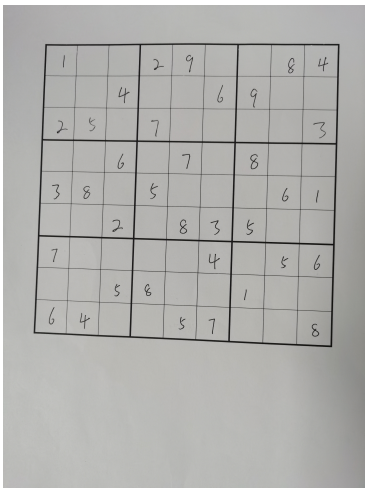
```
count = 0
for (cellRow, boardRow) in zip(cellLocs, puzzle.res_board):
  for (box, digit) in zip(cellRow, boardRow):
    startX, startY, endX, endY, flag = box

```

```
    textX = int((endX - startX) * 0.33)
    textY = int((endY - startY) * -0.2)
    textX += startX
    textY += endY

    if flag:
      if puzzle.pos_correct is None:
        cv2.putText(puzzleImage, str(digit), (textX, textY), cv2.FONT_HERSHEY_SIMPLEX
            , 0.9, (0, 0, 0), 2)

      elif count == (puzzle.pos_correct[0]*9 + puzzle.pos_correct[1]):
        cv2.putText(puzzleImage, str(digit), (textX, textY), cv2.FONT_HERSHEY_SIMPLEX
            , 0.9, (255, 0, 0), 2)
      else:
        cv2.putText(puzzleImage, str(digit), (textX, textY), cv2.FONT_HERSHEY_SIMPLEX
            , 0.9, (0, 0, 0), 2)
    else:
      cv2.putText(puzzleImage, str(digit), (textX, textY), cv2.FONT_HERSHEY_SIMPLEX,
          0.9, (0, 0, 255), 2)
    count += 1
```

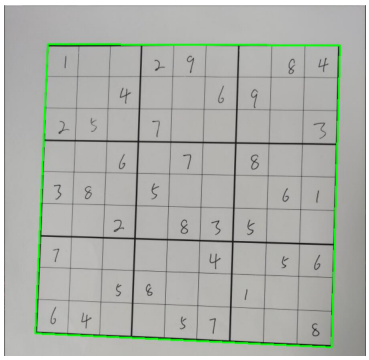
The vivid examples of my output can be seen in the next sections.

