ME5411 Robot Vision and AI Computing Project

AY24/25 Semester 1

Professors：

Chen Chao Yu, Peter
Chui Chee Kong

Group 27 members：

Duan Yutong	A0304370R
Gao Yilin	A0303608L
Shen Yusen	A0304452N

CONTENTS

Task 1: Contrast Enhancement

Display the original image on screen. Experiment with contrast enhancement of the image. Comment on the results.

1.1 An Introduction of Task

The purpose of Task 1 is to read and display a color raw image, afterwards convert it into a grayscale image, and eventually adjust the contrast of this grayscale image.

1.2 Description of Algorithms

1.2.1 Loading Images and Function Libraries

Before the beginning of the task, it is needed to create two folders, *functions* and *pictures*, and add the directory path *functions* (containing the customized functions) to MATLAB's search path via *addpath*, and specify the file path of the original pictures as *". /pictures/charact2.jpg"* and store it in the variable *picture* (Figure 1).

Figure 1 charact2.jpg

First, the image in the picture path is loaded using the *imread* function, and then stored in *origin_image*. Then it creates a new window to display the image by *figure* function, and then uses the *imshow* function to display the original image *charact2.jpg*, which is stored in *origin_image*. After that, this new color image is converted into grayscale image, which means *origin_image* is converted to grayscale with *rgb2gray* function and finally stored in the variable *image*.

1.2.2 Customizing Contrast Function

The next step is to perform the definition of the contrast function. This task adopts the contrast stretching method and customizes the *contrast* function, which receives three parameters: *image*, the input grayscale image; x, the offset value of the contrast, which is used to control the offset of the dark and light areas of the image; and y, the contrast factor, which is used to enhance the contrast effect of the image.

Then the function *min* is called to get the minimum pixel value in the image and the *max* function is used to get the maximum pixel value in the image to facilitate the normalization process in the next step. The normalization operation is performed by

adjusting the pixel values of the image to the range of [0,1], where the minimum and maximum values become 0 and 1, as a way of removing the original brightness difference in the image. The normalized pixel values are then subtracted from the overall value of x by applying an offset value of x. The range of pixel values will then become [-x, 1-x]. The purpose of this is that when the contrast factor y is adjusted next, it will make the darker areas darker and the brighter areas brighter, thus improving the contrast. And the larger the y value, the greater the enhancement of the light and dark areas, and the greater the difference between light and dark, resulting in higher contrast.

However, pixel values outside the [-x, 1-x] range can then be cropped out: values less than -x are set to 0 (black), and values greater than 1-x are set to 255 (white). This step eliminates extreme pixel values and keeps the image in the proper brightness range. Immediately after this, the range of pixel values in [-x, 1-x] is restored to [0, 1] multiplied by 255 to get the standard image format. Finally, the incoming image needs to be converted back to the *double* type in order to perform the subsequent math operations. The mathematical expression for contrast stretching is shown below:

1) Apply normalization, brightness offset x and contrast factor y:

$$I_{contrast} = \left(\frac{I - I_{min}}{I_{max} - I_{min}} - x\right) \times y$$

2) Limit the range so that pixel values less than $-x$ are considered black, and pixels greater than $1-x$ are considered white:

$$I_{cropped} = min(max(I_{contrast}, -x), 1 - x)$$

3) Shift the pixel values back to the range [0,1] and convert back to [0,255]:

$$I_{new} = \left(I_{cropped} + x\right) \times 255$$

The final result I_{new} is the resized image, controlled by brightness offset x and contrast factor y.

1.3 Discussion

Figure 2 First Contrast Adjustment

Figure 3 Second Contrast Adjustment

Two contrast adjustment has been tested for the original picture. The first contrast adjustment invokes the custom function *contrast* and sets the parameters to $x=1$ and $y=1$ for basic contrast adjustment, while the second time, a substantial adjustment is conducted, setting the parameters to $x=0.4$ and $y=5$ respectively. Comparing the results of the two operations, we can intuitively observe that there is no obvious change in the contrast of image after the first parameter adjustment (Figure 2). Therefore, parameters $x=1$ and $y=1$ do not lead to much difference between the image and the original one. However, the contrast of the image (Figure 3) is significantly increased by the second parameterization $x=0.4$ and $y=5$. The difference between the bright and dark parts is very obvious, in which the bright parts are brighter and the dark parts are darker.

1.4 Conclusion

Overall, the contrast stretching method is simple but effective and fast to compute. It can effectively improve the visualization of low-contrast images through linear scaling. Although this method is not suitable for images with complex details and multiple brightness ranges, it is definitely a fast and efficient method for the original images in this task. From the output results, the visibility and readability of the images are significantly enhanced.

Task 2: Averaging Filter

Implement and apply a 5x5 averaging filter to the image. Experiment with filters of different sizes. Compare and comment on the results of the respective image smoothing methods.

2.1 An Introduction of Task

This task is to apply a 5x5 averaging filter to the imported image, after which adjust different parameters, compare and discuss the output. In the same way as in Task 1, first read and display a color original image and convert it to grayscale. Then invoke the custom function *average_filter* to apply an average filter. Next it can be set into different filter sizes, such as 5x5, 9x9, 11x11, etc. And eventually use *figure* and *imshow* to create a new window and display the adjusted image.

2.2 Description of Algorithms

2.2.1 Converting Image Types and Initializing

Defining a function on the average filter for an image is the key to this task. So set this function named *average_filter* and store it in function *image*, which takes two input parameters: the image to be processed (*image*) and the size of the filter (x). The incoming image then needs to be converted to a *double* type in order to perform the subsequent math operations. This is because the original image is usually stored as a *uint8* type (each pixel is an integer in the range 0-255), but *float* operations are required in the filtering calculation. And *double* is a double precision float type, especially in the next convolution operation can represent more precise values, so that the pixels of the image can be smoothed in the convolution. It is also necessary to capture the dimensions of the image, including the number of rows and columns, and when completed, it initializes the *result_image* that is the same size as original image, with initial values 0 of the number of rows and columns by *zeros* function.

2.2.2 The Kernel of Averaging Filter and Padding

Next it is essential to define the details of the filter. The first step is to create an averaging filter with a *kernel* of size x*x and all weights within the filter are 1/x^2, which sums to 1. The matrix of this kernel is:

$$K_x = \frac{1}{x^2}\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{x\times x}$$

The second step is to calculate the size of *padding*, which is equal to half of the filter's size minus the centroid (x-1)/2. (Figure 4) The edge padding is then added to the image so that it is larger than the original image in order to handle pixels on the edges even during convolution operations, otherwise pixels at the edges of the image would be inaccurately calculated due to a lack of sufficient neighborhood data. Therefore, the size of transformed image is equal to the length of the rows and columns respectively coupled with the size of the two edge paddings (*rows+2*padding, cols+2*padding*), which are then initialized with the zeros function and deposited in *padding_image*. At last, the center of the new padding image is aligned with the original image. The index range of the inserted part is defined as: *1+padding: rows+padding* means that in the filled matrix from *1+padding* rows to *rows+padding* rows. Similarly, *1+padding: cols+padding* means that in the filled matrix from *1+padding* columns to *cols+padding* columns. In this way, the pixel values of the original image are copied to the middle region of *padding_image* in the defined index range, while the rest of the region remains zero, ensuring that the convolution can operate on the entire image including the edges.

