ME5411 Final Project Report

AY2022/2023, SEMESTER 2

14/APR/2023

Group 13: Song Jiaxuan (A0268452X) Xue Haoyan (A0268454U)

Zhang Zihao (A0268338R)

NATIONAL UNIVERSITY OF SINGAPORE

Table of contents

Chapter 1: Pre-processing

1.1 Overview

In this session, our goal is to extract the letter outlines from the original image (Shown in Figure 1.1) by utilizing various image pre-processing methods, such as smoothing, cropping, thresholding, segmentation. Since the background contains a significant amount of noise in the form of small white circles. To address this issue, we will apply spatial domain filtering techniques first to smooth the image and reduce the negative impact of the noise.

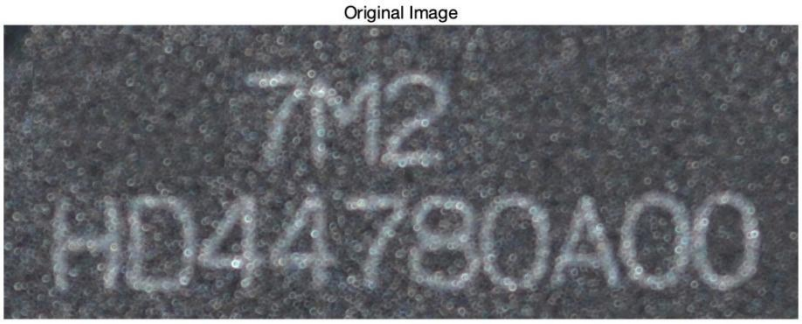

Figure 1.1 original image

1.2 Smoothing

1.2.1 Introduction

Spatial filtering is a widely used image processing technique that operates directly in pixel space. It is typically employed to remove noise, smooth images, and enhance image details. This method involves filtering each pixel in an image, with filter weights and sizes that can be tailored to specific needs. As shown in Figure 1.2, spatial filters can be classified into two categories: smoothing filters and sharpening filters. Smoothing filters can further be classified as linear or nonlinear, depending on their characteristics. Sharpening filters, on the other hand, commonly use differential operators, such as the Sobel filter and Laplacian filter. In this assignment, the averaging mask is an example of a linear smoothing filter, while the rotating mask is an example of a nonlinear filter.

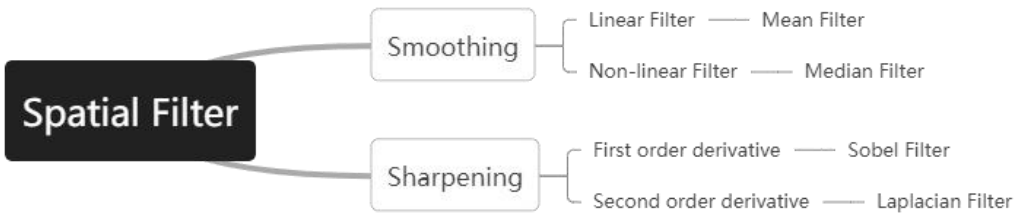

Figure 1.2 Classification of spatial filter

In addition to the requested averaging mask and rotating mask, we also explored other spatial filtering techniques, such as the median filter and Laplacian filter. We conducted a comparison of the processed results using masks of different sizes to evaluate their performance. We will analyze it in detail with our own codes and screenshots below.

1.2.2 Mean filter

Mean filter, also known as the averaging filter, is a simple image processing technique for removing noise and blurring an image. This filter is implemented by convolution with a kernel of equal weights. The resulting pixel value is obtained by summing the products of each pixel value in the kernel and its corresponding image pixel, and then dividing the total by the number of pixels in the kernel. This operation effectively replaces each pixel value in the image with the average of its neighboring pixel values, resulting in a smoother, less noisy image.

Figure 1.3 illustrates our flow to apply the mean filter. The function we have developed has three input parameters: the input image, the mask size, and the boundary processing method. The available boundary processing options are 'zero' and 'replicate'. We initially considered this issue because we were concerned that pixels at the image boundaries might be affected by fewer neighboring pixels, resulting in outliers that could potentially impact the image processing results. However, we later concluded that it was not necessary to consider boundary processing in the subsequent image pre-processing steps. This is because our objective in this part of the task is to extract the outline of letters in the image, which is unrelated to the pixels located at the image boundary.

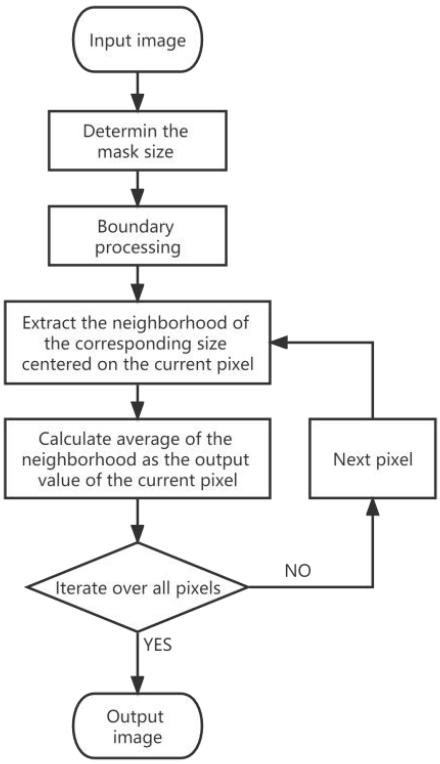

Figure 1.3 The flow chart of mean filter

Figure 1.4 Processing results of the 3×3 mean filter

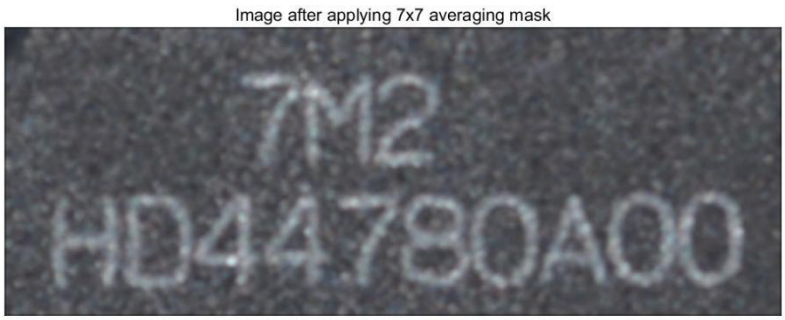

Figure 1.5 Processing results of the 7×7 mean filter

Figure 1.4 and Figure 1.5 depict the respective smoothing outcomes obtained using a 3×3 mask and a 5×5 mask with a mean filter. When the mask size is 3×3, the resulting image bears a resemblance to the original image illustrated in Figure 1.1. However, upon increasing the mask size to 7×7, it is apparent that the noise of the image is reduced, albeit at the expense of increased blurring.

Based on our experiments, we can conclude that the larger the mask size of the mean filter, the more blurred the image we get. Therefore, it is important to choose an appropriate mask size that can effectively remove unwanted details while preserving the important features of the object of interest.

1.2.3 Median filter

The median filter is a popular nonlinear filter that replaces the value of each pixel with the median value of the grayscale in its surrounding pixel field. Unlike mean filter which obtains the output value by a linear combination of pixel points and filter coefficients, the median filter does not satisfy the linear superposition principle. This feature makes it excellent for reducing certain types of random noise, such as pretzel noise, which is superimposed on the image in the form of black and white dots.

Based on the principles of the median filter, we implemented our own Matlab code, and the flowchart of the code implementation is shown in Figure 1.6. The flow chart of the median filter implementation is similar to that of the mean filter. However, the main difference lies in the processing within the current pixel neighborhood for each loop. Instead of computing the average of the pixel values in the neighborhood, the median filter uses statistical ordering to find the median value of the grayscale in the pixel field. This fundamental difference makes the median filter a nonlinear filter and distinguishes it from the mean filter.