7 Result of Different Steps for Solving Test Cases

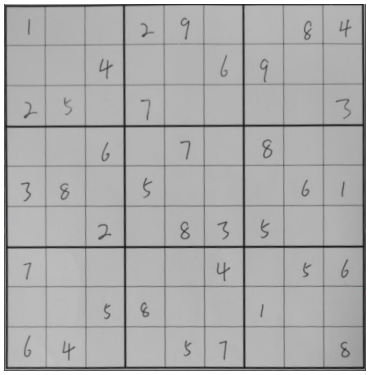
7.1 Case 1-1



(a) Original Image



(b) Detect Board Contour



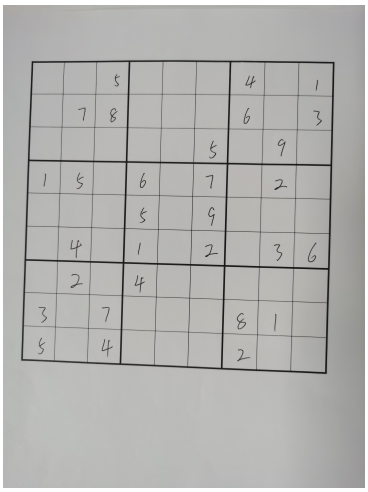
(c) Top-Down View of Board

(d) Digit Identify Image

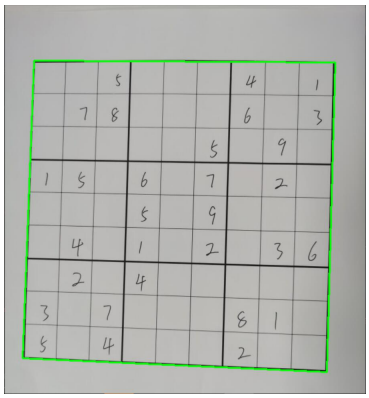
(e) Result

Figure 14: Steps for 1-1

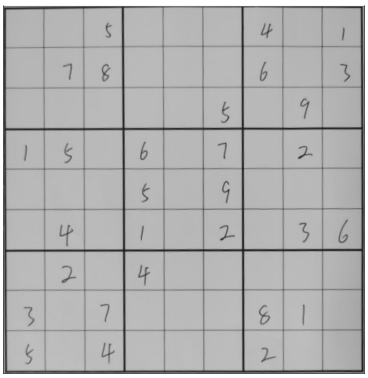
7.2 Case 1-2



(a) Original Image



(b) Detect Board Contour



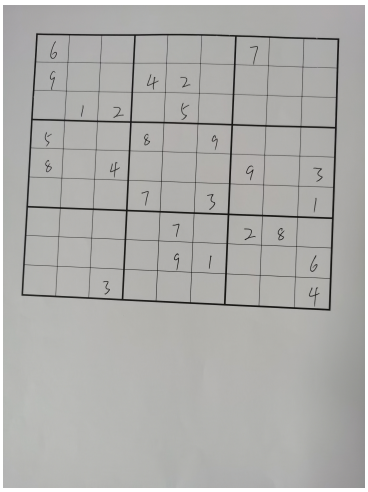
(c) Top-Down View of Board

(d) Digit Identify Image

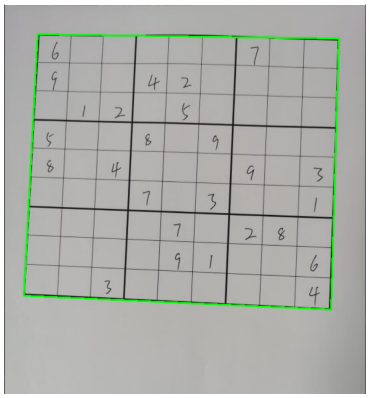
(e) Result

Figure 15: Steps for 1-2

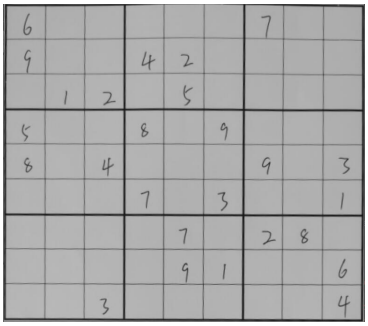
7.3 Case 1-3



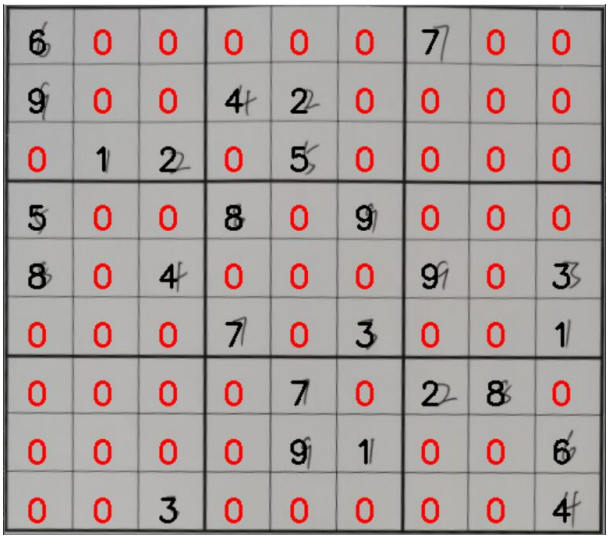
(a) Original Image



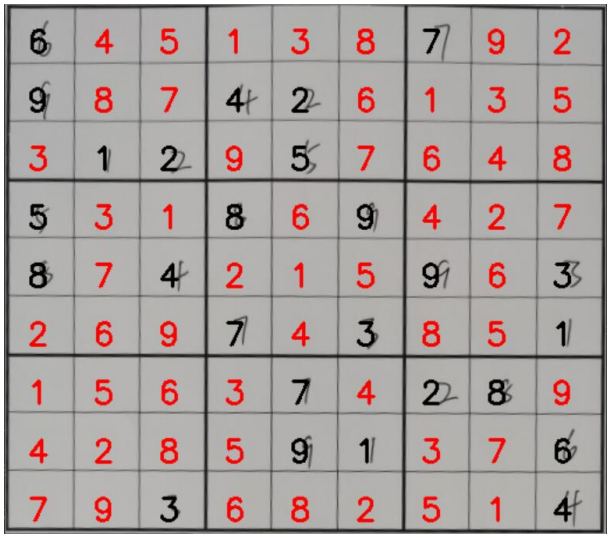