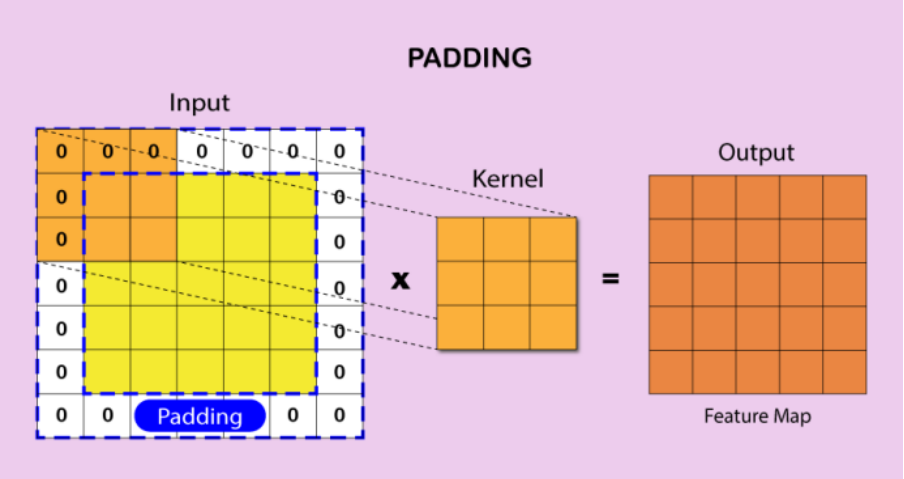


Figure 4 Kernel and padding operation

https://developersbreach.com/convolution-neural-network-deep-learning/

Taking a 5*5 averaging filter as an example, each weight within this filter is 1/25, with all weights summing to 1. Its padding is (5-1)/2, which is equal to 2, so the transformed image size is (rows+4, cols+4). This results in the original image being aligned to the new image with the edges filled in, with the index range starting at row 3, column 3 and ending at rows+2, cols+2, so as to preprocess an image for subsequent operations.

$$K_5 = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

2.2.3 The Convolution Operation

The core step, which is the convolution operation on the prepared image (Figure 5), proceeds immediately afterward. This step uses a nested loop to traverse each pixel in the indexed region of *padding_image* (index range for row *i*: *1+padding:rows+padding*, index range for column *j*: *1+padding:cols+padding*). Then take out the x*x size region centered on the current pixel one by one (from *i-padding* to *i+padding* rows, from *j-padding* to *j+padding* columns) and stored in the variable *area*. After that, calculate the weighted sum of each *area* and *kernel* row by row and column by column in order to get the filtered pixel value *filtered_value* and store it into the corresponding position (*i-padding, j-padding*) in *result_image*.

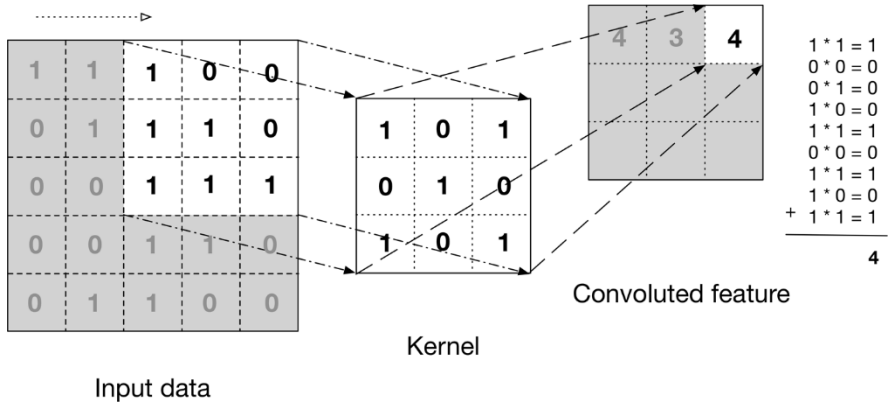


Figure 5 Example of a convolution operation

https://www.davidsbatista.net/blog/2018/03/31/SentenceClassificationConvNets/

Still take the 5*5 averaging filter as an example, assuming that the pixel (1, 1) in the first row and column of the original image is currently being processed, then the index position in *padding_image* is (3, 3). Next take out the 5*5 size region centered on the current pixel (from row 1 to row 5, from column 1 to column 5) and calculate the weighted sum with kernel. The final *filtered_value* will correspond to the position of the original image (1, 1) into the corresponding position of the *result_image* (in particular, it should be noted that the size of the *result_image* and the original image is same, both are missing the padding part compared to *padding_image*).

2.2.4 Converting Back to uint8 Type

After processing is complete, the *result_image* of type *double* should be converted back to type *uint8* to be able to display the normal effect in the normal format. Then save the image into the variable *image*, so that the output image can be compatible with other image processing tools. The output image processed by 5*5 averaging filter is shown following (Figure 7).

2.3 Discussion

Figure 6 Origin Image

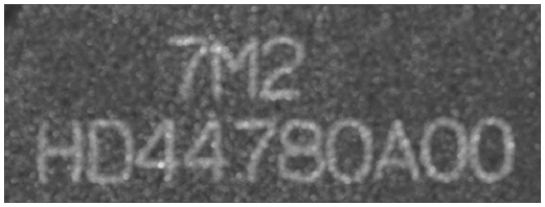


Figure 7 Image by 5x5 Averaging Filter

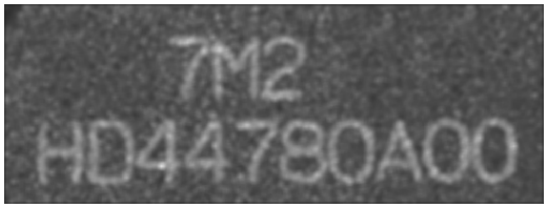


Figure 8 Image by 9x9 Averaging Filter

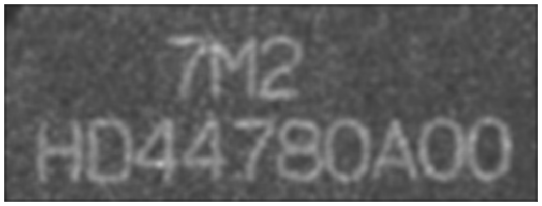


Figure 9 Image by 11x11 Averaging Filter

Filter Size	5x5	9x9	11x11
Smoothness	low	medium	high
Noise Effects	high	medium	low
Edge Blurriness	low	medium	high

With the output image, it is very intuitive to see that different sizes of mean value filters will have different effects on image processing, mainly in the three parts of the image, namely smoothness, noise effect and edges. In this task, three different sizes of filters, 5x5, 9x9 and 11x11, are selected respectively, and then the effect of image processing is observed and compared horizontally based on their output images. Comparing Figure 7 with the original image Figure 6, it can be noticed that the smoothing effect of the small-sized filters is relatively slight, and also only removes some slight noise, but retains more edge information, and the contours of the image subject are not blurred excessively. As the size of the filter gradually becomes larger, as shown in Figure 8 and Figure 9, the smoothing effect becomes stronger and stronger, while effectively attenuating the strong noise, but also consequently causing the contours of the objects in the image to become more indistinct.

Therefore, it is necessary to choose a suitable mean filter size according to the actual needs. If you want the image to be smoothed while retaining more image details, and if you want the computation to be low and the image to be processed faster, then a small filter size is preferred. If the image is noisy and a strong smoothing effect is desired, a larger filter is suitable, but the processing speed is slower and may not be suitable for real-time image processing.

2.4 Conclusion

In general, averaging filters are effective in removing high-frequency noise and smoothing the image, especially the small size filters are very fast in computation and can process the image quickly. However, the averaging filter leads to loss of details while smoothing the image. Especially when applying larger size filters, images with complex details cannot be finely processed, so more advanced filters may be needed to fulfill more requirements.

Task 3: High-Pass Filter

Implement and apply a high-pass filter on the image in the frequency domain. Compare and comment on the results and the resultant image in the spatial domain.

3.1 An Introduction of Task

This task 3 is to implement and apply a high-pass filter in the frequency domain and observe its effect on the input image. Firstly, import and display a color original image and convert it to a grayscale image. Then define a custom high-pass filter function *high_pass_filter* to process this image. After that, repeat the filtering operation under different parameter settings and show the image effect after each adjustment. Finally, compare and discuss the effects of different parameters, especially the visual effects in the spatial domain, so as to better understand the role and logic of the high-pass filter.

3.2 Description of Algorithms

3.2.1 Loading Paths and Image Preprocessing

Above all, it is necessary to load the custom function directory and invoke *imread* to load a required image *charact2.jpg*, the same as the method in Task 1 above. Then use the *rgb2gray* function to convert *origin_image* into a grayscale image to simplify the filtering process. And at last, pass the pre-processed image into the custom function *high_pass_filter* for subsequent image processing.

3.2.2 Customizing a High-Pass Filter

For task 3, a custom function named *high_pass_filter* is created, which takes two input parameters: *image*, the image to be filtered; *D0*, used to represent the cut-off frequency of the filter. Secondly, the logic of constructing this function is mainly achieved through the following four steps:

1) Fourier transform and move the frequency center,
2) calculate the center coordinates in the frequency domain,
3) generate and apply the filter,
4) restore the frequency center and inverse Fourier transform.

The explanation of the code will be explained step by step below.