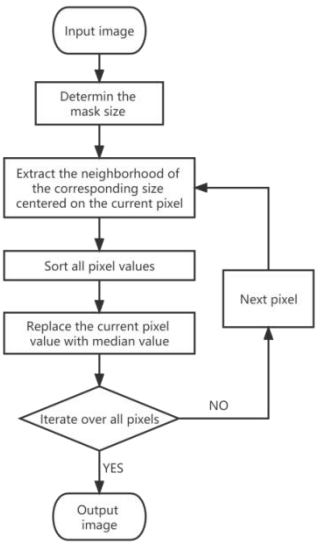

Figure 1.6 the flow chart of median filter

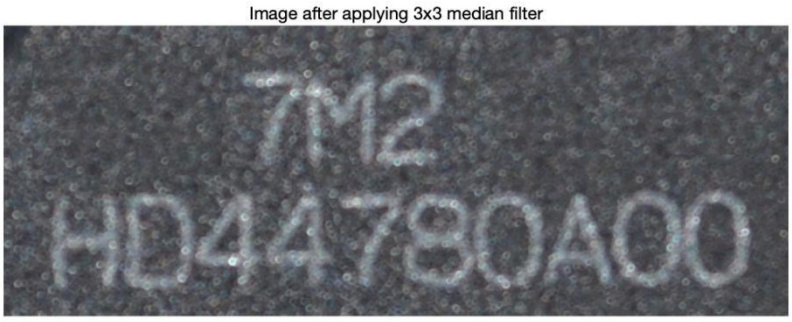

Figure 1.7 Processing result of the 3×3 median filter

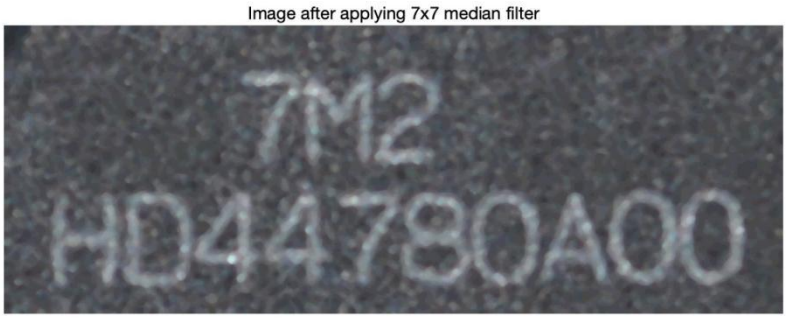

Figure 1.8 Processing result of the 7×7 median filter

Comparing Figure 1.7 and Figure 1.8, we observe that as the mask size increases, the output image is also becoming more blurred, which is similar to the mean filter. However, comparing Figure 1.8 and Figure 1.5, with the same size mask, the median filter can better retain the sharpness and details of the letters while removing the noise, and the processing effect is better than the mean filter.

1.2.4 Smoothing using a rotating mask

Smoothing using a rotating mask is a non-linear image filtering technique that is applied to a digital image in a rotating manner. The goal of this technique is to smooth out the image while preserving its edges.

Images processed with rotational masks are often considered to be sharpened because the rotational mask process actually works by enhancing the image edges. It involves applying a mask to the image and rotating the mask at each pixel location. The pixel value at each location is then updated based on the average brightness of the pixels within the mask with minimum variance. This process is repeated for all rotation angles, resulting in a smoothed image with reduced noise while retaining

sharp edges.

The flow chart of rotating mask is shown in Figure 2.8, and we implemented our own Matlab code. Due to the high time complexity associated with smoothing using a rotation mask, we opted to convert the RGB image to a gray image before applying the filter. This approach saves time by eliminating the need to calculate results for each of the three channels.

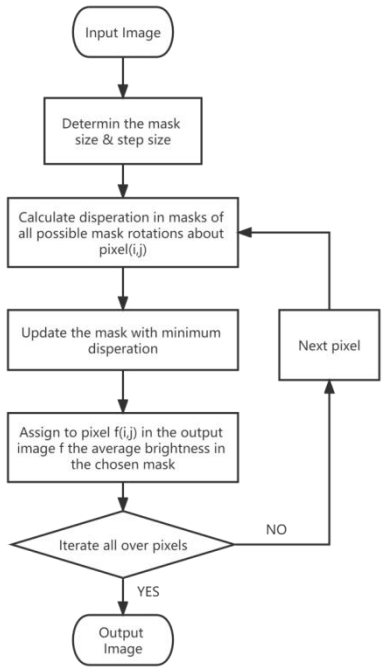

Figure 1.9 the flow chart of rotating mask

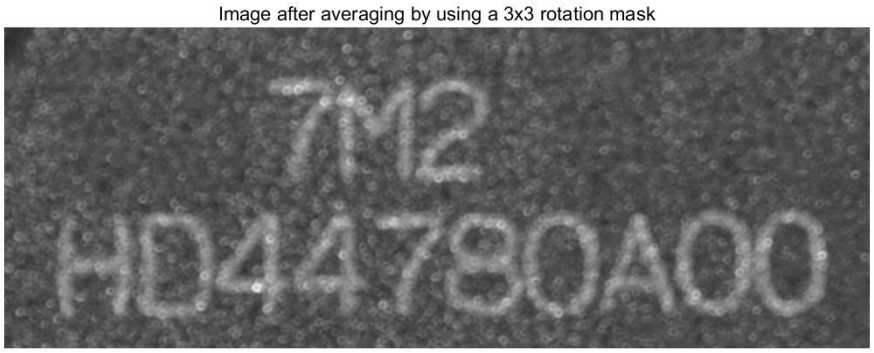

Figure 1.10 Processing result of the 3x3 rotating mask

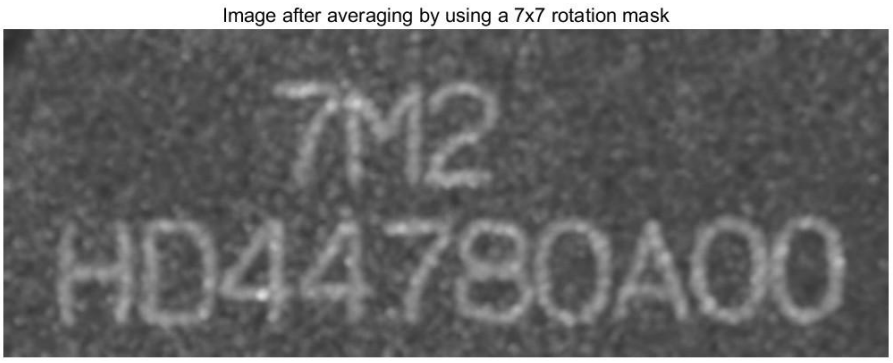

Figure 1.11 Processing result of the 7×7 rotating mask

The output images of applying rotating mask are displayed in Figure 1.10 and 1.11. In theory, the image is sharpened to a certain extent after applying the rotational masking process. However, in this particular task, the resulting image is not significantly different from that obtained using a mean filter of the same size and magnitude. We hypothesize that the presence of a considerable amount of noise in the background of the original image, with similar brightness as the letter outlines, significantly impacted the effectiveness of the rotational mask.

1.2.5 Laplacian filter

The Laplacian filter is a second-order derivative filter, meaning it calculates the rate of change of intensity in an image, allowing it to identify edges and other high-frequency features. Therefore, it is commonly used in image processing for edge detection and image sharpening. It highlights the areas of rapid intensity change in an image and can be implemented using a discrete convolution kernel, usually a 3×3 or 5×5 matrix.

Since Laplacian operator is a differential operator that emphasizes sudden changes in gray levels in the image, and it does not emphasize areas of slowly changing gray levels. As a result, applying Laplacian filter alone can produce images that superimpose light gray borders and mutation points into a dark background, making them difficult to interpret.

To address this issue, a common approach is to superimpose the original image and the Laplacian image together. This process helps to recover the background properties and maintain the results of the Laplacian sharpening process, resulting in a sharper image that is easier to interpret. In this task, we use a Laplacian filter with a negative central coefficient, which requires us to subtract the transformed image from the original image in order to obtain the sharpened result. Additionally, the flow chart of the Laplace edge detection algorithm shown in Figure 1.12.

