CA Project Report – Group 8

WANG CHEN A0268426W E1101725@u.nus.edu

LIU YUCHI A0268564N E1101863@u.nus.edu

XU GUANGDA A0268548L E1101847@u.nus.edu

1 Introduction

This research project is primarily focused on the field of image processing and character recognition, which has garnered significant interest in the computer science and engineering communities. The primary objective of this study is to develop an efficient and accurate character recognition system through the use of various image processing techniques and classification methods.

The first segment of the project, consisting of tasks 1 to 6, involves processing an image to extract characters and segmenting them into individual components. In the second part of the project, the task 7 requires using a provided dataset to train and validate two classification systems for the characters presented in the second line of Image 1. In sub-task 1 of task 7, a Convolutional Neural Network (CNN) is developed for classification, while a non-CNN based method is required for sub-task 2. In sub-task 3, the effectiveness and efficiency of the two approaches are compared.

Finally, in task 8, the sensitivity of the two approaches to various changes are tested by exploring a range of pre-processing methods and tuning hyperparameters.

2 Description of Algorithms

2.1 Part 1

2.1.1 Task 1

The initial step in the code involves reading an image from a graphics file named "charact2.bmp" through the implementation of the "imread" function. Subsequently, the "imshow" function is employed to display the original image.

In order to enhance the legibility and comprehensibility of the output, descriptive titles have been incorporated for each step via the utilization of "title" and "sgtitle" functions, which will not be reiterated in the succeeding sections.

2.1.2 Task 2

In the field of image processing, a mask, also known as a kernel or filter, is a small matrix that is utilized to modify or filter an image. Each pixel of the image is subjected to the mask, resulting in a new pixel value that represents the altered or filtered image[1].

In task 2, we experimented with two common types of masks, namely averaging masks and rotating masks. Averaging masks are square matrices of uniform values that are summed and then divided by the total number of elements in the mask. On the other hand, rotating masks usually contain a combination of positive and negative values that are rotated around the center of the mask.

To implement a series of averaging masks and rotating masks, we defined a function "ave_rot_mask" at the end of the code. The function takes four inputs: the original RGB image, the size of the mask, the type of mask (either 'a' for averaging masks or 'd' for rotating masks), and the color space (i.e. ‘rgb’ or ‘gray’). It outputs the filtered image, which is in the same color space as the input image.

The flowchart of the "ave_rot_mask" function is depicted in Figure 2.1.

For RGB images, the function applies the generated mask to each color channel of the input image using the ‘conv2’ function, while for grayscale images, the mask is directly applied. After that, the output image is then

converted by ‘mat2gray’ and then to 8-bit unsigned integer format, which has a range of 0 to 255, using ‘im2uint8’ function. The resulting filtered image is then normalized to be correctly displayed through ‘rgb_to_bin’, which converts the RGB input image to grayscale using the ‘rgb2gray’, and then binarizes the grayscale image using a threshold value as an input parameter, resulting in a binary output image.

In the realm of image processing, the selection of an appropriate filter can have a profound impact on the quality and accuracy of the output image. In our project, we expect that the application of an averaging mask will lead to a visually smoother output image by averaging the pixel values within the specified area, thereby reducing the noise and achieving uniformity. On the other hand, however, the use of rotating mask will help to preserve the edge information and produce a sharper appearance in the output image.

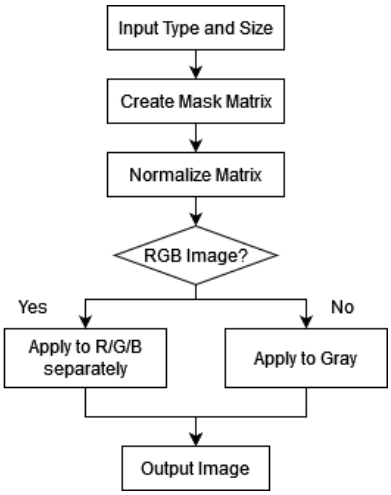


Figure 2.1 ‘ave_rot_mask’ function

2.1.3 Task 3

To accomplish this task, we utilized the "size" function to obtain the height, width, and number of channels of the original image. From there, we created a sub-image that includes only the middle line of the image. Specifically, the rows starting from the middle of the image is selected, which includes the middle line, and proceeded to the end of the image (i.e., HD44780A00).

It should be noted that this method of sub-image creation is dependent on the orientation of the original image. In this case, the middle line is oriented horizontally, and thus the rows containing the middle line could be easily extracted. If the middle line were oriented vertically, a different approach to sub-image creation would be required.

2.1.4 Task 4

In this part of the program, the 'rgb_to_bin' function is utilized to convert both the original image and the sub-image into binary images. This conversion is accomplished by means of thresholding all the pixel values of the image based on a specific value. Ultimately, the binary image produced only contains black and white pixels, and that feature is particularly advantageous for subsequent segmentation and labeling processes [2].

The flowchart of ‘rgb_to_bin’ function is shown in Figure 2.2.

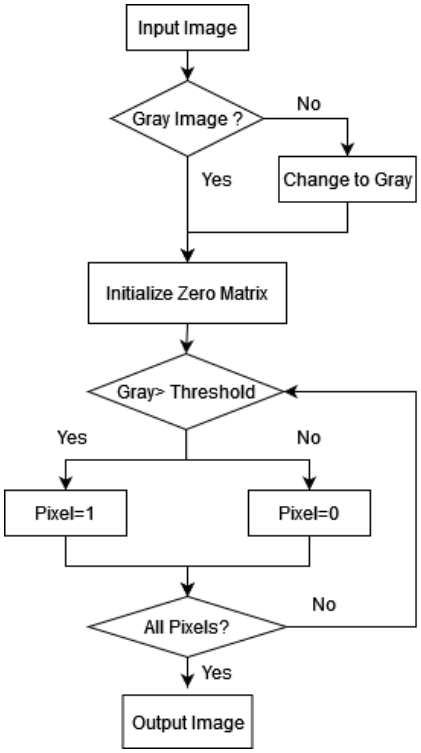


Figure 2.2 ‘rgb_to_bin’ function

It should be noted that an alternative

approach to this process was initially considered, namely using the 'graythresh' function to calculate the global image threshold and the 'imbinarize' function to binarize the image. Despite this, our experimental results revealed that the global image threshold obtained using the 'graythresh' function was not optimal for our purposes. Therefore, we implemented a manual fine-tuning process to obtain a threshold value that would yield more desirable results.

2.1.5 Task 5

In this task, we employ the ‘bwperim’ function to identify the outlines of the binary images created in the previous task 4. This function generates another binary image, which solely contains the perimeter pixels of objects within the input image. That is, the perimeter pixels are marked with the value of 1 and then all other pixels are set to the value of 0 [3].

In fact, when it comes to determining outlines of characters, there are two common used functions – ‘bwperim’ and ‘edge’.

The ‘bwperim’, which follows the outlines by scanning the image in a clockwise direction from the top-left corner, is better at finding the perimeter of foreground objects. Its limitation is that it not only extracts the outer contour, but also extracts the inner edges enclosed by the holes in the graphics area, which is sometimes unwelcomed. However, in our project, this will not cause any negative impact.