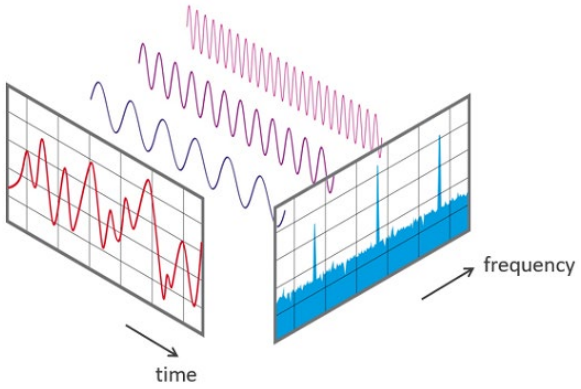
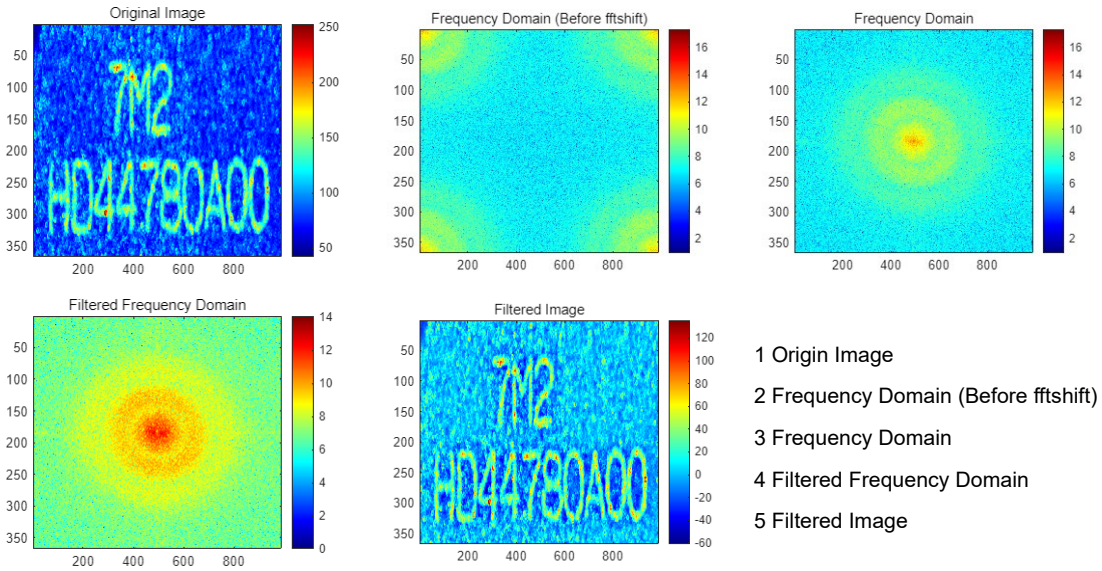


Figure 10 View of a signal in the time and frequency domain

https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft

The first step is Fourier transform and move the frequency center. Before starting, invoke the function *size* to get the number of rows *rows* and columns *cols* of the image, which represents the size of the image. Then convert the image to double floating format, and then use the *fft2* function to perform a two-dimensional Fourier transform on the image, converting the image from the spatial domain to the frequency domain (Figure 10) so that high-frequency filter can be applied in the frequency domain. Then use the *fftshift* function to move the low-frequency part to the center of the spectrum, which makes it easier to observe the spectrum diagram.



The second step is to calculate the center coordinates in the frequency domain. By calling the *floor* function, inputting half of the number of rows and columns, we get the center coordinates of the image *(rows_mid, cols_mid)*. The reason of doing this operation is because the center coordinates of image are the zero point of frequency, which is also the low-frequency part.

The third step is to generate and apply the filter, which is also the most critical step. This part of the code will create a matrix h through a nested *for* loop to store the filter coefficients. First, use *zeros(rows, cols)* to initialize the matrix, then traverse each pixel *(i, j)* in the image, and calculate the square of its distance to the center of the image. The mathematical expression is:

$$d = (i - rows_{mid})^2 + (j - cols_{mid})^2$$

Then, according to the formula of Gaussian high-pass filter, calculate the filter coefficient at corresponding position. Its expression is:

$$h(i,j) = 1 - e^{-\frac{d}{2D_0^2}}$$

Where *D0* is the cut-off frequency of the filter. The larger the value, the less the filter weakens the low frequency. In this way, the low-frequency components can be removed and the high-frequency details can be retained. Finally, the filter matrix h is multiplied by the image spectrum *frq_d* to complete the filtering operation and obtain *filtered_freq*. The formula is:

$$F(u,v)_{new} = H(u,v) \cdot F(u,v)$$

In the frequency domain, it is notable that the transform of convolutional operation can be simplified as the multiplication of h and *frq_d*.

The last step is to perform two inverse operations in sequence: restore the frequency center and perform inverse Fourier transform. So invoke the *ifftshift* function to first move the spectrum back to its original position. Then use *ifft2* to carry out an inverse two-dimensional Fourier transform to transform the image from the frequency domain back to the spatial domain. Finally, call the *real* function to extract the real part of the transformation result to obtain the filtered image *image*.

3.2.3 Outputting Results

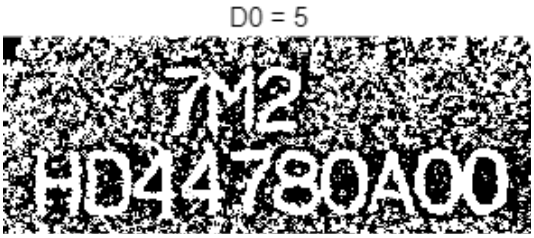


Figure 11 Non-normalized Output Image

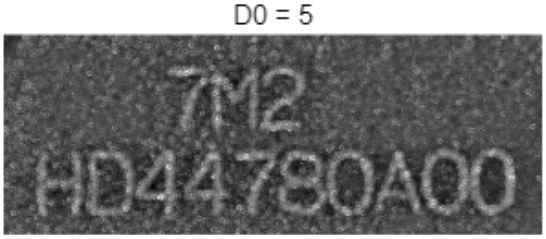


Figure 12 Normalized Output Image

Finally, use *figure* and *imshow* to display the original image, non-normalized and normalized processed image respectively.

3.3 Discussion

3.3.1 Setting the Cut-off Frequency D0

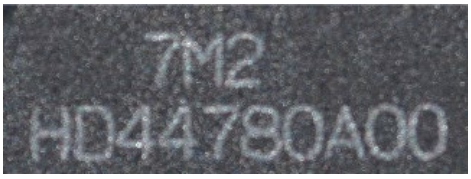


Figure 13 Origin image

Figure 14 Cut-off frequency D0 = 5

Figure 15 Cut-off frequency D0 = 10

Figure 16 Cut-off frequency D0 = 15

Cut-off Frequency(D0)	D0 = 5	D0 = 10	D0 = 15
Smoothness	Sharp	Relatively sharp	Relatively smooth
Noise Effects	High	Medium	Low
Edge Blurriness	Enhanced	Moderately enhanced	Slightly enhanced

By setting different sizes of cut-off frequency D0, we can observe and compare the corresponding output images. If D0 value is smaller, the filter will remove low-frequency components more strictly, retaining only higher-frequency details and edges, making the edges and textures of the image more obvious, but it may also amplify noise, causing the image to look sharper or even rougher. A larger D0 value allows more low-frequency components to pass through, making the filtering effect softer and not over-enhancing edge details, thereby maintaining the naturalness of the image. Therefore, the cut-off frequency D0 is a key parameter of the high-pass filter, which directly determines the degree to which the filter retains or suppresses the frequency components in the image. Only by selecting a suitable D0 to balance edge enhancement and noise suppression, can a relatively better filtering effect be obtained. For this image, it is most appropriate to choose 10 as the filter size.

3.3.2 Averaging and High-Pass Filter Comparison

Filter Type	Averaging Filter	High-pass Filter
Smoothness	Smoother	Sharper
Noise Effects	Noise reduced	Noise saved
Edge Blurriness	Blurrier	More clarity

In order to compare the high-pass filter with the averaging filter in the previous task, we analyzed and observed the principles and output results of these two different filters. It can be seen that their filtering types are very different. The high-pass filter focuses on retaining high-frequency components, while the averaging filter can essentially be classified as a low-pass filter, which retains low-frequency components.

According to the output pictures, their functions and effects are also very different. The high-pass filter suppresses low frequencies and enhances the edges and details in the image, making the edges clearer and the image sharper. In contrast, the average filter calculates the average value of pixels in the neighborhood, removes high-frequency noise, and makes the image smoother and softer, but the details will be weakened. Therefore, the high-pass filter is more suitable for image processing tasks that enhance details, while the average filter is more suitable for scenes that require noise reduction and smoothing effects.