(b) Detect Board Contour



(c) Top-Down View of Board



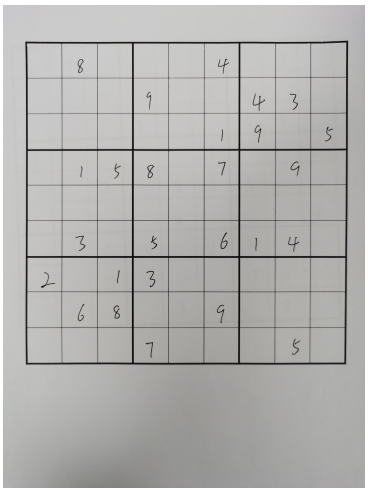
(d) Digit Identify Image



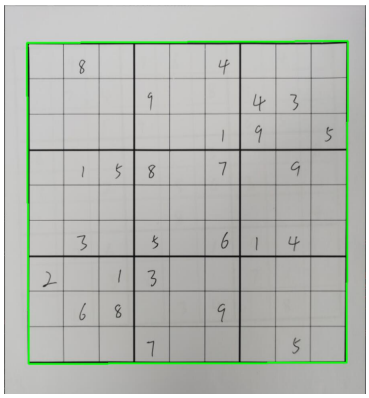
(e) Result

Figure 16: Steps for 1-3

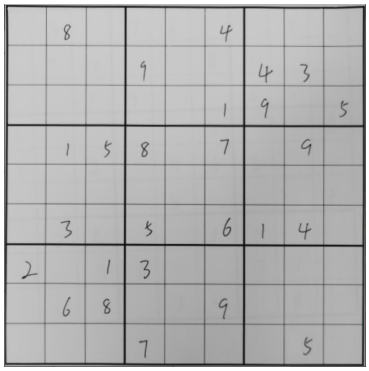
7.4 Case 1-4



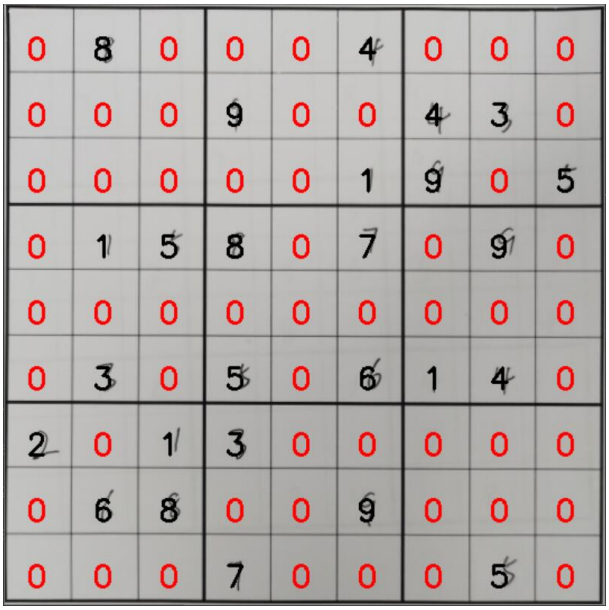
(a) Original Image



(b) Detect Board Contour



(c) Top-Down View of Board



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

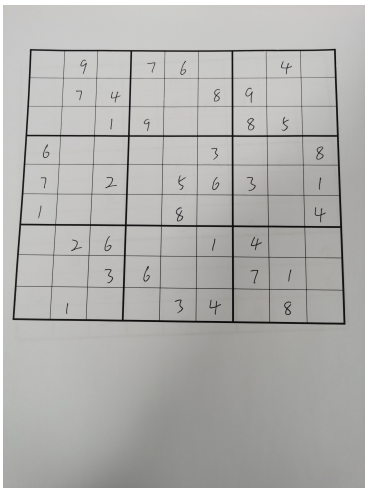
(d) Digit Identify Image

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

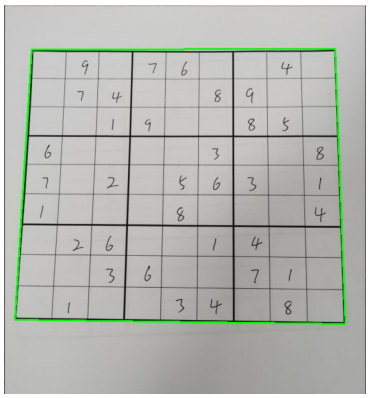
(e) Result

Figure 17: Steps for 1-4

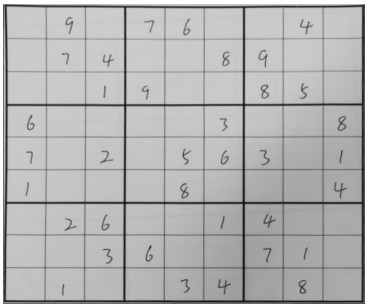
7.5 Case 1-5



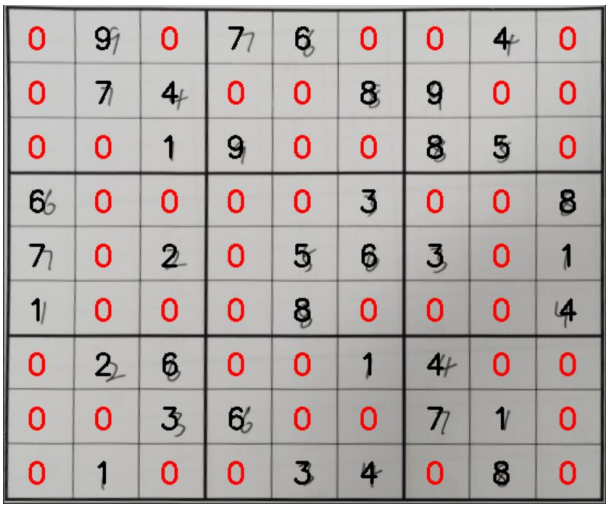
(a) Original Image



(b) Detect Board Contour