The 'edge' function, on the other hand, involves a variety of algorithms. For example, the Sobel operator convolves the input image with two 3x3 kernels to determine the horizontal and vertical gradients. The magnitude of the gradient is then used to detect edges [4].

Based on the characteristics of these two algorithms, we chose the former for our purpose.

2.1.6 Task 6

This task involves performing image processing operations aimed at extracting and labeling connected components from previously created binary images. To achieve this, various image processing techniques are applied to the original image to enhance contrast and reduce noise[5].

This part of the program begins by taking the binary image as input and initializing an all-zero matrix labeled_image of the same size as the binary_image to store the labeled image. Furthermore, the labeled_num variable is set to 0, indicating the number of labels.

The program then iterates over each pixel in the binary image. If the pixel is a foreground pixel (i.e., a pixel with a value of 1) but has not been labeled, the program follows the steps below.

The labeled_num variable is incremented to create a new label for the current connected region. Subsequently, a queue is initialized with the current pixel, and the pixel is added to the queue.

A breadth-first search is initiated, and for each pixel in the queue, the program carries out the following steps. The pixel at the top of the queue (i.e., the first pixel added) is ejected and labeled with the current label number in the labeled_image matrix. Then, the 8 surrounding pixels of that pixel are checked. If a pixel is a foreground one but has not been labeled, it is added to the queue and labeled with the current label number.

The search continues until the queue is empty, indicating that all pixels in the current connected region have been labeled. The number of labels num_regions is then counted, representing the total number of connected regions in the image.

Finally, the labeled_image matrix is returned as output. This output provides a visual representation of the connected regions in the binary image and facilitates further

analysis of the image. Overall, this process involves utilizing various image processing techniques to accurately identify and label connected regions in binary images.

From tasks 1 to 6, the 'subplot' function is used in displaying, which greatly improves the readability.

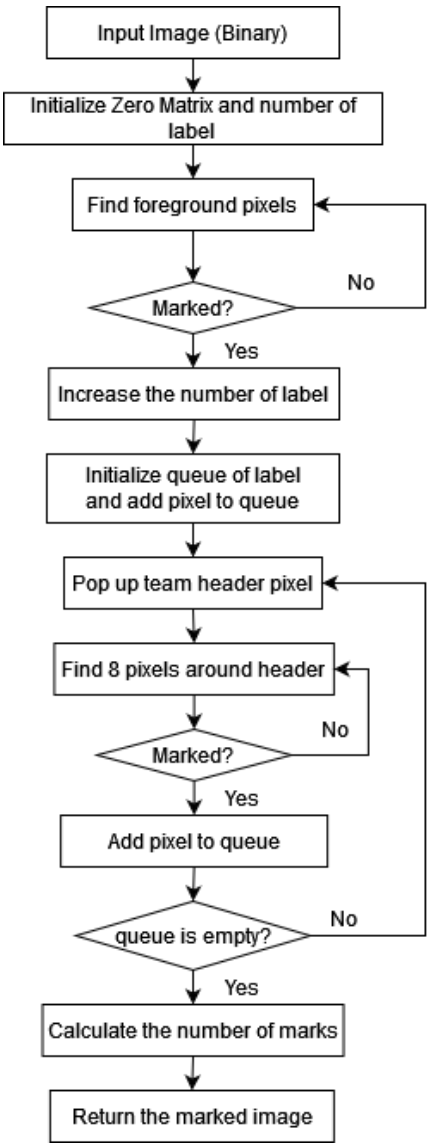


Figure 2.3 'segment_bwlabel' function

2.1.7 GUI Description and Screen Captures of Each Stage

Graphical User Interface (GUI) is a type of user interface that allows users to interact with a software program or application using graphical elements such as buttons, menus, and icons[6]. In this app, the GUI is designed to provide users with a user-friendly interface for image processing tasks.

To use this app, the user needs to run the "app2_exported.m" file which will launch the GUI. Upon launching, the user will see a screen same as Figure 2.4.

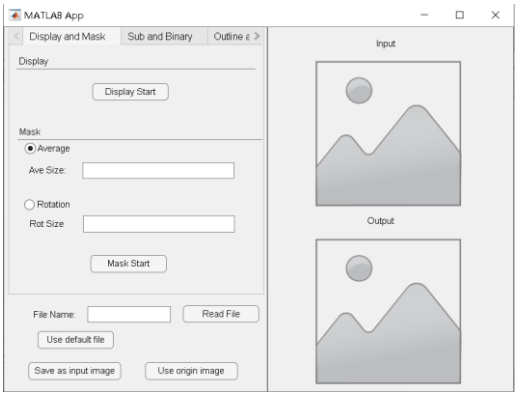


Figure 2.4 App Screen

Task 1 involves displaying the original image. To do this, the user can either use the default image provided by the app by clicking on the "Use default file" button or enter their own image name in the "File Name" textbox. After selecting an image, the user needs to click on the "Read File" button to read the image data into the MATLAB workspace. The input and output image frames will both display the original image, as shown in Figure 2.5.

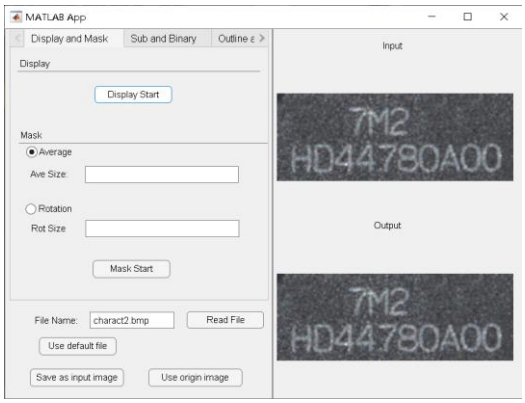


Figure 2.5 Original Image

Task 2 involves applying a mask to the image. The user can choose to apply an averaging or rotating mask. To apply the rotating mask, the user needs to enter the mask size into the textbox and then click on the "Mask Start" button. The output image with the

applied mask is displayed, as shown in Figure 2.6. The user can save the output image as input by using the “Save as input image” button.

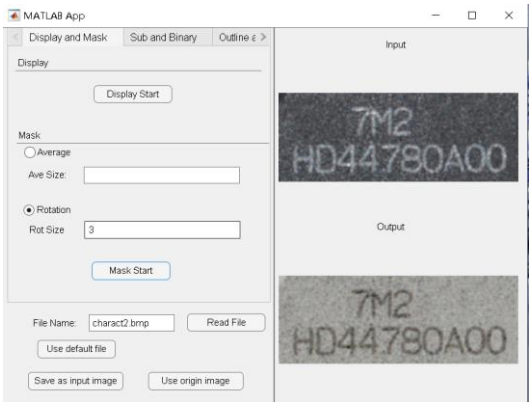


Figure 2.6 Mask

Task 3 involves creating a sub-image from the original image or masked image. The user can select either the upper part or the bottom part of the image and use the slider to control the percentage of the sub-image. For example, if the user selects the “Bottom_Half” option and drags the slider to 0.5, a sub-image will be created as shown in Figure 2.7. The user can save the sub-image as input.