3.4 Conclusion

The custom high-pass filter function for this task uses Gaussian high-pass filtering to focus on enhancing the high-frequency components in the image. By setting the cut-off frequency D0, the filter can effectively suppress low-frequency components while retaining and highlighting the edges and details in the image. Unlike traditional average filters, it is not used to remove noise or smooth images, but to highlight the structure and texture information of the image. Therefore, the high-pass filter is very suitable for tasks that require enhanced edge features, such as edge detection, feature extraction, or image enhancement, which helps to obtain clear contours and details in detail-rich image processing.

Task 4: Image Cropping

Create a sub-image that includes the middle line – HD44780A00.

4.1 An Introduction of Task

The requirement of Task 4 is to crop a sub-image from the lower half of the original image, but includes the middle line - HD44780A00. The overall logic is not complicated, it involves reading the image, then cropping the lower half, and finally displaying and saving it as a new image.

4.2 Description of Algorithms

Consistent with the approach of the previous tasks, the file path of the original image is first stored in the variable *picture*, and then the *imread* function is applied to read the file, which is then stored in the variable *origin_image*. (Figure 17)

Figure 17 Origin Image

Next, start cropping the original image. First use function *size* to get the size of original image [rows, ~, ~], where rows represent the height of image, but not interest in the number of columns and channels of color. After obtaining the number of rows, it is time for the cropping operation. Here the lower half of the image is obtained by the *origin_image* (ceil(rows/2) : rows,:,:) statement, which means that it starts from the middle of all the rows of the image to the bottom of the image. Then save these rows as a new image. The *ceil* function means rounding up in Matlab and it will remove the fractional part and round the numeric part up to the nearest integer. In this task it starts with the row containing the center and extracting all rows to the end.

Figure 18 Cropped Image

In practice, the size of *charact2.jpg* is 367x990, half the number of rows is 183.5, and after *ceil* the number of rows is 184. The final cropped image is the size of the rows from the middle row 184 to the last row 367. Then use *imshow* to display the cropped image and *imwrite* to save image to specified path *". /pictures/sub-image.jpg"*.

4.3 Discussion & Conclusion

The code for this task is suitable for extracting the lower half of an image and saving it as a new file, allowing for quick local cropping of the image without focusing on the specific width and color channels of the image.

Task 5: Image Binarization

Convert the sub-image into a binary image.

5.1 An Introduction of Task

This section focuses on processing sub-regions of an image, aiming to convert a color image into a binary format to bring out its main features. The basic idea is to convert the original color image into a grayscale image and then apply a custom binarization function to display the image in black and white based on a set threshold. By sequentially performing grayscaling and binarization, the contours of the objects in the image can be highlighted and the effect of detail detection can be enhanced, laying a good foundation for subsequent object recognition and feature extraction.

5.2 Description of Algorithms

5.2.1 Loading Functions and Image Paths

The core code for this task consists of adding a function path, loading a sub-image (see Task 1 for detailed explanations), and then sequentially grayscaling the sub-image and binarizing it with a custom function. Finally, the new sub-image is displayed and saved. The main two steps, grayscaling and binarization, are explained in more detail below.

5.2.2 Image Grayscaling

The first step is grayscaling, which uses *rgb2gray* to convert a color image to grayscale. This operation is aiming to simplify the data and highlight the luminance information of the image, so this color image is converted into a single channel grayscale image (Figure 19). This function *rgb2gray* is a built-in Function in Matlab, which weights and averages the pixel values of the red (R), green (G), and blue (B) channels to get the new grayscale values. The formula is as follows:

$$Gray = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B$$

Here the weighting coefficients reflect the difference in the sensitivity of the human eye to different colors, i.e., it is more sensitive to green and less sensitive to blue, so the weighting coefficients for green are relatively large, followed by red and blue. The results are then stored in image BW to preprocess for the next binarization. Since the binarization requires only luminance information, the grayscale image is sufficient to satisfy its condition.

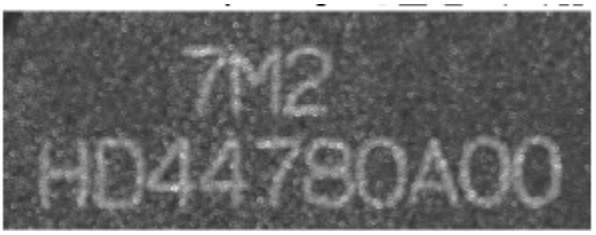


Figure 19 Grayscale image

5.2.3 Image Binarization

Next is the core of this task--binarization, which invokes a custom function to convert a grayscale image into a binary image. The binarization function is called *rgb2binary* and takes two arguments: *image*, the input image; *threshold*, the threshold for binarization, which is used to distinguish between black and white pixels. As before, the data type of the image is converted first, because images are usually stored as *unit8* in Matlab, and converting it to double-precision can make the calculation more accurate, and will not cause errors due to the integer limitation of the value. Then use the *size* function to get the rows and columns of the image, which are stored as *rows* and *cols* respectively. Next step, use a *for* loop to traverse each row and column of the image to get the information of each pixel. Also use an *if* conditional statement to check if the current pixel value is less than the set *threshold*. If the value of the pixel is less than the threshold then it is treated as low brightness and set to black (pixel value 0), otherwise the pixel is high brightness and set to white (pixel value 255). Immediately after judging each pixel, the image is converted from *double* type back to *uint8* type for subsequent display and saving operations.

Figure 20 Binary Image, threshold = 120

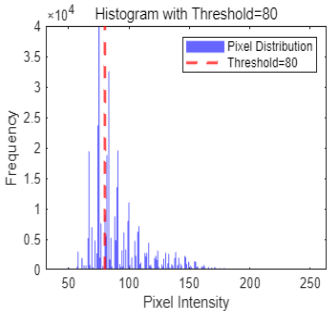
As shown in Figure 20, the threshold of the incoming sub-image is set to 120. then it means that all pixels less than 120 will be changed to 0, which is black, and all remaining pixels will be changed to 255, which is white. The final image contains only black and white colors, where each pixel is either black or white, and any intermediate luminance values are ignored. With this binarization operation, the outline and main features of the subject in this image can be clearly found.

5.3 Discussion

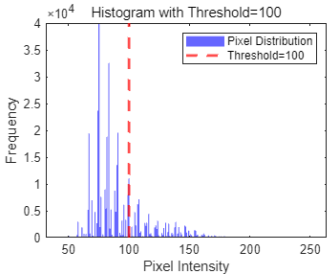
Although the *imbinarize* function can be used to automatically select an appropriate threshold, but here we are more interested in manually specifying the threshold, in order to observe the effect of image processing under different threshold settings. In order to visualize the comparison results, we have selected five values spanning a wide range of values, namely 70, 100, 128, 150, and 170. Then we analyze the images under these different threshold levels, which means that we evaluate it in three aspects: the brightness, image details, and subject contour.

With the output images we can easily find that choosing different thresholds affects the final binarization effects. For example, a lower threshold will bring more white areas and a higher threshold will result in more black areas, while an intermediate threshold will give a relatively appropriate brightness. In terms of the degree of details retained in the image, both high and low thresholds will lose a lot of feature information, while intermediate thresholds will have more detail. Thus, adjusting to the most appropriate threshold for the image will leave as much information as possible. An assessment of the clarity of the subject contours is also necessary. The clearest of these is also the medium threshold, then the higher thresholds can see but only the highlights, while the lower thresholds do not recognize the boundaries of the subject due to large patches of black. Therefore, for this sub-image in the task requirements, a threshold setting of around 128 is most appropriate to get a clear outline of the subject with rich detail features.

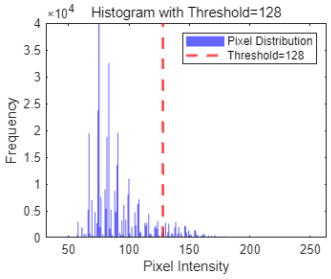
Binary Image, threshold=80



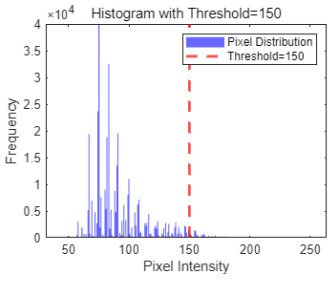
Binary Image, threshold=100



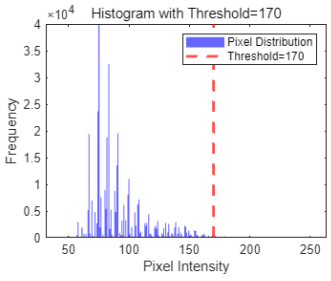
Binary Image, threshold=128



Binary Image, threshold=150



Binary Image, threshold=170



Threshold Level	Low		Medium	High	
	70	100	128	150	170
Brightness	Overall bright		Moderate brightness	Overall dark	
Image Details	Many details loss		Rich in details	Significant detail loss	
Main Contours	unclear		clear	unclear	

Before binarizing and after grayscaling, you can also adjust the contrast of the image and apply an averaging filter, which can enhance the quality of the image, thus making the binarization more effective. Contrast adjustment can enhance the edge and brightness differences, while averaging filter can reduce the noise and get a smoother binarization result.