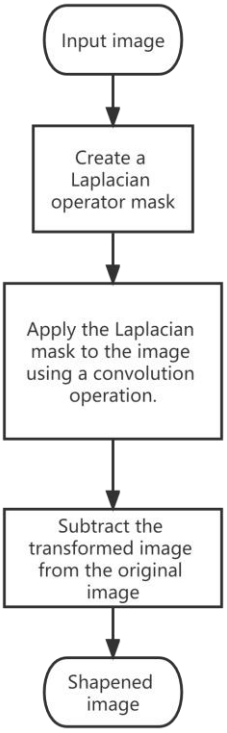

Figure 1.12 the flow chart of Laplacian filter

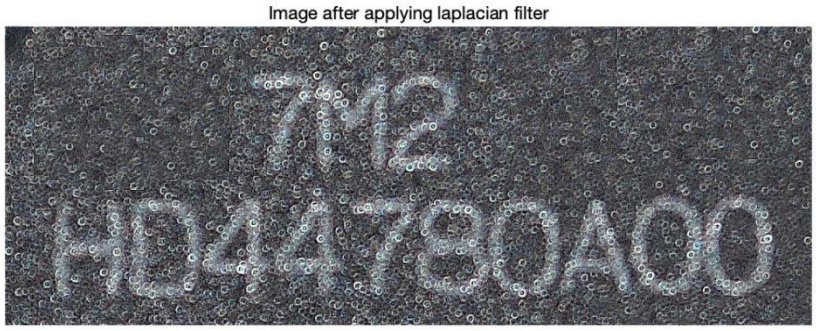

Figure 1.13 Processing result of Laplacian filter

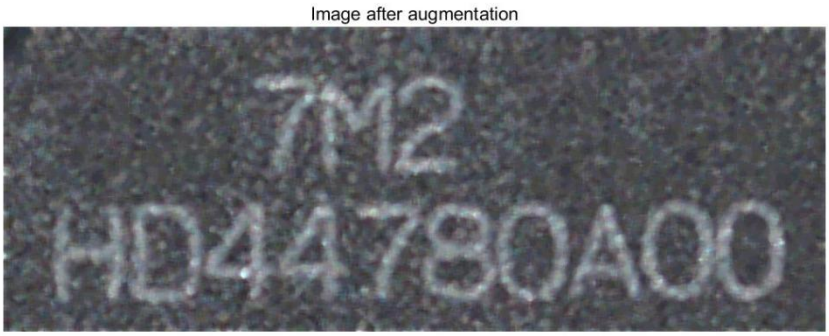

Figure 1.14 Image augmentation by multiple methods

Figure 1.13 displays the results of applying the Laplace filter to the original image. Upon comparing it with the original image in Figure 1.1, it is evident that the outline of letters in the sharpened image is clearer, and the details are more enhanced. However, as a trade-off, the noise in the background also becomes more severe, which might impact our subsequent processing steps.

Based on our previous experience with image smoothing, it was natural to consider combining enhancement methods for image processing. As a result, we applied median filtering to the sharpened image, which resulted in Figure 1.14. The processed image in Figure 1.14 shows significant improvement compared to the image obtained through a single processing method. This is due to the complementary nature of different image processing techniques.

1.3 Sub-image

1.3.1 Introduction

The goal of task 3 is to extract a sub-image from the original image that contains only the middle line – HD44780A00. To do this, the image is first split into sub-images that can be processed separately.

Digital image processing refers to the use of computer algorithms to manipulate digital images. Since computer images are represented as two-dimensional arrays of pixels, we can divide the array in half to extract the sub-image as required by the task.

1.3.2 Methodology

To begin, we read in the image that we want to process. Next, we determine the length and width of the image to obtain the maximum horizontal and vertical coordinates of the image matrix. Finally, we halve the number of rows in the matrix and take the bottom half to generate the sub-image that contains the middle line - HD44780A00.

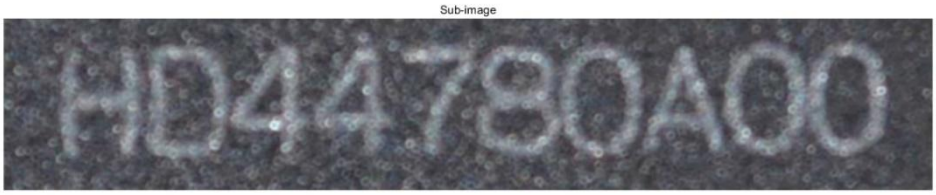

Figure 1.15 Sub-image

1.4 Thresholding

1.4.1 Introduction

Task 6 aims to generate a binary image from the sub-image obtained in Task 3 using thresholding.

In digital image processing, an image is represented as a two-dimensional array of pixels, with each pixel containing a color value that represents the color displayed at that location. For black and white images, each pixel has a grey value, usually expressed as an integer between 0 and 255, where 0 represents black and 255 represents white. The values in between represent varying degrees of grey.

The RGB (Red, Green, Blue) color model is commonly used to represent color images, where each pixel is represented by three components - red, green, and blue - each with a value between 0 and 255. Thus, a color image can be viewed as a two-dimensional matrix, where each element is a three-dimensional vector.

In a binary image, each pixel point can have only two possible values, usually black and white (0 and 1), making it much simpler than a grey-scale or color image. Binary images are commonly used in various areas such as image segmentation, morphological operations, and character recognition.

To convert the sub-image to a binary image, we first need to convert the color image to a grayscale image. This can be done by averaging the values of the red, green, and blue channels for each pixel, or by using the conversion formula:

$$Grayscale\ image\ pixel\ value = 0.299 * R + 0.587 * G + 0.114 * B$$

Once the image has been converted to grayscale, we need to choose a threshold value for binarization. This threshold value is used to set the pixel values below it to 0 and those above it to 1. The methodology section explains the process for choosing the appropriate threshold value.

1.4.2 Methodology

I. Grayscale

In MATLAB, converting an image to grayscale can be done directly using MATLAB's built-in functions to convert to grayscale images:

```
Img = rgb2gray(img)
```

To convert the original image to grayscale, we follow these steps:

a) Obtain the dimensions of the image, m rows and n columns, where l = 3 represents the three color channels (red, green, and blue) of each pixel.

b) Create a new m*n matrix to store the converted gray values for each pixel.

c) Iterate over each pixel, calculating its gray value using the formula: gray value = 0.299 * R + 0.587 * G + 0.114 * B, where R, G, and B are the color values of the pixel in the red, green, and blue channels, respectively.

d) Assign the gray value to the corresponding pixel position in the new matrix, ensuring that each gray value is represented as an 8-bit unsigned integer (unit8) between 0 and 255, the range of valid gray values.

II. Selecting threshold

We have the option of manually selecting the threshold, but doing so would require adjusting it through trial and error. By using an adaptive thresholding algorithm, the threshold can be automatically adjusted based on the overall brightness and contrast of the image. For this reason, we have opted to use the adaptive thresholding algorithm:

$$level = adaptthresh(I, sensitivity)$$

We utilize the adaptive thresholding algorithm manually to determine the most appropriate threshold. The following method is used:

a) Obtain the maximum and minimum thresholds of the image and calculate the average of these values as the initial threshold, which is denoted as T1. We then proceed to the next step and enter a loop.

b) The loop process: using the current threshold as a divider, we iterate through the gray value of each pixel. For each iteration, we calculate the average of the gray value of all pixels greater than this threshold (denoted as z1) and the average of the gray value of all pixels less than this threshold (denoted as z2). If the average of z1 and z2 is equal to the current threshold, then the iteration is complete, and we exit the loop. Otherwise, we adjust the threshold and repeat the loop until the threshold no longer changes.

c) During each iteration process, we adjust the threshold, leading to an increase in the difference between the foreground and background. This process continues until the threshold converges to an optimal value that maximizes the difference between the foreground and background. As a result, the accuracy of the binarization is improved to a certain extent.

```
Initial threshold T1:128
Updated threshold T2:108
```

Figure 1.16 Threshold results

This process can be represented by the following flow chart:

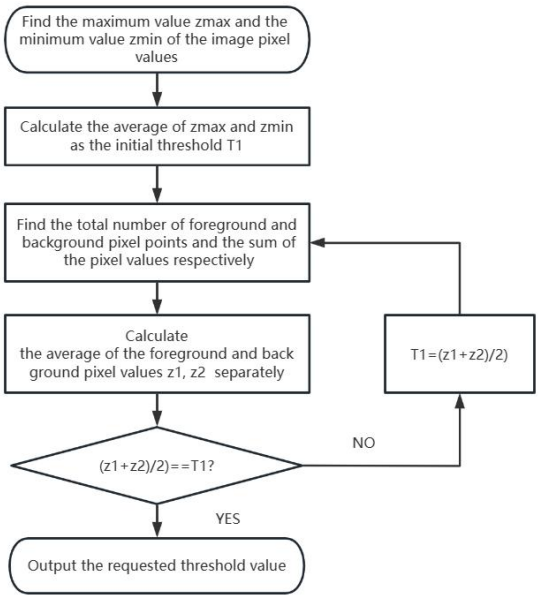

Figure 1.17 Flow chart of selecting threshold

III. Convert to binary image

In MATLAB, the conversion of a grayscale image to a binary image can be done using the following built-in MATLAB functions:

img_binary = imbinarize (I, T1/255)

Where I represents the image matrix and the threshold needs to be transformed in the range 0-1.