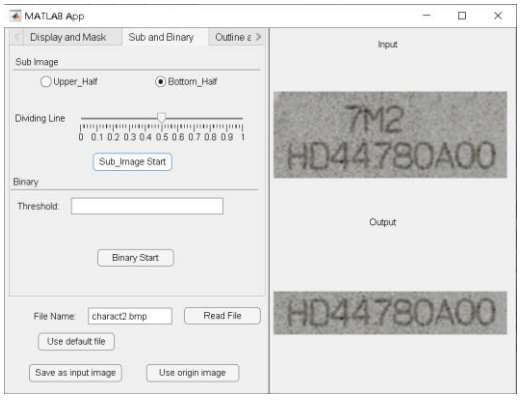


Figure 2.7 Sub-Image

Task 4 involves creating a binary image from sub-image. The user needs to enter the desired threshold value in the “Threshold” textbox (between 0 and 255) and click on the “Binary Start” button. The sub-image will be converted to a gray image and then a binary will be created and displayed as shown in Figure 2.8. The user can save the binary image as input. (It will be used in the following tasks.)

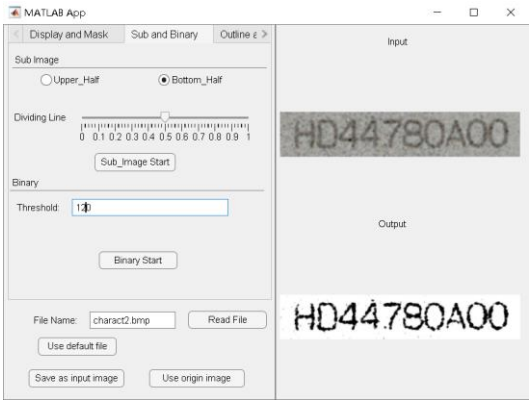


Figure 2.8 Binary Image

Task 5 is aiming to create an outline of the binary image. The user need to select the number of neighbors (usually 8 neighbors) and click on the “Outline Start” button. The outline of the binary image will be displayed as shown in Figure 2.9.

However, the user cannot save the outline of binary image as input, because what the segmentation of the image in task 6 requires is a binary image.

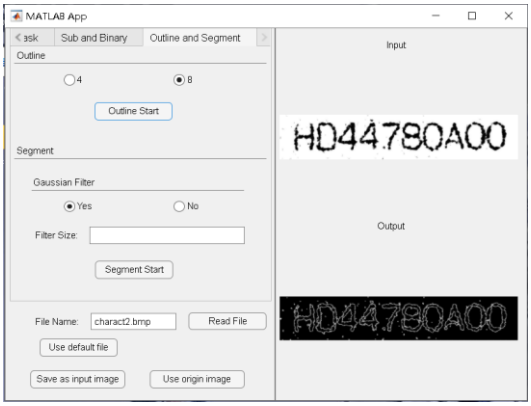


Figure 2.9 Outline

Task 6 involves segmenting the binary image. The user needs to enter a binary image and choose whether to add a Gaussian filter before segmenting. It is generally better to add a filter before segmenting. In Figure 2.10, if the user choose to add a Gaussian filter of size 3 before segmenting, the segmented result is displayed as the output image.

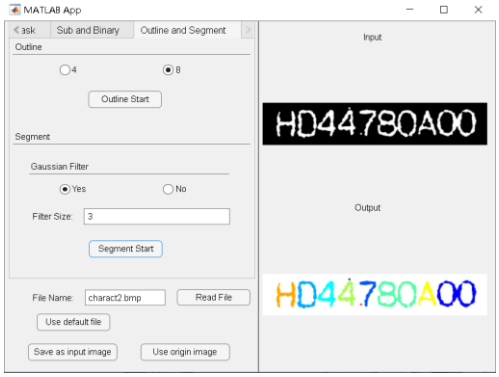


Figure 2.10 Segmentation

In conclusion, the GUI provided by this app allows users to perform various image processing tasks easily and efficiently. The user-friendly interface makes it easy for even non-experts to manipulate images and carry out various image processing operations.

2.2 Part 2

2.2.1 Task 7

2.2.1.1 Create a CNN to classify characters

In this section, we need to create a CNN neural network model using the provided p_dataset_ 26 training data trains the model, then saves the trained model. Then loads the trained CNN model through a test program, and classifies and recognizes the alphanumeric images segmented from previous tasks.

2.2.1.1.1 Create CNN model and train

In this step, we will classify the data (part for training and part for testing) and perform a small amount of preprocessing, then create a CNN model, input data, and train this classification model. Finally, we will test the data, draw a table to verify the accuracy of the classification, and explain the classification effect of the model.

1.Loading the dataset

The provided code starts by loading the dataset using the imageDatastore function, which creates an image datastore from the 'p_dataset_26' directory. This function also includes the option to read images from subfolders and use the folder names as labels.

2.Splitting the dataset

The dataset is then split into a 75% training set and a 25% validation set using the splitEachLabel function.

3.Constructing the LeNET CNN architecture

The LeNET architecture is then defined, which includes an input layer, three convolutional layers, two max-pooling layers, two fully connected layers, a softmax layer, and a classification output layer.

4.Analyzing and visualizing the network

The constructed network is then analyzed and visualized using the layerGraph and analyzeNetwork functions.

Figure 2.11 Analysis CNN Network

5.Resizing the training and validation sets

The training and validation sets are resized to match the input layer size of the LeNET architecture using the augmentedImageDatastore function.

6.Configuring the training options

The training options are configured with the trainingOptions function. The configuration includes using Stochastic Gradient Descent with Momentum (SGDM) as the optimization algorithm, an initial learning rate of 0.001, a maximum of 20 epochs, shuffling the data every epoch, and setting the validation data and frequency.

7.Training the CNN

The CNN is then trained using the trainNetwork function with the resized training set, the LeNET architecture, and the specified training options.

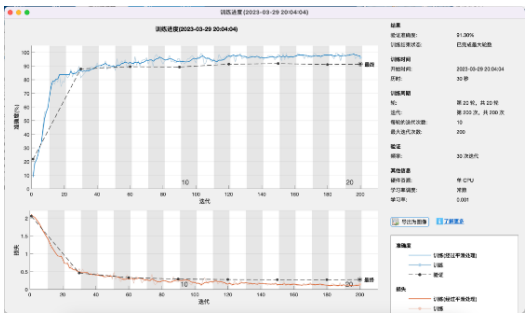


Figure 2.12 CNN Training

8.Saving the trained model

The trained model is saved as 'trainedCNNNet.mat' for future use.

9.Evaluating the model

The trained model is evaluated by predicting the class labels of the resized validation set using the classify function. The accuracy of the predictions is calculated as the percentage of correct predictions.

10.Displaying the confusion matrix

Finally, a confusion matrix is plotted to visualize the performance of the model on the validation set using the confusionchart function.