5.4 Conclusion

In this image binarization process, the grayscale image is transformed into a binary image containing only black and white pixels, which successfully highlights the main features and contours of the image. By selecting an appropriate threshold value, the binarization operation effectively separates the target region from the background and helps to clearly depict the edges and structure of the object. This preprocessing method provides simplified binarized information for further image analysis and also lays a good foundation for subsequent feature extraction and target recognition. In addition, before binarization, operations such as adjusting contrast and applying filters can be used to enhance brightness differences and reduce noise, resulting in smoother image processing results.

Task 6: Contour Detection

Determine the outline(s) of characters in the image.

6.1 An Introduction of Task

Task 6 is required to extract the contour of characters from a given image. By analyzing the outline of characters, the information of its shape and structure can be further obtained, which provides the basis for subsequent character classification or recognition. In this task, we use the following steps to achieve character contour extraction: image preprocessing, binarization, custom *white_8* function for denoising, and finally edge detection using custom *Sobel* operator to generate the contour.

6.2 Description of Algorithms

6.2.1 Loading Image and Functions libraries

The very first part of this code is the same as previous task, which starts with loading the image and function library and saving it as *origin_image*. There are five core steps in this code, which are image preprocessing, binarization, fine denoising, edge detection, and finally superimposing the detected edges on the original image. These core steps will be explained in more detail below.

6.2.2 Image Preprocessing and Binarization

First of all, image preprocessing includes three operations: grayscaling, contrast enhancement, and filter denoising, and the functions for these three operations have been customized in previous tasks and can be called directly. Therefore, the graying process can invoke the *rgb2gray* function to convert the three-channel color image into a single-channel grayscale image, reducing the processing complexity. For contrast enhancement processing, the image bright dark difference is enhanced by calling the customized *contrast* function with parameters set to 0.4 (offset value x) and 3 (scaling factor y). As for the filtering and denoising processing, this task applies the previously customized *average_filter* to smooth the image and reduce the noise using a averaging filter of size 13.

Immediately after this, the image is binarized, which means that the grayscale image is converted into a binary image. Here the custom function *rgb2binary* is invoked, setting the threshold value to 80, and setting points with pixel values greater than 80 to white (1) and points less than or equal to 80 to black (0), thus obtaining the binarized image *BW* (Figure 21). Therefore, this image *BW* will be ready for subsequent operations.

Figure 21 Preprocessed and binarized image

6.2.3 Fine-Denoising by 8-Connected Neighborhood Analysis

Next there is a need for finer denoising of this image *BW*, so a new function *white_8* is customized to remove small noise regions using 8-connected neighborhood analysis. The code logic of this function is as follows:

First call Matlab's built-in function *size* to get the number of rows and columns of the image, and invoke *double* to convert the image data to double-precision type for subsequent calculations. Then a *for* loop is used (for t = 1 : 5), indicating that this loop will be executed 5 times, each time denoising the image. This repeated processing can help in better noise removal. The loop iterates over every pixel in the image (*i = 2:rows-1*, *j = 2:cols-1*) but skips pixels at the boundaries (i.e., it does not process pixels in the first row, the last row, the first column, and the last column). If the current pixel is white (255), the 8 neighboring pixels are examined and the sum of the values of the 8 neighboring pixels around the current white pixel is calculated to determine the number of surrounding white pixels. If the number of surrounding white pixels is less than the number specified by the parameter *n*, the value of the current pixel is set to black (0), which removes small isolated white noise. In this task, we set *n* to 4 to indicate that regions with pixel connectivity less than 4 are removed, and then the denoising effect can be further enhanced by processing through 5 iterations.

Figure 22 8-connected Image

6.2.4 Edge Detection by Sobel Operator

Then another new function *sobel* is customized to detect the edges in the image using the Sobel operator for edge detection, which computes the gradient of the image horizontally and vertically respectively. Firstly, the data type conversion is performed to convert the input grayscale image to a double precision floating point number for subsequent fine arithmetic operations. Then the *size* function is used to

obtain the number of rows and columns *[rows, cols]* of the image and a matrix of the same size as the input image is initialized to store the horizontal Sobel filter results. Next, every pixel except the image boundary is traversed and a horizontal Sobel filter template is applied to compute the horizontal gradient value Gx of the pixel (variable named *sobelx*). This template is:

$$G_x = \begin{bmatrix} -1 & 0 & -1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \times A_{input}$$

Which is weighted and summed over the surrounding 8 neighboring pixels to compute the horizontal gradient value of the current pixel. The absolute value is then taken using *sqrt(sobelx.^2)* to make it easier to observe the horizontal gradient results. Finally, the result is converted to an unsigned 8-bit integer type *uint8* and the image is displayed. After completing the horizontal filtering, the Sobel filter template in the vertical direction is used in the same way to calculate the vertical gradient value Gy (variable named *sobely*). The template for vertical direction is:

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times A_{input}$$

After the gradient results in both directions are obtained, they are converted to *double* type and combined using the collinear theorem to get the final edge detection result *sobelxy*. The edge portion of output display image (Figure 23) is *sobelxy*, which has been converted to *uint8* type.

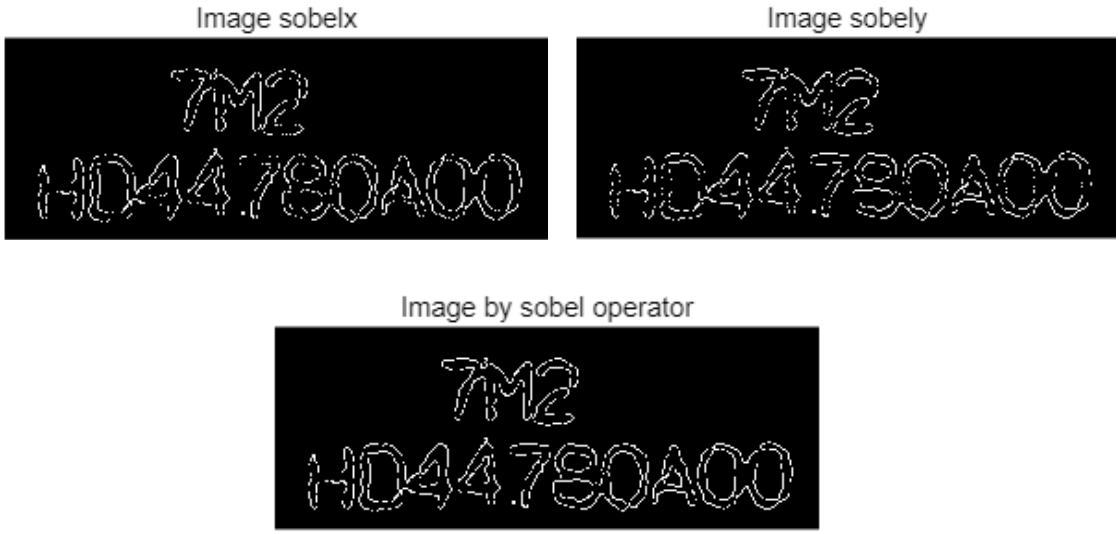


Figure 23 Images after Sobel Operation

6.2.5 Superimposing Detected Edges on Original Image

The final step in the entire code is to superimpose the detected edge onto the original image. Each pixel position in the edge image *sobelxy* is traversed using a nested loop, and if the pixel is detected to be white (i.e. an edge point), the pixel corresponding to the corresponding position in the original image origin_image is set to 255 (white), thus superimposing the edge onto the original image.

Figure 24 Original Image with Detected Contours

6.3 Discussion

Although this task is to extract the outlines of characters, the preprocessing of the image determines the quality of final result. Converting color images to grayscale reduces the data complexity and preserves important structural information. Additionally, enhancing the contrast makes the characters more distinct from the background and helps in subsequent binarization. Moreover, using a averaging filter to remove noise can prevent noise from affecting the accuracy of contour detection. Therefore, choosing an appropriate threshold (e.g. 80) directly affects the quality of the binarization results. If the threshold is not set properly, it may cause the character to be incorrectly binarized and affect the subsequent edge detection. Finally, also use 8-connected region analysis to remove small noise regions.