(c) Top-Down View of Board

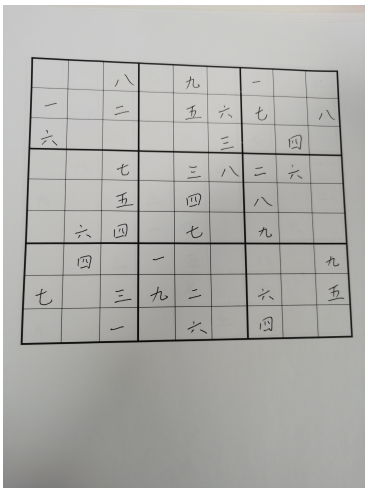


(d) Digit Identify Image

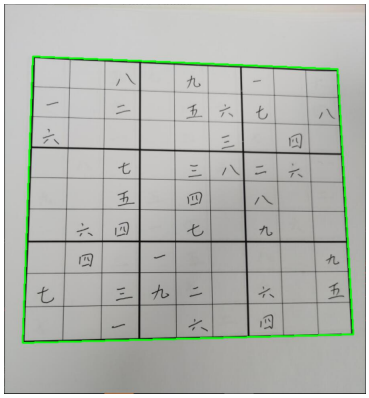
(e) Result

Figure 18: Steps for 1-5

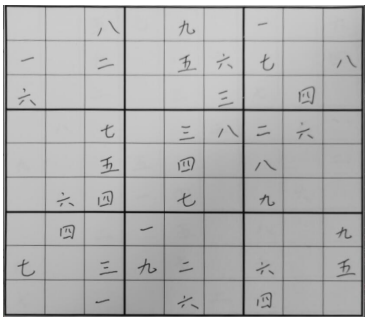
7.6 Case 2-1



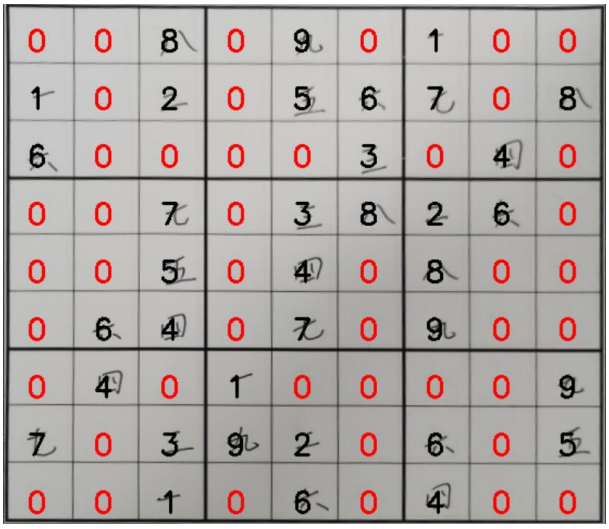
(a) Original Image



(b) Detect Board Contour



(c) Top-Down View of Board

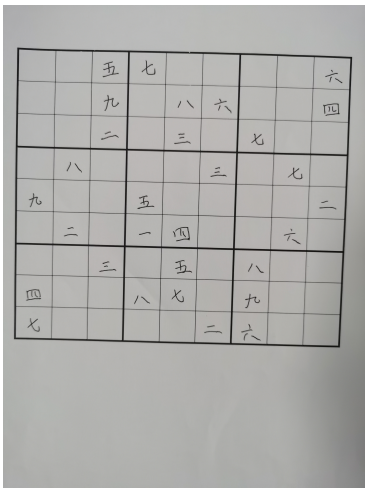


(d) Digit Identify Image

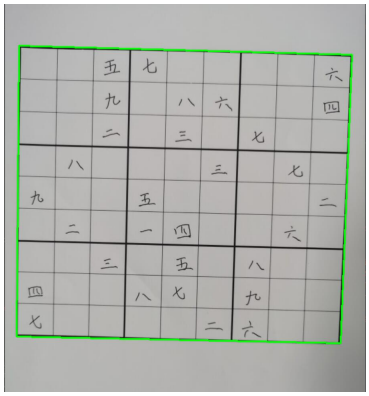
(e) Result

Figure 19: Steps for 2-1

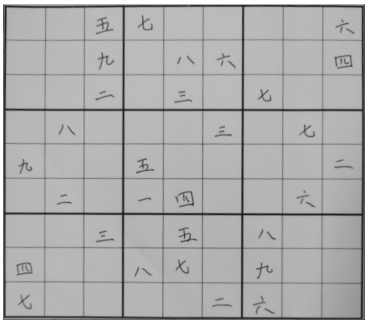
7.7 Case 2-2



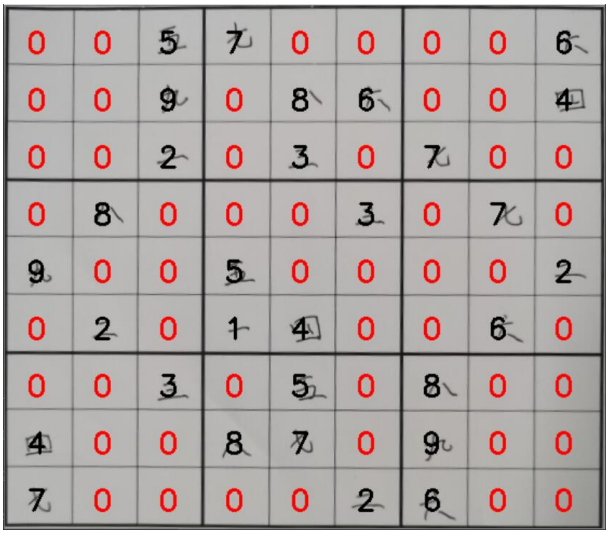
(a) Original Image



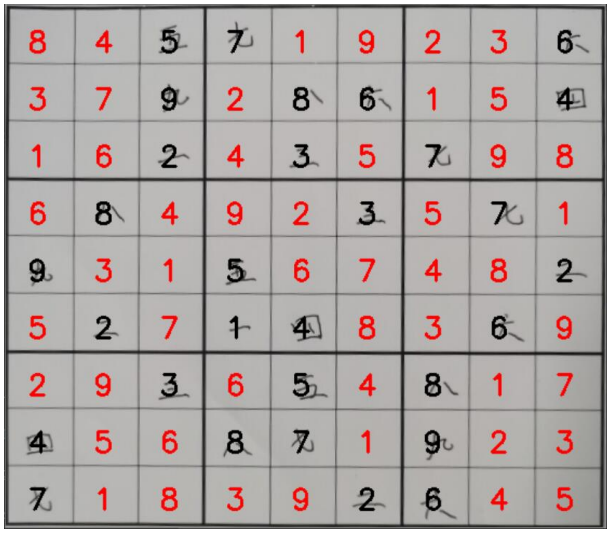
(b) Detect Board Contour



(c) Top-Down View of Board



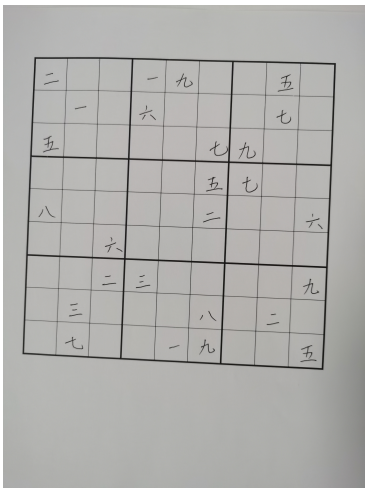