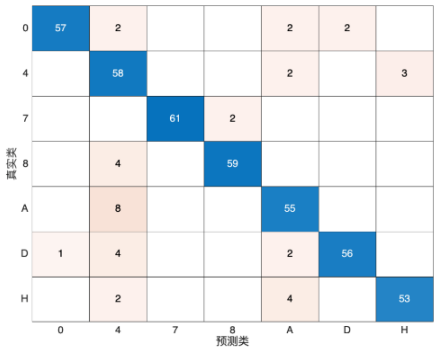


Figure 2.13 Confusion Matrix

2.2.1.1.2 Load the model and classify the results of previous segmentation

In this section, we wrote a simple code to preprocess and classify a single test image using the trained CNN model，the process at below:

1.Loading the test image

The code starts by loading a test image ('test_h.png') and displaying it using the imread and imshow functions.

2.Inverting the image

The image is then inverted (white-to-black and black-to-white) using the imcomplement function, and the result is displayed using imshow.

3.Converting the image to grayscale

The image is then converted from RGB to grayscale using the im2gray function.

4.Resizing the image

The grayscale image is resized to match the input dimensions of the trained CNN model (60x20) using the imresize function.

5.Loading the trained model

The trained CNN model ('trainedCNNNet.mat') is loaded into the workspace using the load function.

6.Classifying the test image

The loaded CNN model is used to classify the preprocessed test image using the classify function.

7.Displaying the classification result

Finally, the predicted class label is displayed using the disp function.

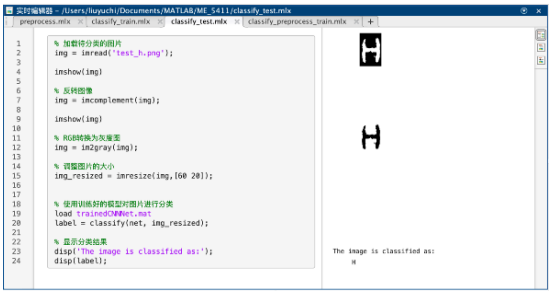


Figure 2.14 Predicted Class Label

2.2.1.2 Create a MLP to classify characters

In this section, we will no longer use the CNN model but instead use the MLP model to replace it. Similarly, we will create the structure of the MLP model and use the same methods as before to train and analyze the results of the new model. In order to simplify the report and the duplicate parts of the previous code, we will no longer display them here, but only display different parts, such as the visual analysis of the MLP model The training process and the results obtained from testing the test dataset

2.2.1.2.1 Create MLP model and train

Here, we will show three different parts from previous CNN training, such as model

structure, training process, and classification prediction results, to reduce redundancy

1.Analyzing and visualizing the network

The constructed network is then analyzed and visualized using the layerGraph and analyzeNetwork functions.

Figure 2.15 Analysis MLP Network

2.Training the MLP

The MLP is then trained using the trainNetwork function with the resized training set, the LeNET architecture, and the specified training options.

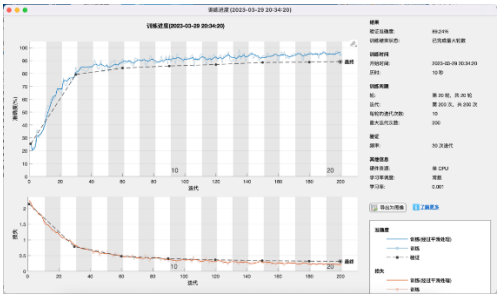


Figure 2.16 Training MLP

3.Displaying the confusion matrix

Finally, a confusion matrix is plotted to visualize the performance of the model on the validation set using the confusionchart function.

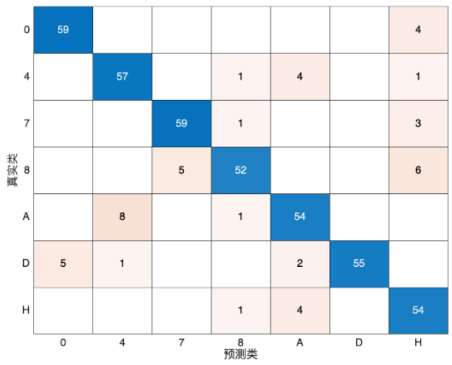


Figure 2.17 Confusion Matrix

2.2.1.2.2 Load the model and classify the results of previous segmentation

The basic code for this section is basically the same as the previous section, except that we have loaded the MLP model we just trained here instead of the CNN model.

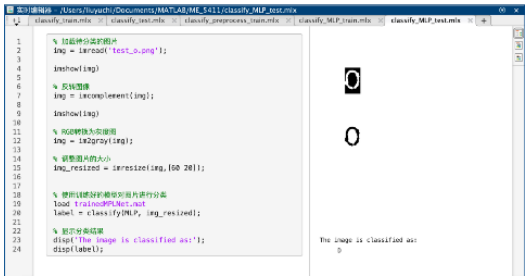


Figure 2.18 Predicted Class Label

2.2.2 Task 8

In this section, we will discuss the sensitivity of the character classification approaches (CNN and MLP) to data preprocessing techniques and hyperparameter tuning.

2.2.2.1 Influence of pre-processing of the data

Both CNN and MLP models can be sensitive to data preprocessing techniques, such as padding and resizing input images. For instance, resizing images can affect the models' performance by either enhancing or diminishing the relevant features within the images. Additionally, padding can help preserve information at the edges of images, which can be beneficial for the classification task.

In our experiments, we resized the input images to match the input dimensions of the models. It is important to choose an appropriate resizing strategy to preserve the aspect ratio of the images and retain as much relevant information as possible. Inappropriate resizing could lead to distorted images and decreased model performance.

In this section, I conducted various forms of preprocessing on the input data based on the CNN model used in Task 7, and conducted model training on these preprocessing methods.

Finally, I made a histogram of the accuracy of these training results to visually see the contrast effect, the process is at below(Some steps similar to the previous process will not be repeated):

1.Load the image dataset from the folder 'p_dataset_26', including subfolders, and obtain labels from the folder names.

2.Split the dataset into training (75%) and validation (25%) sets.

3.Define the LeNet convolutional neural network architecture.

4.Visualize the network architecture using layerGraph and analyzeNetwork functions.

5.Resize the images in the training and validation sets to match the input size of the LeNet network (60x20 pixels, grayscale).

6.Configure the training options, such as Stochastic Gradient Descent with Momentum (sgdm) optimization, initial learning rate, maximum number of epochs, data shuffling, validation data, validation frequency, verbosity, and progress plots.