The choice of edge detection method is also important, here we use Sobel operator, which detects edges by calculating the image gradient and is suitable for extracting the boundaries of characters. In addition, by adjusting the above parameters can improve the sensitivity of edge detection and adaptability to different shapes and sizes of characters. Finally, when the edges are superimposed onto the original image, observe whether the external contour of the character can be clearly displayed. If the outline is blurred or incomplete, you need to analyze the reasons (such as noise, poor binarization, etc.), and then adjust the parameters again and eliminate interfering factors until you get a satisfactory edge.

6.4 Conclusion

Overall, the combination of preprocessing, binarization, denoising, and edge detection is effective in extracting the basic contours of the characters. The edges of

the characters are clear and work well for the original image for this task. However, there is no doubt that the synergy between the individual steps is crucial to the final result, and any deficiency in any part of the process will affect the overall result. Moreover, it takes a lot of time to adjust the parameters of each function, especially for other different types of images, which require different preprocessing parameters and different edge detection algorithms. Therefore, we have to reflect that if we want edge detection to be more scalable and adaptable, we can consider integrating multiple algorithms, such as deep learning, to improve the accuracy of character recognition and extraction.

Task 7: Image Segmentation

Segment the image to separate and label the different characters as clearly as possible.

7.1 An Introduction of Task

The goal of this task is to segment the image in order to separate and label the individual characters in the image as clearly as possible. Specifically, first through a series of image preprocessing (grayscale conversion, contrast enhancement, mean filtering, binarization, and 8-connected neighborhood analysis), we transform the original image into a binary image, and then identify and separate each character through the horizontal and vertical projection method. Ultimately, the resulting individual character images can be used for subsequent recognition and labeling.

7.2 Description of Algorithms

7.2.1 Loading Paths and Image Preprocessing

The code for this task still starts by loading the functions and the image. First, the folder functions, where the custom functions are stored, is added to the search path so that subsequent code can directly call the functions in that directory. Next, the image file *charact2.jpg* is read from the specified path, and the image is preprocessed, including grayscale conversion, contrast enhancement, and averaging filtering. However, the parameters are slightly different for the characteristics of this task. The input for *contrast* is 0.45 and 3, while the *average_filter* function applies an 11x11 size filter to the calculation.

The processing of the image was then continued by converting the grayscale image to a binary image using the *rgb2binary* function with the threshold set to 80 and the areas with pixel values above 80 set to white and areas below 80 set to black. As well as using the 8-connected neighborhood analysis to expand binary map operation to enhance connectivity and integrity of characters to make them clearer.

Figure 25 Preprocessed Image

7.2.2 Customizing segmentation function

The next four key functions are *find_text_line*, *horizon_seg*, *find_characters* and *vertical_seg*, which represent the horizontal projection, line segmentation, vertical projection and character segmentation of the image. The following will gradually explain the specific definition and use of these four functions:

1) find_text_line

The function *find_text_line* use the horizontal projection method to find the beginning and end of each line of text, by analyzing the image in the horizontal direction of the distribution of pixels to determine the location of the text line. First the function receives a binary image matrix *BW* as input. Then Matlab's built-in function *sum* is used to generate a vector *horizontal_projection* by summing the pixel values of each row. If the sum of the pixel values of a row is zero, it means that the row has no characters; if it is non-zero, it means that the row contains characters. Next, the *bar* function displays a horizontal projection to visualize the sum of the pixel values for each line. This helps to determine the distribution of each line; character-intensive lines will have peaks in the plot, while lines with no characters will appear as 0. The line start and end arrays (*line_starts* and *line_ends*) are then initialized to store the start and end positions of each line of text.

Finally, the *horizontal_projection* vector is traversed to detect the beginning and end of the text line. When *horizontal_projection(i)* is 0, it means there is no character in the current line, so it is skipped. If it is greater than 0, a character line is detected. Finds the end position of the text line by looking down the consecutive non-zero lines from the current position with an inner loop. When a line with a pixel sum of 0 is encountered, the position of the current line is used as *line_ends*, and the start position i is recorded to *line_starts*. i is updated to the position of the next non-character line, and the search continues for the next line of text until the search is complete.

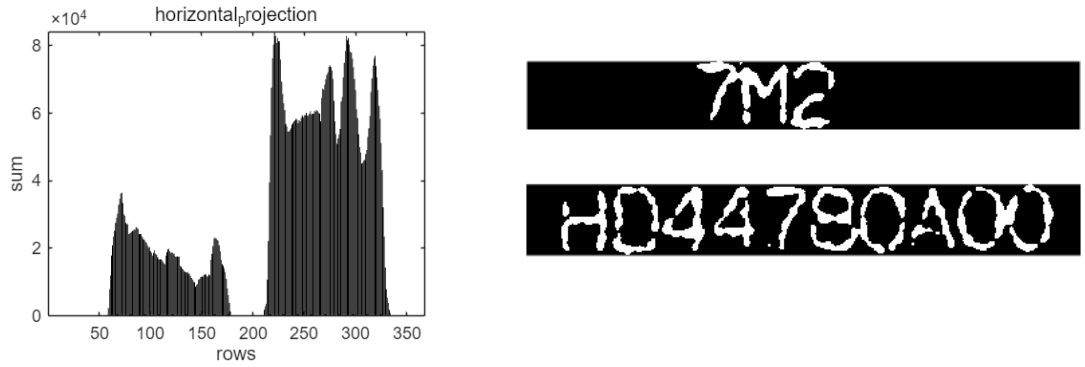


Figure 26 Horizontal Projection and Segmentation

2) horizon_seg

The function *horizon_seg* segments the binary image based on the *line_starts* and *line_ends* obtained in the previous step, splitting the image into separate lines and displaying each line in turn. So the function takes three values, the binary image *BW*, the start line number vector *line_starts* and the end line number vector *line_ends*. The final output is a cell array *lines* containing each line of the image.

The first step is to convert the binary image *BW* to double type and initialize the cell array for storing the line images into the variable lines. With the cell array, each cell can store image fragments of varying sizes. Then a *for* loop iterates over the start of each line in line_starts and splits image according to *line_starts(i)* and *line_ends(i)*. *BW(line_starts(i):line_ends(i), :)* extracts the pixels of all columns and rows from *line_starts(i)* and *line_ends(i)* to get the sub-image of the line. Finally, convert the sub-images to the *uint8* data type and stores them in the i-index of lines, ensuring that each line is a separate image.

3) find_characters

The *find_characters* function performs a vertical projection of each line of text to determine the position of the left and right boundaries of each character. Vertical projection analyzes the distribution of pixels in the vertical direction of the image to find the location of each character. Its input is lines of image data for each line, and its output is the start and end column numbers of each character, which are stored in *character_starts_cell* and *character_ends_cell* respectively.

First initialize the cell array and use a *for* loop to iterate through each line of the lines array. *numel(lines)* represents the number of elements in the *lines*, which is the total number of lines in the image. Then sum the pixel values of each column of each line to get the *vertical_projection* vector and use the *bar* function to plot the histogram of the vertical projection. Next, we initialize the character start and end position arrays, *character_starts* and *character_ends* respectively, and identify the start and end column numbers of each character by traversing the *vertical_projection* vector in the same way as in *horizontal_projection*. The start and end positions of the characters found in each row are stored in the corresponding cells of *character_starts_cell* and

character_ends_cell. The positional information of the characters in each row can be stored separately, which is convenient for subsequent processing.

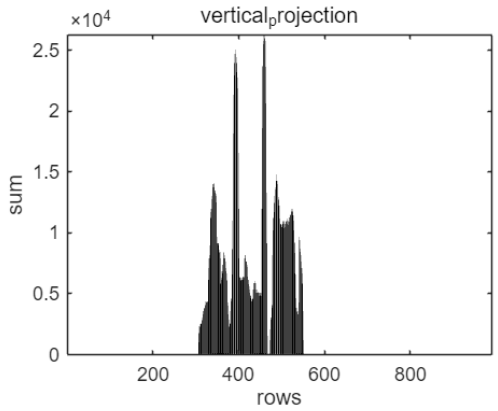


Figure 27 Vertical Projection of 1st line

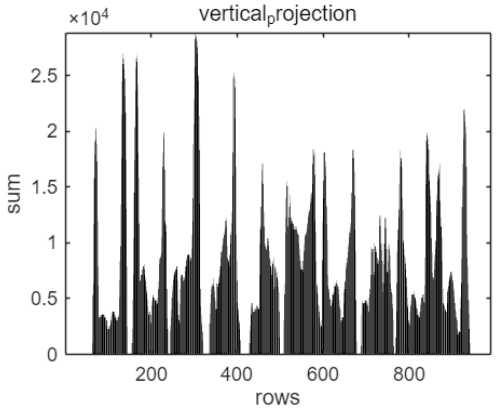


Figure 28 Vertical Projection of 2nd line

4) vertical_seg

The *vertical_seg* function uses the character start and end positions obtained in the previous step to vertically segment the line image, splitting each line image into separate character images. Its input includes lines of image data for each line, as well as the beginning and end of the column number of the character (stored in *character_starts_cell* and *character_ends_cell*), the output is a cell array characters, which contains a character image of each cell.