(d) Digit Identify Image



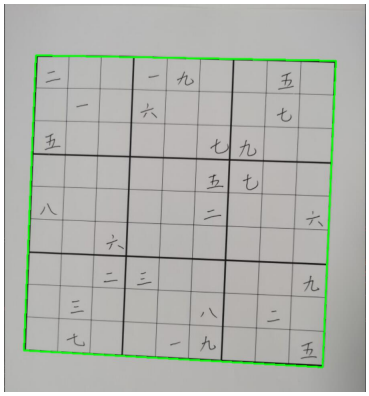
(e) Result

Figure 20: Steps for 2-2

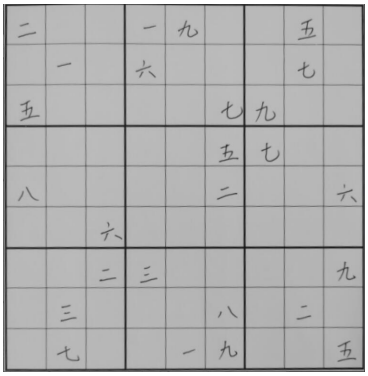
7.8 Case 2-3



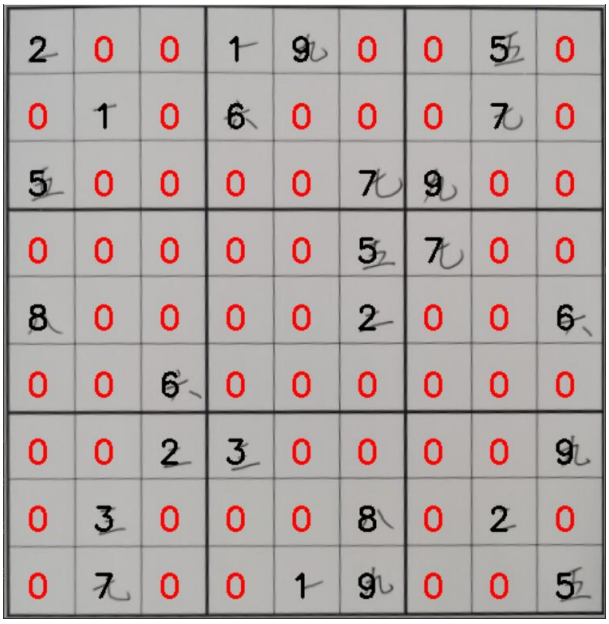
(a) Original Image



(b) Detect Board Contour



(c) Top-Down View of Board

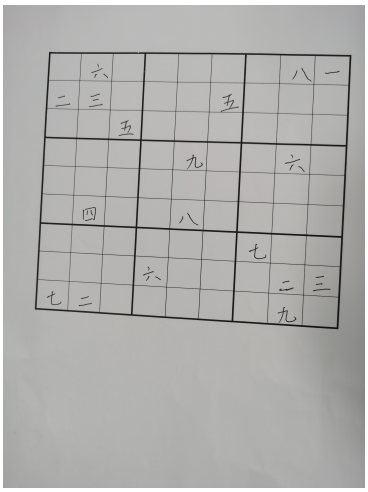


(d) Digit Identify Image

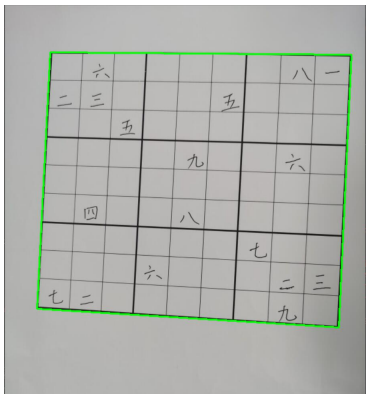
(e) Result

Figure 21: Steps for 2-3

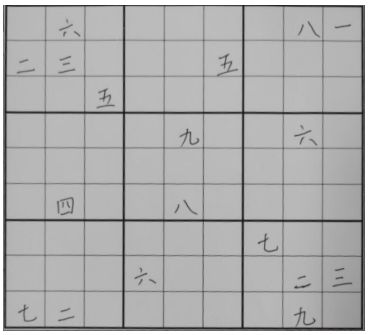
7.9 Case 2-4



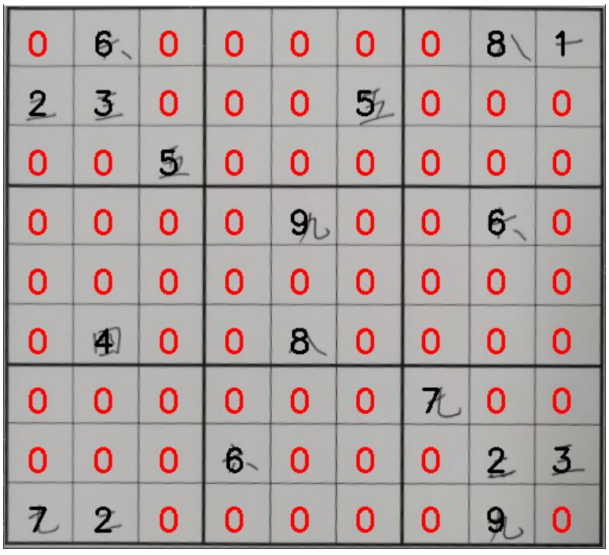
(a) Original Image



(b) Detect Board Contour



(c) Top-Down View of Board



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

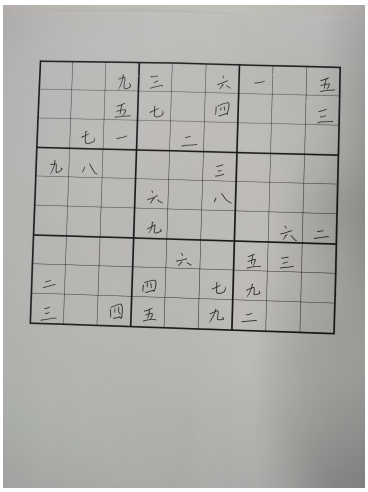
(d) Digit Identify Image

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