7.Define various data augmentation scenarios: no augmentation, horizontal flip, horizontal flip and rotation, and all augmentations (including translation, shear, and scaling).

```
% Set a series of data enhancement strateg
ies
augmentationScenarios = {
    'NoAugmentation', ...
    'HorizontalFlip', ...
    'HorizontalFlipAndRotation', ...
    'AllAugmentations'
};
% Validation accuracy for storing each enh
ancement policy
validationAccuracy = zeros(1, numel(augmen
tationScenarios));
% Traverse different data enhancement stra
tegies
for i = 1:numel(augmentationScenarios)
    switch augmentationScenarios{i}
        case 'NoAugmentation'
            augmenter = imageDataAugmenter()
;
        case 'HorizontalFlip'
            augmenter = imageDataAugmenter('
RandXReflection', true);
        case 'HorizontalFlipAndRotation'
            augmenter = imageDataAugmenter(
...
                'RandXReflection', true, ...

                'RandRotation', [-10, 10]);
        case 'AllAugmentations'
            augmenter = imageDataAugmenter(
...
                'RandXReflection', true, ...

                'RandRotation', [-
10, 10], ...
                'RandXTranslation', [-
4, 4], ...
                'RandYTranslation', [-
4, 4], ...
                'RandXShear', [-2, 2], ...
                'RandYShear', [-2, 2], ...
                'RandScale', [0.9, 1.1]);
    end
```

8.Initialize an array to store the validation accuracies for each augmentation scenario.

9.Iterate through the augmentation scenarios, applying the corresponding data augmentation techniques to the training set.

10.Train the LeNet network using the augmented training set and the specified training options.

1)case 'NoAugmentation'

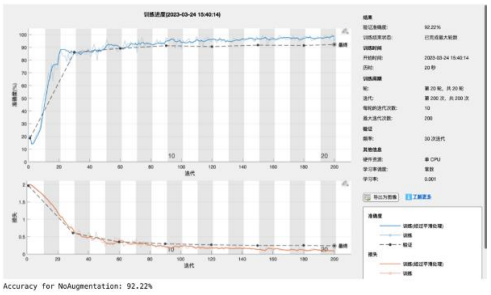


Figure 2.19 'NoAugmentation'

2)case 'HorizontalFlip'

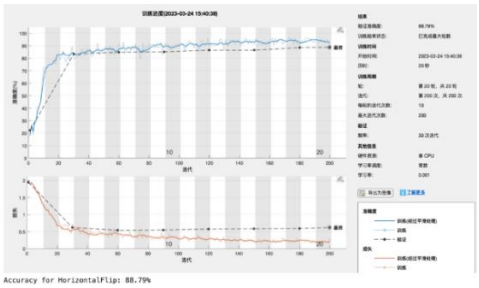


Figure 2.20 'HorizontalFlip'

3)case 'HorizontalFlipAndRotation'

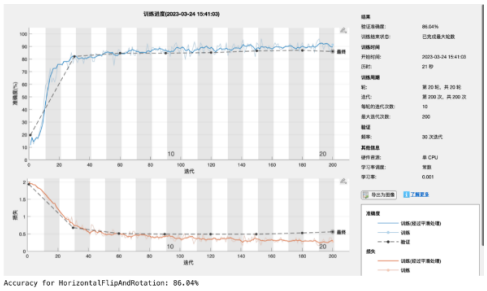


Figure 2.21 'HorizontalFlipAndRotation'

4)case 'AllAugmentations'

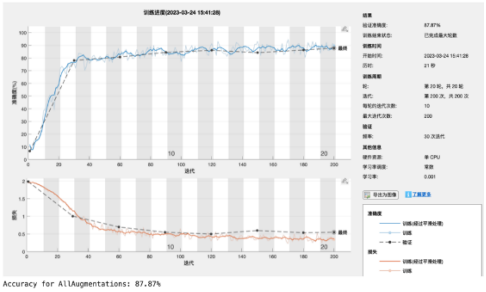


Figure 2.22 'AllAugmentations'

11.Evaluate the trained network on the validation set by calculating the prediction accuracy.

12.Store the validation accuracy for the current augmentation scenario and print the result.

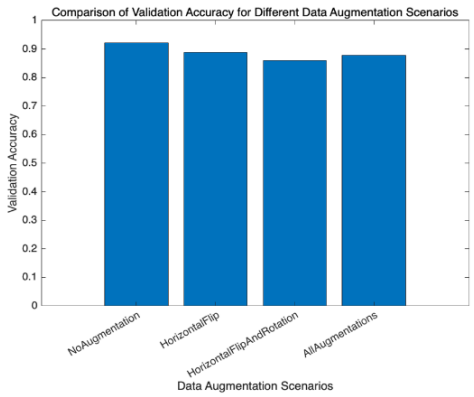


Figure 2.23 Validation Accuracy

2.2.2.2 Influence of hyperparameter tuning

The performance of both CNN and MLP models can be highly sensitive to hyperparameter tuning. Some critical hyperparameters include the learning rate, batch size, number of layers, and the number of units in each layer. Tuning these hyperparameters can significantly impact the models' performance and training time.

```
% parameter settings
options= trainingOptions('sgdm', ...
                         'InitialLearnRate',0.001, ...
                         'MaxEpochs',20, ...
                         'Shuffle','every-epoch', ...
                         'ValidationData',augimdsValidation, ...
                         'ValidationFrequency',30, ...
                         'Verbose',true, ...
                         'Plots','training-progress');
```

Description for the current parameters we already choosed:

1.'InitialLearnRate',0.001:

This sets the initial learning rate for the stochastic gradient descent with momentum (SGDM) optimizer. The learning rate determines the step size taken during each update of the model weights. A smaller learning rate might lead to slower convergence, while a larger learning rate can cause the model to overshoot the optimal solution.

2.'MaxEpochs',20:

This sets the maximum number of complete passes through the entire training dataset. An epoch is one pass through the entire dataset. A larger number of epochs might lead to better model performance but can also increase the risk of overfitting and require more

training time.

3.'Shuffle','every-epoch':

This specifies that the training data should be shuffled at the beginning of each epoch. Shuffling the data can help prevent the model from getting stuck in local minima and improve generalization.

4.'ValidationData',augimdsValidation:

This sets the validation dataset to be used during training. The validation dataset is used to evaluate the model's performance at regular intervals during training and can help monitor overfitting.

5.'ValidationFrequency',30:

This sets the number of iterations between evaluations of the model on the validation dataset. A lower validation frequency means the model will be evaluated

3 Discussion

The ability or tools to accurately separate and label various objects in images has numerous applications nowadays. In this project, we aimed to explore image processing and machine learning techniques to recognize and classify characters in an image.

During the course of our work, it is apparent that for many of the steps, the solution involved is not unique. Therefore, it is important for us to carefully select the most appropriate method based on the specifics of the problem at hand, according to our investigation. The insights garnered from this experience are invaluable and have motivated us to discuss them in detail within this section.

3.1 Part 1

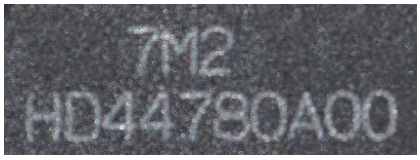
For the first part (tasks 1 - 6), the primary objective is to process an image to extract the characters and segment them into individual units. Different techniques are combined and applied for this purpose, including the use of averaging and rotating masks of varying sizes, thresholding, outlining, and labelling of characters. To the end, we obtained a clean and well-segmented image of the required characters, which can be used for further analysis and processing.