After finding the optimal threshold, we manually convert the grayscale image to a binary image, the method used here is:

a) Create an empty image matrix of the same size.

b) Iterate through each pixel point of the original grayscale image, assigning a value of 1 to the corresponding position in the new image if the grayscale value is greater than that point, and a value of 0 to the corresponding position in the new image if the grayscale value is less than that point, outputting the new binary image.

c) Binary images generally use 0 and 1 to represent pixel values rather than 0 and 255, because binary images have only two pixel values and using 0 and 1 is a more accurate representation of the image information and easier to process and calculate.

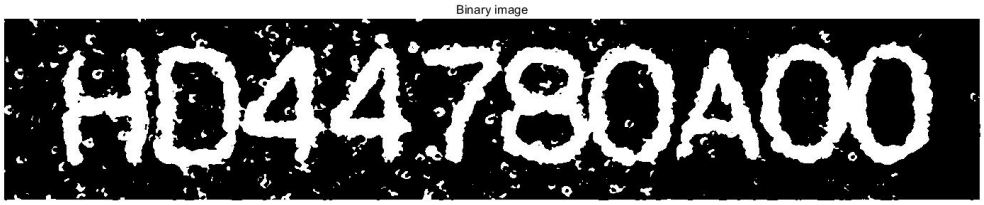

Figure 1.18 Binary image

1.4.3 Problems

After completing the binarization process, we output a single image that is the same size as the original image. The output image displays the same binary image, but rather than being arranged in three columns, it is presented as a single image.

Figure 1.19 The Problem in binarization process

The cause of the error was a misunderstanding of the image format. The image provided for the task was a color image, but it was incorrectly interpreted as a grayscale image. As a result, the process of converting the color image to a grayscale image was not performed before carrying out the binarization process.

After performing the binary convolution process, the loop method stitched together the three components of the color image to create a grayscale image. However, the resulting image was three times the size of the original image and displayed as three separate images joined together. To address this issue, it is necessary to convert the color image to a grayscale image before performing the binarization process.

1.5 Extracting outlines

1.5.1 Introduction

The aim of task 5 is to determine the outline of characters of the image. We manually segment the adhered letters, then mark the connected region of the image, and later remove the noise points according to the number of pixel points. Then the connected

region is marked again, the image is edge detected using the Sobel operator, and finally the position of the character outline is determined based on the edge position, and the character outline is displayed.

1.5.2 Methodology

I. Convert the image to a binary image for subsequent processing

As some characters in the image are linked, we located the coordinates of these connections and manually segmented some of the linked characters. The resulting processed images are displayed below, and we separated the linked characters for subsequent detection of connected areas.

Figure 1.20 Split image

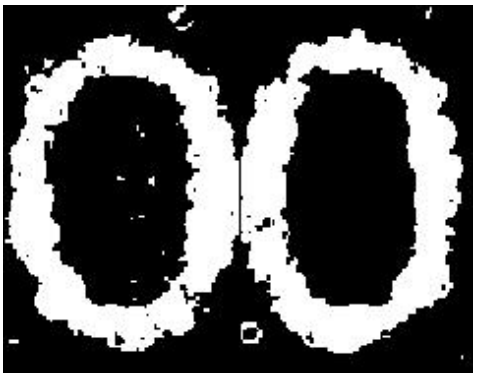

Figure 1.21 Divide 0 and 0

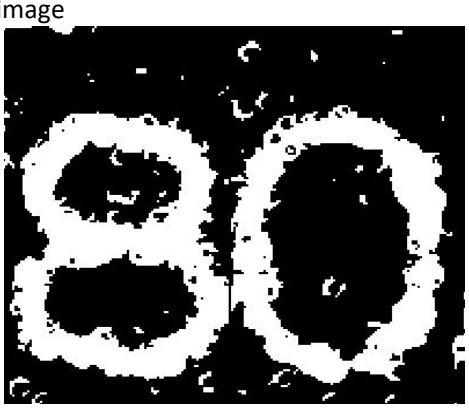

Figure 1.22 Divided 8 and 0

II. Label the connected area and derive the set of coordinates

We developed a function to label the connected areas, which returns the set of coordinates containing the pixels of the connected area, the total number of areas, and the labeled image. After experimenting with various search algorithms, we settled on using a queue scan algorithm for connectivity region marking.

Initially, we attempted to use Depth-first search (DFS) to search every pixel and determine whether it belongs to a connected region. However, the DFS algorithm took longer to execute as it explores deeper along the search path until it can no longer search, and then goes back to the previous node to continue the search. Additionally, the algorithm's performance was even slower due to the multiple recursive algorithms

used in the code.

In practice, BFS is generally more suitable for handling large-scale data than DFS. BFS uses queues to store nodes to be accessed without storing information about all accessed paths, which results in relatively smaller memory requirements when working with larger image sizes. Therefore, we ultimately decided to use the BFS algorithm to improve the efficiency of our labeling function.

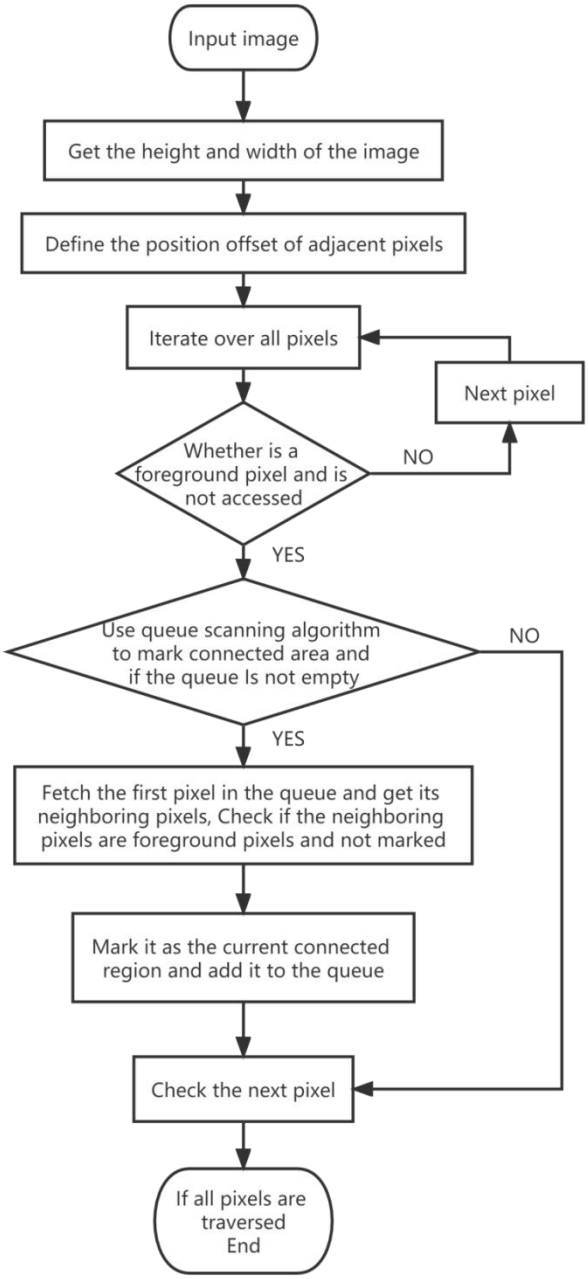

Figure 1.23 Flow chat of the queue scan algorithm

III. Remove the noisy point areas

There are many white noise spots in the image after marking, as shown in the

following figure. Therefore, we found the areas with less than 1000 pixels and removed the noisy point areas. Here is a comparison of the before and after treatment.