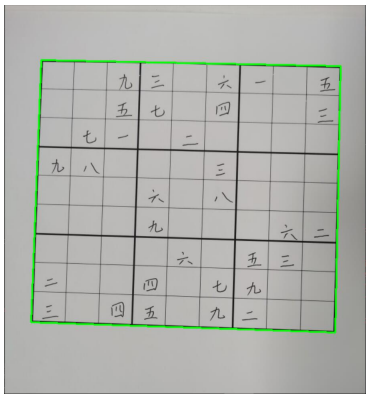
(e) Result

Figure 22: Steps for 2-4

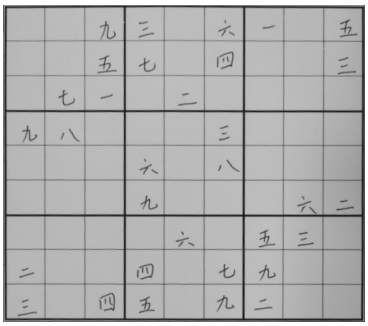
7.10 Case 2-5



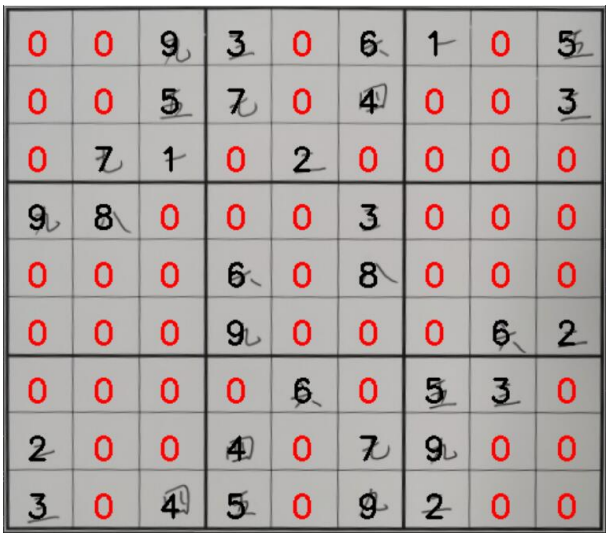
(a) Original Image



(b) Detect Board Contour



(c) Top-Down View of Board



(d) Digit Identify Image

(e) Result

Figure 23: Steps for 2-5

8 Correction for Cases

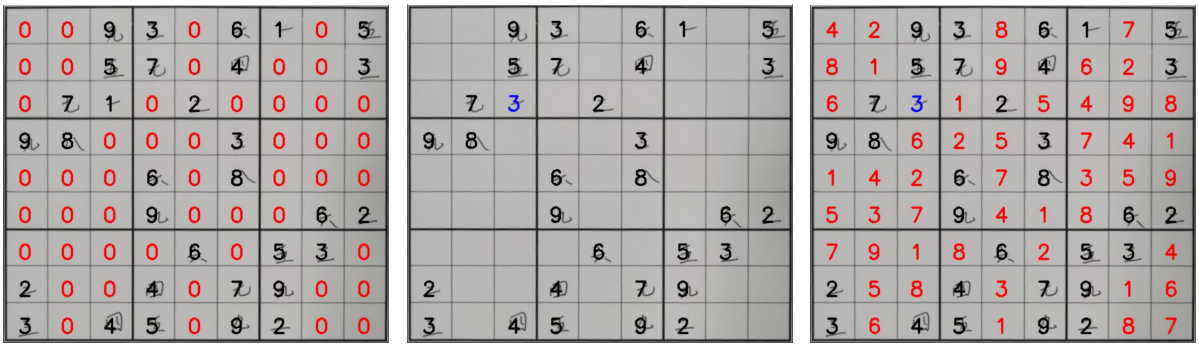
This section exhibits our correction results for Case 1-5 and Case 2-5.

8.1 Case 1-5

(a) Original Board (b) Correct Board (c) Result Board

Figure 24: Correction Steps for Case 1-5

8.2 Case 2-5



(a) Original Board (b) Correct Board (c) Result Board

Figure 25: Correction Steps for Case 2-5

9 Conclusion and Reflect

In general, there are three main procedures when we implement a project: data process, model training and testing. All these three parts are of great significance and are tightly related to our model's performance.

1) **Data Process**:

The normalized format of data points is necessity. From the results and comparison in the first section, we can see that using data with normalized format can improve the accuracy from 0.70+ to 0.94+ compared with directly using the raw data. This

is because our neural network extracts features based the pixels in an image. If our data points do not have the same format, one feature extraction layer might extract another different kind of feature of the input image which will surely deteriorate the performance of our model.

2) **Model Training**:

Because we do not know the characteristics of the task in advance, we need to try different neural networks and many parameters in this procedure in order to extract the most important features of the data. From our experiments and results of different net models, we can see that a fitter model can improve the performance when testing with the accuracy increasing from 90%+ to 97%+.

3) **Testing**:

In some situations, the testing data is really different from the training data because they might be from different sources and have different distributions. However, in our project, because the size of each data point is relatively small 28×28, the difference between various data points can not be really large. Therefore, my model performs perfectly in the testing cases which is shown in the last two sections.