3.1.1 masks of different kinds and sizes

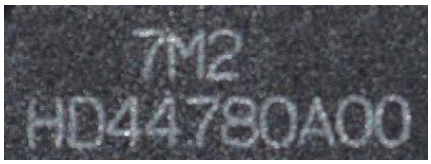
In task 2, when experimenting with masks of different sizes (3×3, 5×5 and 7×7), it can be found that larger masks often produce a smoother appearance but can also blur the image or reduce the level of detail. Smaller masks can highlight more specific features but may not produce as much smoothing or noise reduction.

In the realm of image processing, it is a common practice to assess the impact of different masks on the output image, and then make a well-informed selection based on the intended application. By doing so, it is possible to achieve desired results that cater to the specific requirements of the problem[1].

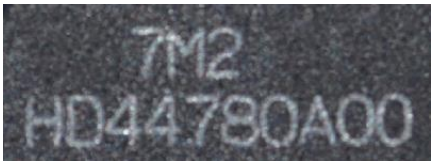
The comparison of the impact of diverse sizes and types of masks when applied to the original image is shown in Figures 3.1.



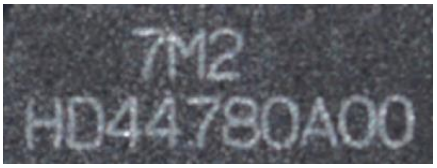
(a)Origin



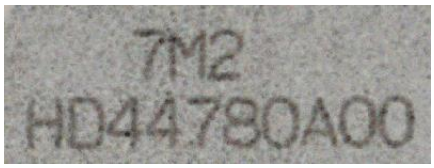
(b)Mask_Avg_3



(c)Mask_Avg_5



(d)Mask_Avg_7



(e)Mask_Rot_3

(f)Mask_Rot_5

(g)Mask_Rot_7

Figure 3.1 Comparison of Mask

3.1.2 The calculation methods of threshold

In task 4 (section 2.1.4), we established a manually specified value as the threshold, and explained the reasons for doing so. As a clarification, in the event that the 'graythresh' function was used to calculate the threshold value.

```
grayImage = rgb2gray(origin_image);
graySubImage = rgb2gray(sub_binary_image);
%Calculate global image threshold
thresholdValue1 = graythresh(grayImage);
thresholdValue2 = graythresh(graySubImage);
% Binary
binaryImage1 = imbinarize(grayImage, thresholdValue1);
binaryImage2 = imbinarize(graySubImage, thresholdValue2);
```

Then the binary image shown in Figure 3.2 is obtained.

Figure 3.2 Binary Image

Evidently, the level of noise present in Figure 18 is notably greater than that observed in the output image of our program.

The 'graythresh' function here utilizes Otsu's method. This method is a commonly employed image thresholding technique that is effective in cases where the histogram of an image exhibits a bimodal distribution with a distinct and pronounced valley between the two peaks[7]. However, this method may not yield desirable results when dealing with images that are affected by significant levels of noise, contain small objects, exhibit uneven illumination, or display a larger degree of intra-class variance as compared to inter-class variance. In such scenarios, the performance of Otsu's method may be suboptimal and alternative approaches may need to be considered.

There exist two additional functions, namely otsuthresh and adaptthresh, which can be utilized for the purpose of calculating threshold in MATLAB. The otsuthresh function, akin to graythresh, employs the Otsu's method to compute the threshold value. However, the key distinction between the two functions lies in the fact that graythresh computes the image histogram with 256 bins prior to invoking otsuthresh, which subsequently performs the threshold calculation using the Otsu's method.

The adaptthresh function, however, chooses the threshold based on the local mean intensity (first-order statistics) in the neighborhood of each pixel[8]. As a comparison, in the event that the 'adaptthresh' function was used to calculate the threshold value.

```
I = imread('charact2.bmp');
% Read the original image
grayI = rgb2gray(I);
T = adaptthresh(grayI, 0.5);
% Here we use the default sensitivity (0.5).
Binary_image = imbinarize(grayI,T);
figure
imshow(Binary_image)
```

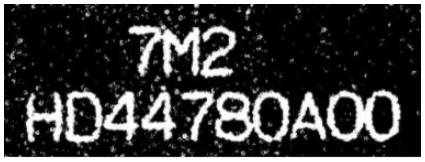
Here, T = adaptthresh(I,sensitivity) calculates a locally adaptive threshold for image segmentation, with the degree of sensitivity towards thresholding more pixels as foreground specified by the sensitivity factor. This 'sensitivity' parameter is a scalar value within the range of [0,1], which indicates the extent to which the algorithm should prioritize thresholding additional pixels as foreground.

The output image of this function is shown in Figure 3.3.

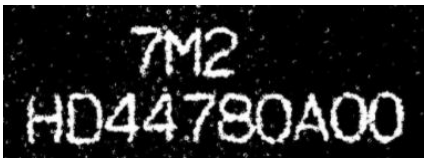
Figure 3.3 Output Image

Clearly, the binarization effect illustrated in Figure 3.3 is suboptimal for the purposes of this project. This inadequacy is likely to impede the efficacy of subsequent processing efforts.

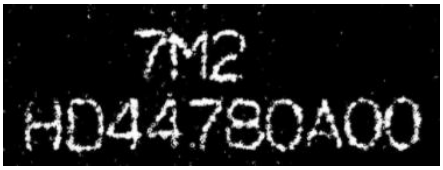
Therefore, here we use a manually specified value as the threshold. In order to explicate the manual adjustment process in our project, a range of threshold values specified manually are presented alongside their corresponding results.



(a)Threshold: 110



(b)Threshold: 120



(c)Threshold: 130

Figure 3.4 Comparison of Threshold

Upon examining this comparison group, it becomes apparent that a threshold value of 120 is the most optimal. Specifically, a threshold value of 110 results in excessive noise while a threshold of 130 has an adverse impact on the foreground of the image (i.e., the text).

3.1.3 Different effects of 'bwperim' and 'edge' functions

Drawing upon the theoretical principles outlined in section 2.1.5, we conducted a series of experiments to evaluate the efficacy of the 'bwperim' and 'edge' functions. The resulting outcomes are presented in Figure 3.5.

(a) The output of 'edge' function

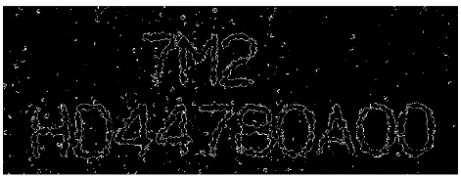
(b) The output of 'bwperim' function

Figure 3.5 Different output

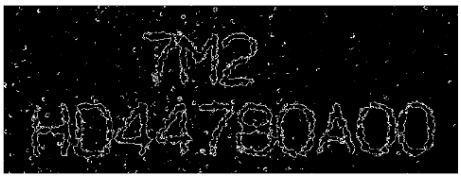
Our findings reveal that, for the present project, the 'bwperim' function outperforms the 'edge' function with regards to the outlining stage.