Figure 1.24 Image before noise removal

Figure 1.25 Image after noise removal

Since the noise is removed, the labeled numbers of the connected regions are now discrete and not continuous, such as 52, 89, 320, 476, ... Therefore, the connected region is labeled again for the image after noise removal, and a new connected region is obtained along with the corresponding label, currently, the numerical numbers of the label are continuous, such as 1, 2, 3, 4 , ...

IV. Sobel operator edge detection for grayscale images

We choose to use Sobel operator to detect the edge and it can detect edges by calculating the gradient of the grayscale of pixels in an image. This method is robust to noise. The Sobel operator consists of two 3x3 convolution kernels, one for detecting edges in the horizontal direction (Gx) and the other for detecting edges in the vertical direction (Gy). We set Gx = [-1 0 1; -2 0 2; -1 0 1] and Gy = [-1 -2 -1; 0 0 0; 1 2 1]. Next, we apply these two convolution kernels to the grayscale image L to compute the gradients in the horizontal and vertical directions, respectively. Then, we obtain the edge intensity by calculating the square root of the sum of squares of the horizontal and vertical gradients.

Finally, we set the threshold at 0.4 for determining edges. Pixels with edge intensities below the threshold are considered non-edge, and pixels with edge intensities greater than or equal to the threshold are considered edge. Below is the picture after edge detection.

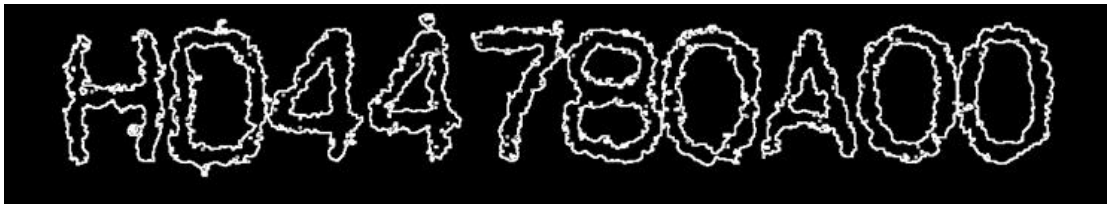

Figure 1.26 Character Outline

1.6 Segmentation

1.6.1 Introduction

The objective of task 6 is to segment the image and separate and label the different characters. Since the connected areas were already marked in task 5, the only step remaining is to assign colors to each labeled area to clearly distinguish the different characters.

1.6.2 Methodology

I. Converting color labels to RGB

Converts characters in an image (which have been segmented and labeled with different integer values) into a color image for more visual display and differentiation of individual characters. We used the “label2rgb” function and it can map integer labels to the RGB color space, making each label have a unique color. The RGB image is shown as below.

Figure 1.27 Labeled Characters

II. Segmentation into individual characters

Since the subsequent character classification requires individual characters, we use the “boundingbox” feature of the “regionprops” function to obtain the boundary position of each character.

Boundingbox is the smallest box that contains this character. Where the first two parameters are the coordinates of the upper left corner of the box and the last two parameters are the length and width of the box. Based on these data, we use the "imcrop" function to split and excise the characters to obtain a single character.

At the same time, we noticed that the characters in the original dataset were in white as the background color, so we used the imcomplementh function to invert the color and finally got the data that could be used, and some of the characters are shown below.

Figure 1.28 Character 0

Figure 1.29 Character H

Figure 1.30 Character A

Figure 1.31 Character 8

Chapter 2: Classification

2.1 Overview

In this section, we implement two classifiers, using CNN and SVM methods, respectively, to recognize characters. We will first present the implementation process for both methods and compare their efficiency and accuracy to analyze the differences between them. We also test the sensitivity of both models to an enhanced dataset based on rotation, translation, and scale transformations.

After comparing the results of the two models, we discuss the difficulties we encountered during the implementation process and our solutions. Through this analysis, we aim to gain a deeper understanding of the strengths and weaknesses of each method and to provide insights into how to optimize the performance of character recognition systems.

2.2 CNN Method

2.2.1 Introduction

In this task, we need to construct a CNN network and improve its recognition accuracy, then use it to recognize characters in the second line - HD44780A00 in Figure 1.1.

2.2.2 Methodology

I. Importing datasets

First, we imported the dataset and randomly divided it into a training set and a test set at a ratio of 3:1. Then, we set the image size to [128, 128, 1] and read the images.

II. Define the layer structure of the CNN

To accomplish task 6, we first define the layer structure, which includes the input layer, multiple 2D convolutional layers, batch normalization layer, ReLU layer, pooling layer, dropout layer, fully connected layer, softmax layer, and classification layer. The dropout layer is used to prevent overfitting.

For improved performance, we made some adjustments to the convolution kernel size, increasing it from 3 to 5, and increased the initial number of convolution kernels to 16. This helps to capture more features and patterns, which can enhance the

representation of the model. Following these adjustments, the accuracy of the validation set improved from 90.48% to 91.38%.

III. Define the training options

Adam (Adaptive Moment Estimation) and SGDM (Stochastic Gradient Descent with Momentum) are both optimization algorithms that are used to update parameters during the training of a neural network. We choose to use “adam” as the optimization algorithm and set the initial learn rate to be 0.001. Smaller learning rates lead to more accurate convergence but require longer training time. 0.001 is an appropriate learning rate due to our high accuracy requirements. Then set the maximum number of training rounds (epochs) to 20, because increasing the number of training rounds helps to improve the accuracy of the model.

After that we set the size of each mini-batch to 64. This is the number of samples used in each iteration. A smaller mini-batch size may help to improve the accuracy of the model. In addition, we change the ExecutionEnvironment to gpu. Using GPU usually speeds up the training process but has no direct impact on the accuracy.

IV. Classification prediction for the test set

First, we define and train the network using the training dataset, specifying the layers and options. The network's predicted results are stored in the variable YPred, while the true results are stored in YTest. Next, we calculate the accuracy of the predictions, generate the confusion matrix, and visualize it. The table and figures below demonstrate the training process and the confusion matrix used to assess the accuracy of the validation set.

Table 1 CNN training results

Round	Iteration	Time passed	Small batch accuracy	Validation accuracy	Small batch losses	Validation loss	Basic Learning Rate
1	1	1	7.81%	37.19%	4.0744	10.1321	0.0010
5	100	7	96.88%	99.09%	0.0867	0.0401	0.0010
10	200	13	100.00%	100.00%	0.0077	0.0064	0.0010
15	300	19	100.00%	100.00%	0.0034	0.0059	0.0001
20	400	25	100.00%	100.00%	0.0051	0.0047	0.0001