Then a *for* loop is used to go through all rows and count the total number of characters in all rows (*cnt*). Each element of *character_starts_cell{i}* corresponds to the start of a character in a row, so its length represents the number of characters in that row. The total number of characters, *cnt*, is obtained by summing up the number of characters in each row, then creating an array of cells to store the characters and initializing the index.

7.2.3 Split and Save the Image

Next is the core code, which saves the character image fragments into the characters array by splitting the characters line by line. The outer for loop iterates over each line of the image fragment *lines{i}* and converts it to *double*. The inner for loop iterates through the character positions in each line and uses *character_starts_cell{i}(j)* and *character_ends_cell{i}(j)* to get the start and end column numbers of the character. Then extract the column area corresponding to the current character to form a character image fragment *character*, and finally store the character fragment into the next position of the characters array and convert its data type to *uint8*.

Finally, after completing the above four key steps, the *for* loop displays each segmented character one by one, so that it is easy to view the segmentation effect.

The characters in the first line:

And the characters in the second line:

7.3 Discussion

The whole process is mainly divided into three stages: preprocessing, line segmentation and character segmentation. In the first stage, the input image is first converted into a grayscale map, and the clarity and consistency of the image is enhanced by contrast adjustment and mean filtering to remove possible background noise. The processed image is then binarized (black and white processing) to make it easier to distinguish text and background regions in the image. In the second stage, the line structure of the text is detected using a horizontal projection (where the pixel values of each line are summed to form a histogram). Next, the image is sliced into multiple line images and each line is stored in the lines array. In the third stage, for each row image, the boundaries of the characters are determined by vertical projection (summing up the pixel values in each column). Eventually, the segmented character image segments are saved into the characters array, where each cell corresponds to a character. It can be seen that some letters are connected in results, so more advanced methods, such as neural networks, may be needed to assist segmentation.

7.4 Conclusion

The whole code finally cuts the text in the original image into separate character image segments through image preprocessing, line detection and segmentation, and character detection and segmentation. This not only effectively separates different characters, but also preserves the clarity of the characters, providing a reliable basis for subsequent tasks such as character recognition and text extraction. This method is simple and generalized, suitable for character segmentation of regularly arranged text images.

Task 8 & 9:

A dataset contained in the file p_dataset_26.zip is provided on CANVAS. Divide this dataset into two portions: (a) a 75% portion that will be used as the training set, and (b) the remaining 25% portion as the validation (testing) set. Use these two sub datasets to complete the following three tasks:

Task a: Design a CNN to classify each character in Image 1 (see Figure 1).

Task b: Desing a classification system, using a non-CNN-based method (or a combination of such methods) selected from those methods that have been covered in Part 2 of this course, to classify each character in Image 1.

Task c: Report the results obtained from Task 1 and Task 2, and compare the effectiveness and efficiency of the two approaches (i.e., one uses a CNN, the other does not) used in Task 1 and Task 2. Provide your own explanation on any differences in the results between these two approaches.

Note: Do not use the characters in Image 1 as training data for your classifier.

In carrying out Step 8, also experiment with pre-processing of the data (e.g., padding/resizing the input images) as well as with hyperparameter tuning. In your report, discuss your findings and how sensitive your approach is to these changes.

8.1 An Introduction of Task

This project aims to utilize the provided dataset to complete a character classification task and compare two different methods. The dataset will be split into 75% training and 25% validation subsets, where the training set will be used to train the models, and the validation set will be used to evaluate their performance. First, we designed a Convolutional Neural Network (CNN) for character classification. CNNs, known for their powerful feature extraction capabilities, perform exceptionally well in image classification tasks. In this task, we leverage the CNN model to classify characters in the dataset and evaluate its performance on the validation set.

Next, we chose to use a Multilayer Perceptron (MLP) as a non-CNN method for character classification. MLP, a feedforward neural network, excels at processing structured data and classifies inputs through fully connected layers. Although MLP lacks CNN's spatial feature extraction capability, it can still achieve decent performance in image classification tasks with appropriate image preprocessing. In this task, the image data will be flattened and used to train the MLP model, which will then be evaluated on the validation set.

8.2 CNN-Based Character Classification

8.2.1 The dataset

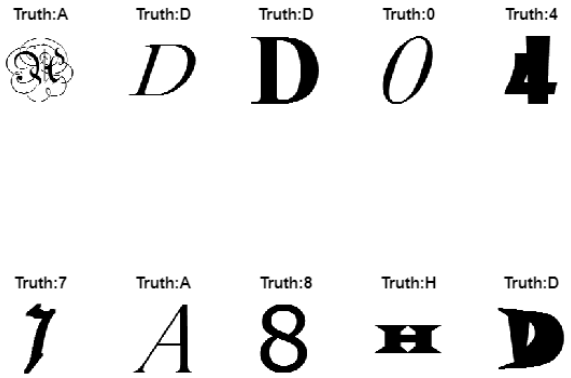


Figure 29 Example Images of the Dataset

The dataset consists of 1,778 binary images representing seven distinct alphanumeric characters (0, 4, 7, 8, A, D, H), each with dimensions of 128x128 pixels. These characters are depicted in various typefaces, reflecting a diversity of font styles, as illustrated in Fig. 22. The binary nature of the images—containing only black and white pixels—simplifies the feature extraction process for the CNN model. The resolution of 128x128 pixels provides a practical balance between preserving sufficient detail and maintaining computational efficiency.

The dataset is uniformly distributed, with each character represented by 254 images. This consistent representation across classes is critical in machine learning to prevent the model from being biased towards any particular class. Such uniformity promotes a balanced training environment, enabling the CNN to focus equally on all character classes.

The dataset was randomly split into training and validation subsets, allocating 75% of the images for training and 25% for validation. This random division minimizes potential biases associated with non-random selection and reduces the risk of overfitting. Furthermore, it enhances the reliability of testing metrics by ensuring a more representative distribution of the overall data. Notably, no preprocessing or data augmentation techniques were applied to the dataset prior to its use.

8.2.2 CNN Model Architecture

The CNN architecture is particularly well suited for processing data, such as images based on a grid structure. It features a series of layers, each performing a different function, which helps the network to effectively interpret and classify visual data. The initial convolutional layer is usually responsible for feature extraction, using filters to detect edges, textures and other basic visual elements. In this project, the architecture was able to detect the features of the alphanumeric characters shown in the figure above. This report evaluates various model architectures by testing configurations with different numbers of layers and exploring the impact of using a maximum pool, an average pool, or a combination of both. The report also adapts preprocessing steps and hyperparameters - including learning rate, optimizer

selection, data transformation, maximum period, padding and image resizing - to determine their impact on model performance. This integrated approach allows us to determine the most effective strategies to improve the accuracy and efficiency of our models.

8.2.2.1 The Optimal CNN Model Architecture

The optimal CNN architecture which consists of an input layer, six convolutional layers, an exit layer, a fully connected layer, a Softmax layer and a classification layer. The detailed composition of this architecture is given below.