3.1.4 Different Pixel Connectivities

In task 5, the 'bwperim' function regards a pixel as part of the perimeter if it is nonzero and it is connected to at least one zero-valued pixel. Therefore, the connectivity should be determined[9]. In the Image Processing Toolbox of MATLAB, the default connectivity is set to 4 for two dimensions. Here, we investigate the impact of this parameter on image processing by comparing the effects of setting connectivity to 4, 8 and 18, respectively. The output images are shown in Figures 3.6.



(a) Outline (Connectivity = 4)



(b) Outline (Connectivity = 8)

(c) Outline (Connectivity = 18)

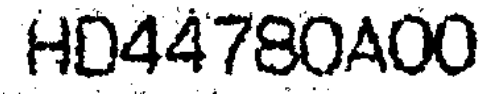
Figure 3.6 Outlines

It can be observed that as the connectivity parameter increases, more and more pixels are considered as outlines. When the connectivity is 4, the outline is very thin; When it is set to 18, the output 'outline' looks almost as thick as the original characters themselves. Therefore, according to the above comparison, the most suitable connectivity should be 8.

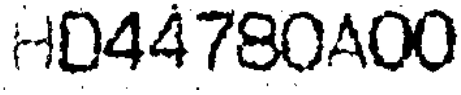
In practical applications, appropriate connectivity values should be selected based on the requirements of the problem and through multiple experiments.

The Gaussian filter, a type of linear filter employed in image processing and computer vision, is frequently employed to blur images, while simultaneously reducing the noise points. One of its distinguishing features is its capacity to preserve edges within the image while simultaneously diminishing high-frequency noise. The filter functions via convolution of the input image with a Gaussian kernel, which is a matrix of values that determine the weighting of neighboring pixels. The extent to which the image is smoothed is influenced by the standard deviation of the Gaussian kernel: the larger the value of the standard deviation, the more pronounced the degree of smoothing. The Gaussian filter is extensively utilized in diverse applications such as image enhancement, feature extraction, and image segmentation [10].

It is apparent that the Gaussian filter is of crucial significance in this project. Regarding the task 6, the outcomes obtained with and without the Gaussian filter (with a filter size of 1 and 3) are compared and shown in Figures 3.7, respectively.



(a)No Gaussian filter applied



(b)Filter Size = 1

(c)Filter Size = 3

Figure 3.7 Gaussian filter

It can be seen that the noise is significantly reduced after using a Gaussian filter.

3.1.6 Different Filters

During our investigation, we discovered that numerous image filters are at one's disposal, and our software finally selected a Gaussian filter. To delve deeper into this, here we scrutinize the impact of various alternative filters that can be executed in MATLAB.

The function 'fspecial' creates a two-dimensional filter of the specified type. Then 'imfilter' filters the a multidimensional array with the multidimensional filter (both as inputs) and returns the result. The following comparison is done mainly with these two functions.

In this section, we take 'binary.png', which is the binary image created in task 4, as the input.

A. Motion Filter

Motion Filter is utilized to approximate the linear motion of a camera. The code to implement this filter is given below.

```
I = imread('charact2.bmp');
% Read the original image
```

```
grayI = rgb2gray(I);
T = adaptthresh(grayI, 0.5);
% Here we use the default sensitivity (0.5).
Binary_image = imbinarize(grayI,T);
figure
imshow(Binary_image)
% =  Start of Code  =
% Read image file and display
I = imread('binary.png');
figure(1);
imshow(I);
% Create a motion filter and blur the image
H = fspecial('motion',30,0);
% H = fspecial('motion',length,theta), where 'length' specifies the length of the motion
% and 'theta' specifies the angle of motion in degrees in a counter-clockwise direction.
% The default theta is 0, which corresponds to a horizontal motion of 9 pixels.
Blurred = imfilter(I,H,'replicate');

% The 'replicate' here is an option that controls the filtering operation. It denotes an
% assumption that values beyond the bounds of the input array are deemed to be
% equivalent to the nearest border value of the array.
figure(2);
imshow(Blurred);
% =  End of Code  =
```

The output image is shown in Figure 3.8.

Figure 3.8 The effect of Motion Filter

B. Disk Filter

Disk Filter is also called circular averaging filter. It takes the average of all elements in a circular window around each location.

To save space, the following code omits the part that reads the image file, which is the same as part A (Motion Filter).

```
% =  Start of Code  =
% Create a disk filter and blur the image
H = fspecial('disk',10);
% H = fspecial('disk',radius) returns a circular averaging filter within the
% square matrix of size 2*radius+1.
blurred = imfilter(I,H,'replicate');

figure(3);
imshow(blurred);
% =  End of Code  =
```

The output image is shown in Figure 3.9.

Figure 3.9 The effect of Disk Filter

C. Gaussian Lowpass Filter

The Lowpass Gaussian Filter is designed to remove high frequency features that are aligned with either the X or Y axis of the scan. Consequently, the filter induces a loss of detail, colloquially referred to as a "blurring" effect.

Comparatively, the Gaussian filter has the ability to smooth features running parallel to an image's Y scan axis while preserving those aligned with the X axis, or vice versa [11].

```
% =  Start of Code  =
% Create a Gaussian lowpass filter and blur the image
H = fspecial('gaussian',10,3);
% H = fspecial('gaussian',hsize,sigma) returns a rotationally symmetric
```

```
% Gaussian lowpass filter of size hsi
ze with standard deviation sigma.
blurred = imfilter(I,H,'replicate');

figure(4);
imshow(blurred);
% =  End of Code  =
```

The output image is shown in Figure 3.10.

Figure 3.10 Gaussian Lowpass Filter

D. Laplacian filter

The Laplacian filter is a type of edge detector that operates by computing the second derivatives of an image, measuring the rate of change of the first derivatives[12].

```
% =  Start of Code  =
% Create a Laplacian filter and blur
the image
H = fspecial('laplacian',0.2);
% H = fspecial('laplacian',alpha) ret
urns a 3-by-
3 filter approximating the shape of the
 two-
dimensional Laplacian operator. 'alpha'
, specified as a number in the range [0
, 1], controls the shape of the Laplaci
an.
blurred = imfilter(I,H,'replicate');

figure(5);
imshow(blurred);
% =  End of Code  =
```

The output image is shown in Figure 3.11.

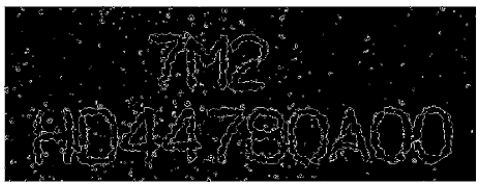


Figure 3.11 Laplacian Filter

Furthermore, the comparison also encompassed the utilization of a rotationally symmetric Laplacian of Gaussian filter and an averaging filter. The respective outcomes of these filters are depicted in Figures 3.12 and 3.14. Due to space limitations, the associated codes have been omitted, because the coding methodology almost remains consistent with the overall approach.