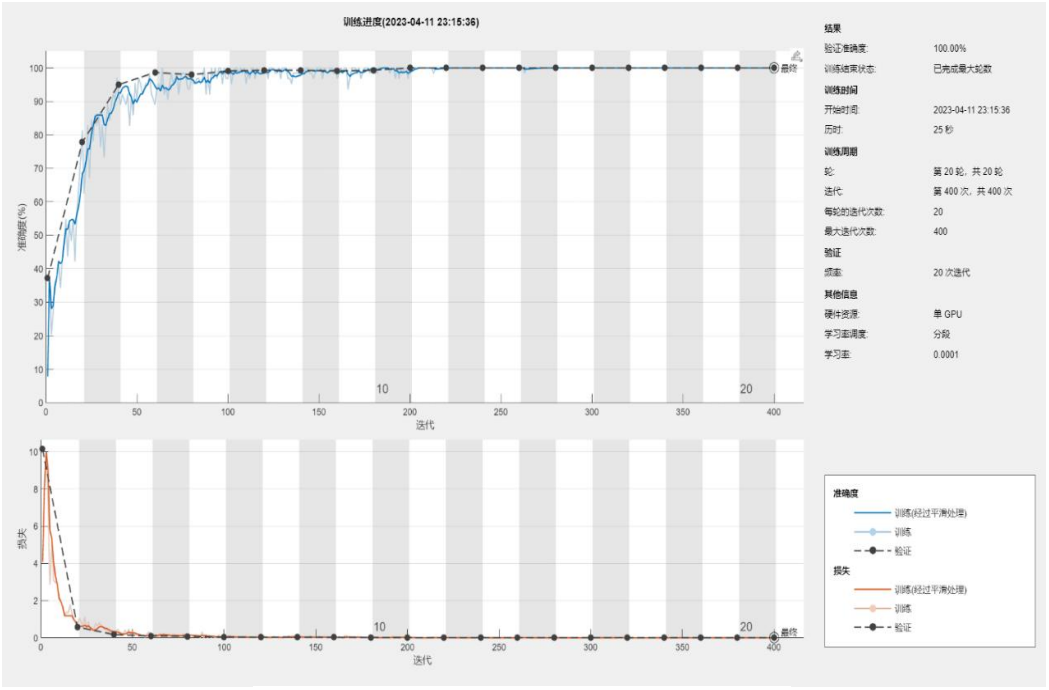

Figure 2.1 training process

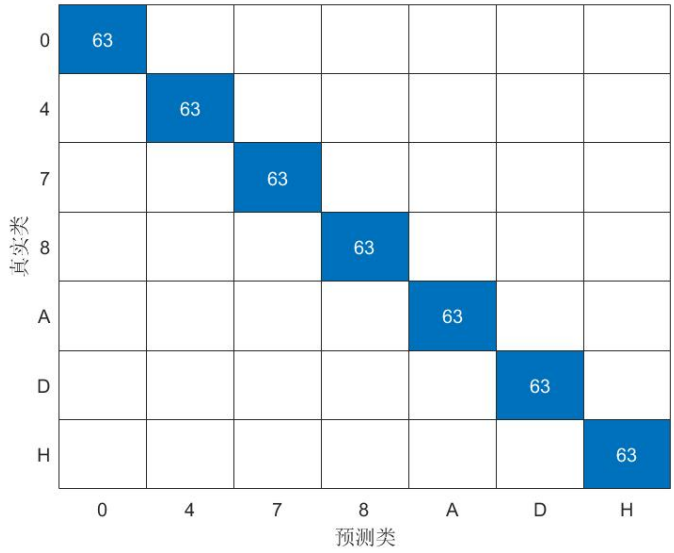

Figure 2.2 confusion matrix

From the figure we can see that both training accuracy and verification accuracy reach 100%.

V. Classification prediction for the characters required by the task

Afterwards, we import our own data and apply the pre-trained CNN model to recognize letters and numbers. However, the initial results of letter recognition using our raw data are not very accurate, with an accuracy rate of only around 70%. To address this issue, we propose a solution in the following section and compare the results before and after processing.

2.3 SVM Method

2.3.1 Introduction

In this task, we focus on creating a classification system that uses non-CNN methods to classify each character in an image. We used the SVM classification method, but with two different methods of training the SVM model and comparing the classification results with those of the CNN.

2.3.2 Methodology

I. Training the model after HOG extraction of features

Before we can train and test our character recognition models, we need to perform some pre-processing operations on the dataset. The steps we followed are:

a) Set the path to the given dataset and used the MATLAB function imageDatastore to read it into our program.

b) Divided the image dataset into a training set and a test set in a 3:1 ratio. This allows us to train our model on a subset of the data and test its accuracy on a separate set.

c) Scaling of images to a specified size for subsequent processing.

To get an idea of the distribution of labels in our dataset, we used the countEachLabel function. This function counts the number of occurrences of each label and returns the result. The output of this function is shown below:

Label	Count
0	191
4	191
7	191
8	191
A	191
D	191
H	191

Figure 2.3 Label and Count

Scaling of images to a specified size for subsequent processing. For each image in the training set, features are extracted: first an empty matrix featuresTrain is created to store the HOG features of the training set images, then each image is scaled and features are extracted using the extractHOGFeatures function.

Next, we use the fitcecoc function to train the extracted features and image labels to obtain the training model classifer:

```
classifer = fitcecoc(featuresTrain, trainLabels);
```

The fitcecoc function is a MATLAB function for multi-category classification that uses the “Error-Correcting Output Codes” method to transform a multi-category classification problem into a number of binary classification sub-problems, each corresponding to a coding method. The fitcecoc function can use different classifiers for the binary classification subproblems, such as support vector machines (SVM) and linear classifiers. It automatically selects the appropriate classifier, trains a multi-category classification model, and returns the model object mod.

```
accuracy =

    0.9932
```

Figure 2.4 Predict accuracy

The trained model performed relatively well on the test set, while the predicted results for the sample are as follows:

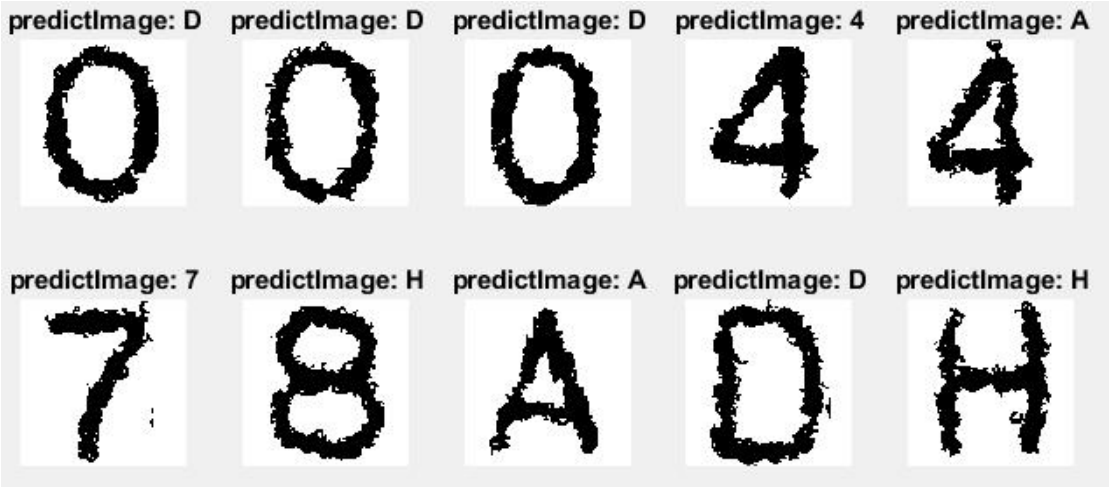

Figure 2.5 Predict results

This method processed to poor recognition results for '0' and 'D', '4' and 'A', and '8' and 'H'. However, it is not possible to manually adjust the training parameters of the

SVM using the fitcecoc function, so we have also used the import of the libsvm library to implement adjustable parameters for model training.

II. Libsvm-based training methods (Adjustable parameters)

Before proceeding with this step, you first need to download the libsvm library and then import it into MATLAB.

The dataset import step is different from the previous one in that the labels of the training and test sets need to be converted into double format, as the svmtrain and svmpredict functions in libsvm only accept data in double format.

For the image dataset, which also needs to be converted to double format for processing, we list the pixel values of all the pixel points of each image and then use the reshape function to list all the pixel values as a row, representing the dataset of an image that is available for svm calculation.

```
Accuracy = 97.0522% (428/441) (classification)
Time: 18.47 s.
```

Figure 2.6 Predict accuracy and Train time

We then validate the image given by the task, for which we need to process the labels and images in the same way as for the previous image dataset. Before outputting the results, the labels need to be converted to their original format to facilitate subsequent comparisons.

The parameters we use when using svmtrain are:

```
options='-s 0 -t 0 -q';
```

Where:
-s : Selects the training algorithm, where 0 denotes C-SVC, 1 denotes v-SVC, 2 denotes a class of SVM, 3 denotes e -SVR, 4 denotes v-SVR and defaults to 0. C-SVC is chosen here, which can classify different samples by finding an optimal hyperplane, while controlling the complexity and generalization ability of the decision boundary through the regularization parameter C.

-t : Selects the SVM kernel function type, where 0 means linear kernel function, 1 means polynomial kernel function, 2 means radial basis kernel function, 3 means sigmoid kernel function, 4 means custom kernel function, and the default is 0. The linear kernel function is chosen here because it is computationally fast for the training set used and is suitable for training on large scale data sets. And the image features of numbers and letters are relatively simple, without too many complex features, and the linear classifier can already do the task well. In some cases, the non-linear kernel

function may over-fit the training set, resulting in degraded performance on the test set. The linear kernel function, on the other hand, is more robust and less prone to overfitting.

-q : Whether to turn on quiet mode, where 1 means no information and warnings are output during training and 0 means output. The default is 0.

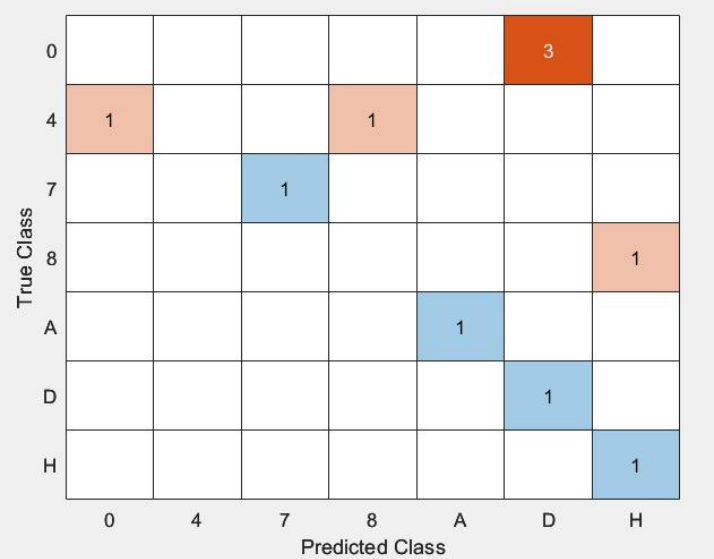

Figure 2.7 Results before image processing

As you can see the recognition results are not very satisfactory. So we use the method of processing the images for both two methods, adding white boxes around each image, which can improve the contrast of the image, making the image features more obvious, so that we can better extract the image features and improve the recognition rate.

After image processing, the recognition results were obtained as follows:

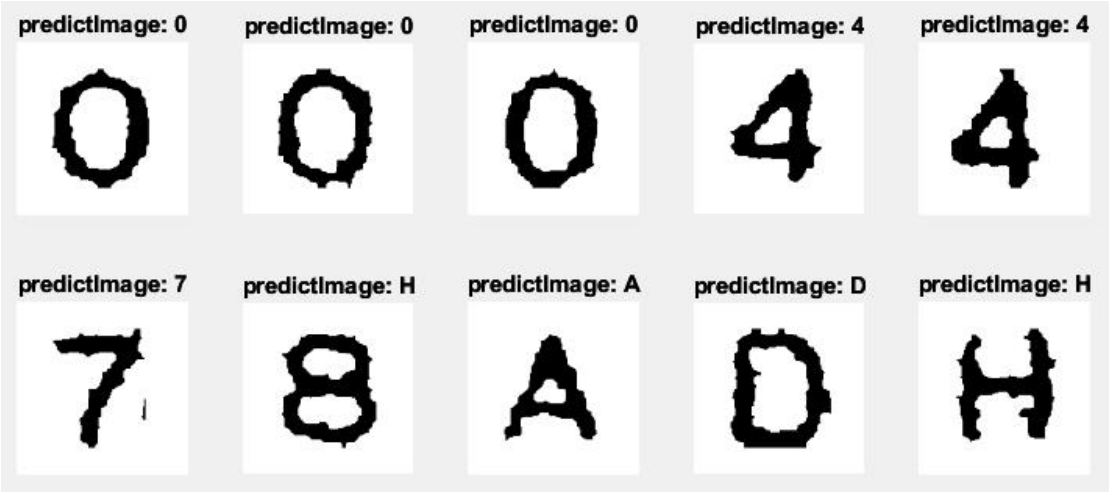

Figure 2.8 Results after image processing（HOG extraction）

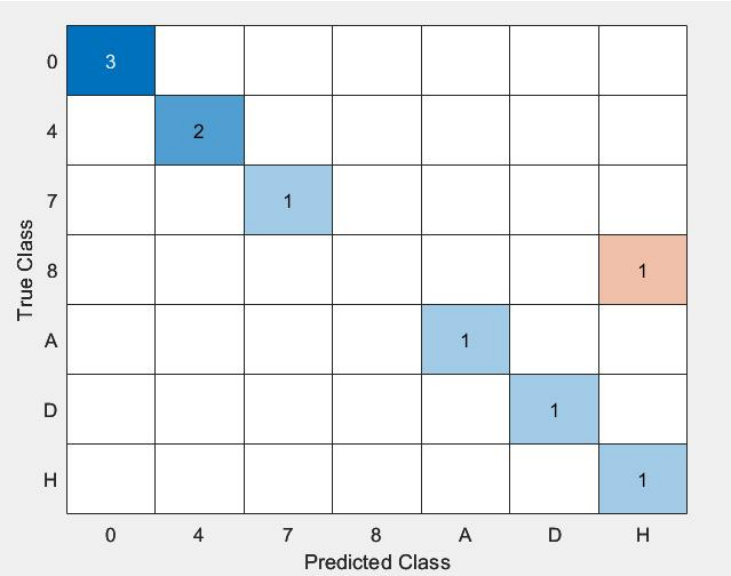

Figure 2.9 Results after image processing（libsvm）

For both methods, the results are much better, but there is still one case where the '8' is incorrectly identified as an 'H', probably because they are very similar in shape, being made up of two vertical lines and a horizontal line. Also in the given training set, there were many images labelled 'H' that were also judged by the naked eye to be the number '8', as shown in the following figure:

Figure 2.10 Confusing images

2.4 Problems and Solutions

2.4.1 Pre-process of the images

To enhance the features of the image and improve character recognition accuracy, we first apply various operations such as erosion and dilation to remove noise, fill in voids, connect broken parts, and smooth borders. This preprocessing step significantly improves the quality of the images, as shown in the comparison below, which illustrates the characters before and after the operations.

Figure 2.11

Figure 2.12

Figure 2.13

Figure 2.14

Figure 2.15

Figure 2.16

Figure 2.17

Figure 2.18

For a more effective morphological treatment, we set both the erosion radius and the expansion radius to 5. As shown in the figure above, the characters after processing appear fuller and more suitable for feature extraction and recognition. To illustrate the impact of this preprocessing step on character recognition accuracy, we compare the results before and after processing in the figure below.

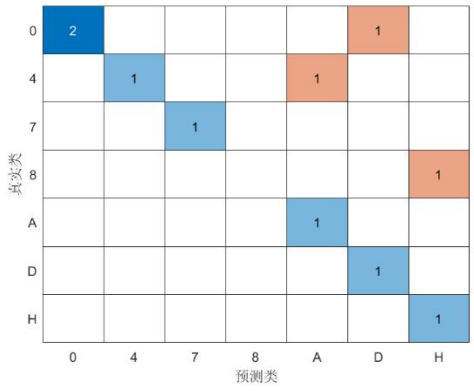

Figure 2.19 Before processing

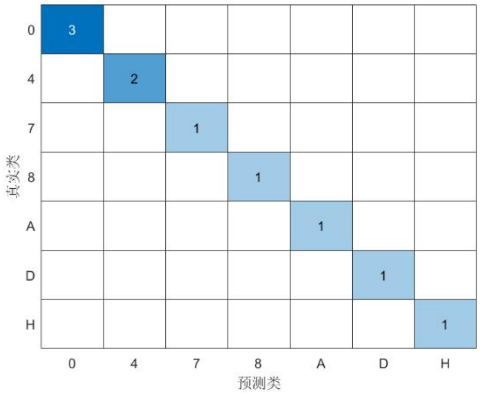

Figure 2.20 After processing

2.4.2 Ignoring the need to randomly assign the dataset

Upon importing and dividing the dataset into two parts, we noticed that the validation set's recognition success rate was only about 92%. Further analysis revealed that the dataset had not been randomly assigned, resulting in biased data sets.

To address this issue, we modified the assignment process by using the 'randomize' parameter to ensure that the images were randomly assigned to the training and validation sets. After this improvement, we achieved a recognition accuracy rate of 100%.

2.5 Results and Comparisons

2.5.1 Performance in training set

The performance comparison of the three models on the training set is shown in the following table:

Table 2 Comparison of 3 models on the training set

Model	Accuracy	Training Time
CNN	100%	26.21 s
HOG-based SVM	97.05%	7.00 s
Libsvm-based SVM	97.05%	13.63 s

```
Training Time: 26.21 s.

accuracy =

     1
```

Figure 2.21 CNN performance

```
Training Time: 7.00 s.

accuracy =

    0.9705
```

Figure 2.22 HOG-based SVM performance

```
Accuracy = 97.0522% (428/441) (classification)
Training Time: 13.63 s.

accuracy =

   97.0522
```

Figure 2.23 Libsvm-based SVM performance

From the table 2, we can see that the CNN model achieved the highest accuracy on the test set, followed by the HOG-based SVM model and the Libsvm-based SVM model. These two SVM method has the same accuracy of 97.05%. However, higher accuracy also means longer processing time, and the training time for the CNN method is also the longest among the three methods, with the shortest training time for the HOG-based SVM.