Figure 3.12 The effect of Rotationally Symmetric Laplacian of Gaussian Filter

Figure 3.13 The effect of Averaging Filter

For each of the aforementioned filters, we have conducted multiple trials using various sets of parameters. In the interest of concision, these results are not reiterated in this section.

Taking all these comparisons into account, we finally concluded that a Gaussian filter would show the best result for this project (as shown in Figure 30 in Section 3.1.5).

3.2 Part 2

3.2.1 The Principle and Effect of CNN and MLP in Image Classification

Convolutional Neural Networks (CNNs): CNNs are particularly well-suited for image classification tasks due to their ability to recognize local patterns within an image. They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

Convolutional layers: These layers apply a set of filters (or kernels) to the input image, capturing local patterns such as edges, corners, and textures. By stacking multiple

convolutional layers, the network can learn to recognize increasingly complex patterns.

Pooling layers: These layers reduce the spatial dimensions of the input, which helps to reduce the number of parameters in the network, minimize computational requirements, and prevent overfitting.

Fully connected layers: These layers are typically placed at the end of the network and are responsible for combining the learned features from the previous layers to make the final classification decision.

Multi-Layer Perceptrons (MLPs): MLPs, also known as feedforward neural networks, consist of multiple layers of fully connected neurons. Each neuron in a layer receives input from all neurons in the previous layer and computes a weighted sum followed by an activation function.

3.2.2 Comparation of the effectiveness and efficiency of the two approaches

Results from Sub-task 1 and Sub-task 2 show that both the Convolutional Neural Network (CNN) and the Multi-Layer Perceptron (MLP) models are capable of classifying the given character images. However, the CNN model yields slightly better results compared to the MLP model.

Effectiveness: The superior performance of the CNN can be attributed to its ability to learn hierarchical feature representations from raw input data. The convolutional layers in the CNN architecture are designed to capture local patterns, such as edges and textures, while pooling layers help reduce spatial dimensions and maintain important information. This hierarchical feature extraction makes CNNs particularly well-suited for image processing tasks. In contrast, MLPs treat input data as a flat vector, which can limit their ability to capture spatial information and recognize patterns effectively.

Efficiency: CNNs are generally more computationally efficient than MLPs when it comes to image processing tasks. This is because CNNs utilize weight sharing and parameter reduction through convolution and pooling operations, which help in reducing the number of trainable parameters. On the other hand, MLPs can have a large number of parameters due to fully connected layers, making them more prone to overfitting and requiring more computational resources.

In conclusion, the differences in the results between the two approaches can be explained by the inherent advantages of CNNs in handling image data. The CNN model's ability to learn hierarchical feature representations and efficiently capture spatial information leads to its slightly better performance compared to the MLP model for this character classification task.

3.2.3 Analysis of slightly reduced accuracy after preprocessing

Preprocessing is the process of cleaning, transforming, and organizing data before inputting it into a model. The purpose of preprocessing is to improve data quality, reduce noise, enhance the generalization ability of the model, and adapt the data to specific machine learning algorithms.

In task 8, we used four different data preprocessing methods and conducted a comparative experiment with the results of the control group that did not undergo preprocessing. However, from the accuracy histogram we obtained, we can clearly observe that after data processing, the accuracy obtained by each method decreased to varying degrees for the untreated group. After discussion, Our group believes that the following reasons may occur:

Information loss: Information important for classification tasks may be lost during preprocessing. For example, when sampling an image, some details may be lost that are critical

for correct classification.

Data distortion: Pre processing can cause data distortion, which can affect model performance. For example, when filling or resizing an image, the shape or position of certain features may change, making it difficult for the model to correctly recognize these features. To solve this problem, you can try using different fill or resize methods to reduce the impact of data distortion.

In conclusion, although preprocessing data increases the acceptance of different types of data by the model and increases the generalization ability of the model, which can enable our newly trained neural network to better adapt to different types of application scenarios, it does not necessarily mean that it will perform better in specific application scenarios. Therefore, when we conduct model training for specific data, It is not necessary to blindly adopt various data preprocessing methods as much as possible, but rather to conduct specific problem specific analysis for specific models.

4 Conclusion

This project involves image processing and classification. The initial tasks include applying different masks, creating binary images, determining outlines, and segmenting images. The latter tasks involve creating and comparing two classification systems, using CNN and MLP, for characters in the image. The experiments involve pre-processing the data and tuning hyperparameters, and the sensitivity to changes is discussed.

This project requires a comprehensive understanding of different image processing techniques and classification methods. By the end of this project, we have developed a sound knowledge of image processing and character recognition, which can be applied to solve various computer vision related problems.

Reference

[1] Sonka, M., Hlavac, V., Boyle, R., Sonka, M., Hlavac, V., & Boyle, R. (1993). Image pre-processing. *Image processing, analysis and machine vision*, 56-111.

[2] Haralick, R. M., Sternberg, S. R., & Zhuang, X. (1987). Image analysis using mathematical morphology. *IEEE transactions on pattern analysis and machine intelligence*, (4), 532-550.

[3] Kazlouski, A., & Sadykhov, R. K. (2014, June). Plain objects detection in image based on a contour tracing algorithm in a binary image. In *2014 IEEE International Symposium on Innovations in Intelligent Systems and Applications (INISTA) Proceedings* (pp. 242-248). IEEE.

[4] Gonzalez, R. C., Woods, R. E., & Eddins, S. L. (2004). Digital image processing using MATLAB. Gatesmark Publishing.

[5] Pal, N. R., & Pal, S. K. (1993). A review on image segmentation techniques. *Pattern recognition*, *26*(9), 1277-1294.

[6] Smith, S. T. (2006). *MATLAB: advanced GUI development*. Dog ear publishing.

[7] Otsu, N. (1979). A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics*, *9*(1), 62-66.

[8] Bogiatzis, A. C., & Papadopoulos, B. K. (2018, July). Binarization of texts with varying lighting conditions using fuzzy inclusion and entropy measures. In *AIP Conference Proceedings* (Vol. 1978, No. 1, p. 290006). AIP Publishing LLC.

[9] Fontaine, M., Macaire, L., & Postaire, J. G. (2000, September). Image segmentation based on an original multiscale analysis of the pixel connectivity properties. In *Proceedings 2000 International Conference on Image Processing (Cat. No. 00CH37101)* (Vol. 1, pp. 804-807). IEEE.

[10] Kumar, A., & Sodhi, S. S. (2020, March). Comparative analysis of gaussian filter, median filter and denoise autoenocoder. In *2020 7th International Conference on Computing for Sustainable Global Development (INDIACom)* (pp. 45-51). IEEE.

[11] Qu, F., Ren, D., Liu, X., Jing, Z., & Yan, L. (2012, October). A face image illumination quality evaluation method based on Gaussian low-pass filter. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems* (Vol. 1, pp. 176-180). IEEE.

[12] Aubry, M., Paris, S., Hasinoff, S. W., Kautz, J., & Durand, F. (2014). Fast local laplacian filters: Theory and applications. *ACM Transactions on Graphics (TOG)*, *33*(5), 1-14.