2.5.2 Performance in test image

The performance of the three methods to detect the letters in Figure 1.1 is shown below：

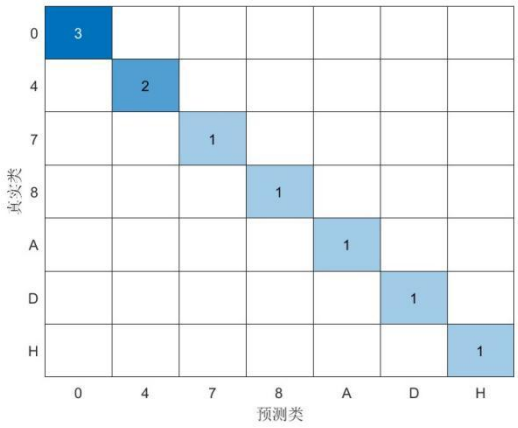

Figure 2.24 CNN performance in test image

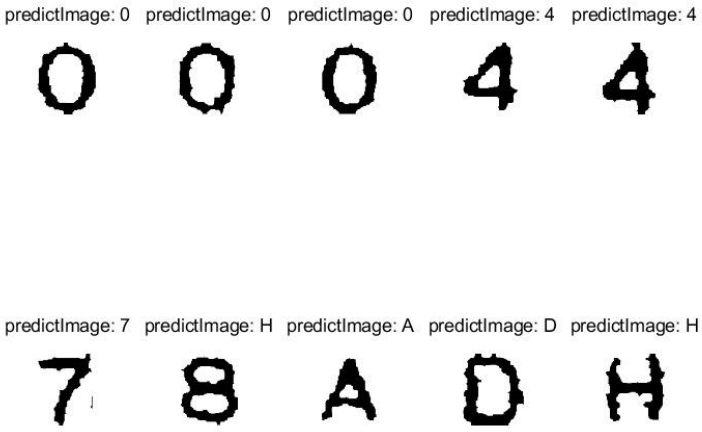

Figure 2.25 HOG-based SVM performance in test image

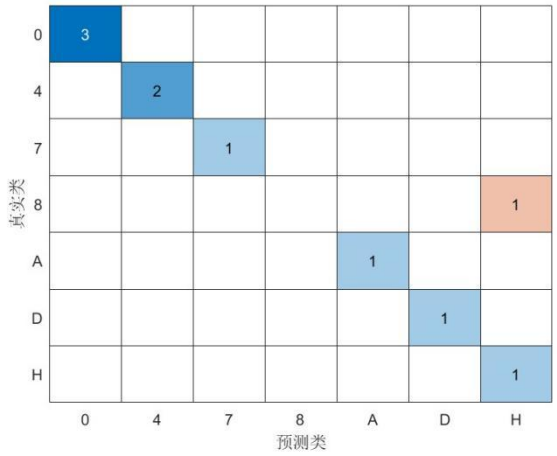

Figure 2.26 Libsvm-based SVM performance in test image

We can observe that the CNN method was able to perfectly identify the characters in Figure 1.1, while both SVM methods incorrectly identified the number 8 as the letter H. And this case corresponds to the comparison result in 2.5.1.

2.5.3 Analysis

Based on the results of the SVM method for identifying false positives between the characters 8 and H, we conducted a thorough examination of the dataset and discovered that certain images labeled as H exhibit remarkably similar features to those of the number 8, and Figure 2.27 gives some examples.

Figure 2.27 some confusing examples in H dataset

As depicted in Figure 2.27, it is noticeable that the letter H appears to be connected at both the top and bottom, thereby bearing a resemblance to the numerical character 8. It is possible that the SVM method struggles to differentiate between the number 8 and the letter H based on other features, which may have contributed to the observed results.

Hence, it can be logically deduced that the CNN model attained the highest accuracy owing to its deeper architecture, which enables it to more effectively extract features from the input images. Although the SVM model also demonstrated commendable performance, its capabilities are limited by its feature extraction method, which may not capture all of the crucial features in the input images.

In addition, we also tested the sensitivity of the three models to the enhanced dataset based on rotation, translation and scale transformation. The results showed that the CNN model was more robust to these transformations than the HOG-based SVM and Libsvm-based SVM models, indicating that the CNN model has better generalization ability.

Chapter 3: Conclusion

3.1 Pre-processing part

In task 2 and 3, we introduced several common filters, such as mean filter, median filter, and Laplacian filter. And we implemented the corresponding algorithms in Matlab by ourselves according to the principles. Through debugging as well as comparing the results, we have come to the following conclusions:

a) Images are processed in the form of matrix.
b) The larger the mask size of the smoothing filter, the more severe the blurring of the image.
c) The sharpening filter highlights the contours in the image, but at the same time introduces low frequency noise.
d) Combining multiple complementary pretreatment methods often yields better results.
e) When processing RGB images in Matlab, the three channels are usually processed separately.
f) Some filters need to use the matrix of unit type for processing, such as the mean filter, while some filters have the output type double, we should pay attention to the data type conversion in practice.

In task 4, we have adopted an adaptive thresholding method based on image averaging, with the advantages of:

a) Less susceptible to noise interference: the method is better at suppressing noise as it is based on local mean for segmentation.
b) Better segmentation results: to a certain extent, the detailed information of the image can be preserved, thus obtaining better segmentation results.

In task 5, we used the connected region labeling function and convolution function written by ourselves, and removed the noise in the image, and finally used the Sobel operator for edge detection, and successfully came up with the final image.

In task 6, since the labeling of connected regions has been performed previously, only the labeled L-matrices need to be used directly in task 6. The point to note is that the labels of the connected region are discontinuous because of the noise removal after the first labeling, so a second labeling is needed to make the labels of the connected region start from 1 and increasing numbers continuously.

3.2 Classification part

In this section, we employed both CNN and non-CNN methods to independently recognize letters.

I. CNN

For CNN method, we employed a Convolutional Neural Network (CNN) as the central model and refined its structure and training parameters through experiments. The implementation involved splitting the image dataset into training and testing sets and creating a multi-layered CNN. We applied techniques such as the Adam optimizer and a piecewise learning rate adjustment strategy during training.

By making predictions on the test set, we assessed the network's accuracy and verified its performance on task images. The experimental results demonstrated that the constructed CNN model performed admirably on both the test set and task images.

Future research can aim to further optimize the network structure and training parameters to enhance recognition accuracy and investigate the model's applicability to a broader range of image classification tasks.

II. SVM

For non-CNN method, We employed the Support Vector Machine (SVM) method as the core model and divided the images into training and testing sets to evaluate the accuracy of the trained model for image recognition. Two approaches were utilized: the first involved extracting HOG features from the image and inputting the feature data into the training function for training, while the second approach directly converted the image pixels into a data set for SVM classification and recognition.

The advantage of SVM is that it can handle high-dimensional space and non-linear classification problems. It can transform non-linear problems into linear ones through kernel functions and identify the optimal hyperplane for classification. SVM is robust and tolerant to noise and outliers, and the model is sparse in that it only employs the support vector in the model, which has no effect on the model for data points that are not support vectors.

The parameters utilized in our ‘svmtrain’ function are the default parameters. If more types of images need to be recognized in the future, the parameters should be changed and optimized accordingly.

III. Comparison

After comparing the experimental results, we found that the CNN model is more accurate than SVM. However, the CNN model requires more computation time due to its large number of parameters to learn and the need for multiple iterations over the entire dataset during training. This can take a significant amount of time and computing power.

On the other hand, traditional machine learning methods such as SVM have fewer parameters to learn and rely on handcrafted feature extraction techniques, such as HOG. However, these techniques may not capture all the relevant information from the input image.

Overall, CNN often outperforms SVM in image classification tasks, particularly when the dataset is large and complex. Nevertheless, choosing the appropriate method should be based on a trade-off between accuracy and processing time, depending on the specific requirements and constraints of the application.