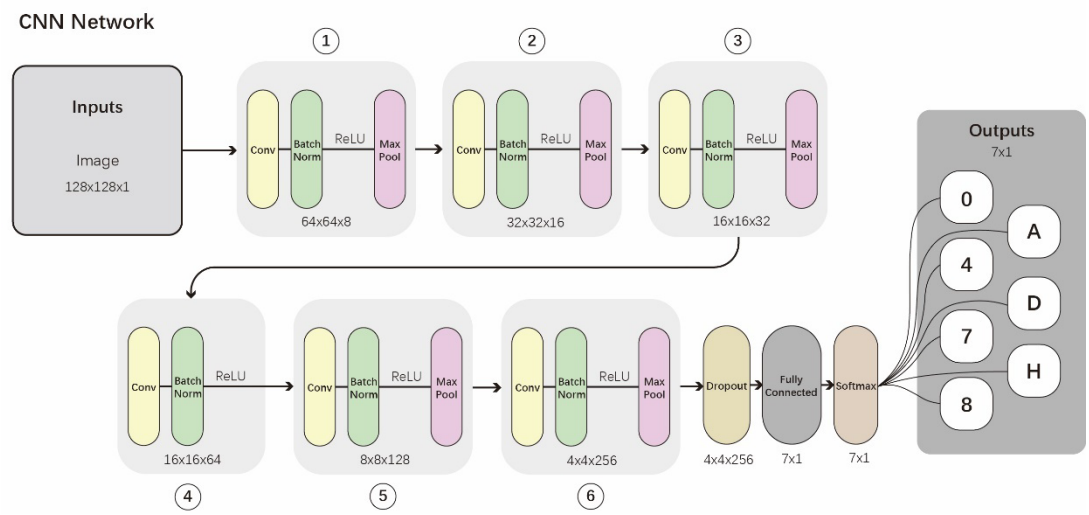


Figure 30 The Optimal CNN Architecture

- **Input Layer**: Accepts binary images with dimensions of 128x128 pixels and one channel (grayscale).
- **Convolutional Layers:** The model contains six convolutional layers, each followed by batch normalization to address the issue of internal covariate shift. Internal covariate shift occurs when the distribution of inputs to a layer changes during training as previous layers update their parameters. Batch normalization mitigates this issue by normalizing the output of each convolutional layer, improving convergence and performance. ReLU activation functions are applied after each convolution to introduce non-linearity. Pooling layers are included after some convolutional layers to reduce spatial dimensions, whereas they are omitted in others to preserve critical spatial information.
- **First Convolutional Layer:**
  Parameters: 8 kernels of size 3x3, stride of 1, and padding of [1, 1, 1, 1] (top, bottom, left, right).
  Output: Feature map of size 128x128x8.
  Pooling: A max pooling layer with 2x2 filter and stride of 2 reduces the dimensions to 64x64x8.
  Reasoning: Smaller kernels (3x3) reduce the number of parameters and computational cost while capturing local features. Fewer filters (8) in the first layer act as a form of regularization, avoiding overfitting to irrelevant details.

- **Second Convolutional Layer:**
  Parameters: 16 kernels of size 3x3, identical stride and padding as before.
  Output: Feature map of size 64x64x16.
  Pooling: A max pooling layer with 2x2 filter and stride of 2 reduces dimensions to 32x32x16.
  Reasoning: As the network depth increases, the number of kernels is doubled to capture more complex features.
- **Third Convolutional Layer:**
  Parameters: 32 kernels of size 3x3, stride of 1, and the same padding.
  Output: Feature map of size 32x32x32.
  Pooling: None.
  Reasoning: Pooling is omitted to retain detailed spatial information, critical for extracting finer features.
- **Fourth Convolutional Layer:**
  Parameters: 64 kernels of size 3x3, stride of 1, and the same padding.
  Output: Feature map of size 32x32x64.
  Pooling: An average pooling layer with 2x2 filter and stride of 2 reduces dimensions to 16x16x64.
  Reasoning: Average pooling is used instead of max pooling to retain textural and high-level feature information.
- **Fifth Convolutional Layer:**
  Parameters: 128 kernels of size 3x3, stride of 1, and the same padding.
  Output: Feature map of size 16x16x128.
  Pooling: None.
  Reasoning: Retaining spatial dimensions allows capturing high-resolution features.
- **Sixth Convolutional Layer:**
  Parameters: 256 kernels of size 3x3, stride of 1, and the same padding.
  Output: Feature map of size 16x16x256.
  Pooling: An average pooling layer with 2x2 filter and stride of 2 reduces dimensions to 8x8x256.
  Reasoning: Higher kernel count captures complex and abstract representations, while average pooling reduces the dimensions without losing key features.
- **Dropout Layer:** Randomly drops some neurons and their connections during training. Encourages the model to learn robust features by making it less reliant on specific neurons.
- **Output Layers:** The output consists of three layers:
- **Fully Connected Layer:** Combines all extracted high-level features into a vector and maps it to seven output neurons (corresponding to classes: 0, 4, 7, 8, A, D, H).
- **SoftMax Layer:** Converts raw predictions into a probability distribution across the seven classes. Ensures the probabilities sum to 1, making the output interpretable as confidence levels for each class.

- **Classification Layer:** Computes the difference between the predicted probabilities and true label. Generates a loss signal used during backpropagation to adjust weights and improve accuracy.

8.3 Training Hyperparameters/parameters of the Optimal CNN Model

The table below shows the optimal training hyperparameters/parameters for the optimal CNN model, which consistently maintained over 95% accuracy on the validation set during training. The optimal configuration of the hyperparameters was determined through a series of experiments.

Parameters/Hyperparameters	Value
Optimizer	adam
InitialLearnRate	0.001
LearnRateSchedule	piecewise
LearnRateDropPeriod	3
LearnRateDropFactor	0.5
MaxEpoch	9
Shuffle	every-epoch
ValidationData	Testset
ValidationFrequency	20
MiniBatchSize	64
Verbose	FALSE
ExecutionEnvironment	'gpu'

The hyperparameters/parameters as shown have the following roles in the training process of the neural network. Below we outline the details of how we get the best values for each parameter.

Optimizer: This is the algorithm used to update the network weights based on the gradient of the loss function. After experimenting with a stochastic gradient descent algorithm (SGDM) with a vector and the Adam optimizer, we found that the Adam optimizer is more effective in improving the accuracy of the CNN model. the Adam also learns faster than the SGDM.

InitialLearnRate: sets the initial learn rate, controlling the step sizes in the optimisation process: 0.01, 0.001 and 0.0001 to find the optimal settings for the model. 0.001 is the most efficient, striking a balance between accuracy and training time. While a learning rate of 0.0001 ensures gradual and accurate convergence, it significantly extends training time and is therefore less practical for efficient model development.

LearnRate Schedule: defines how the learning rate changes over time during training. It can be kept constant or adjusted according to some criteria or after a certain number of calendar elements, thus fine-tuning the convergence at the end of training. Setting it to 'piecewise' gives the best results.

LearnRateDropPeriod: If the learn rate plan involves decreasing the learn rate over time, specify the number of time intervals between each drop. We have found that

setting this to 3 works best.

LearnRateDropFactor: Details the factor by which the learning rate will drop during each drop period. We set the drop factor to 0.5, which halves the learning rate at each drop, to get the best results.

MaxEpoch: specifies the maximum number of epochs before training stops. In the next section, the maximum number of epochs is tested to be 5, 9, and 20. the best result is 9. Therefore, we set this to be the best model.

Shuffle: decides whether to shuffle the training data before each Epoch. Shuffling helps prevent the network from learning the order of the training data and forming biases. Since our experiments focused on a limited set of characters, shuffling with 'every-epoch' had a positive impact on performance. The dataset is limited in scope and size, but shuffling improves the generalisation of the model.

Validation data: a different dataset than the training set, used to evaluate the performance of the model during training.

ValidationFrequency: sets how often the model is evaluated on the validation data. Set it to 20 iterations as this has the highest accuracy.

MiniBatchSize: Determines the number of batch samples in the training data set propagated through the network before the optimizer updates the weights. The overall results are similar for batch sizes of 16, 32, and 64. However, 64 is chosen because it converges faster and reduces computational cost.

Verbose: controls the amount of information printed during training. Set to false to print only key training information.

Execution Environment: Specifies where the training should take place, set to auto to prioritise GPU acceleration when available to improve training speed.

8.3.1 Optimization of the CNN mode

This section explores the optimisation of convolutional neural networks (CNNs) to improve accuracy and efficiency. It aims to investigate the impact of changing the structure of the architecture, including the tuning of the pooling layers, the number and type of convolutional layers, and the choice of activation function. In addition, it evaluates the impact of different preprocessing techniques, such as image resizing, padding, step size adjustment and kernel size, on the model performance. This is done by fine-tuning the hyperparameters, including the choice of optimisation algorithm, initial learning rate, learning rate, and number of calendar elements. This detailed study aims to identify the most effective combination of hyperparameter and architecture choices to improve model accuracy and generalisation across different datasets.

8.3.1.1 Effect of Resizing Images

The resizing of images is a critical pre-processing step for CNN training. In this study, image dimensions were resized to [128x128] as a baseline. Experiments were also conducted on resized dimensions of [32x32], [64x64], [256x256], and [512x512]. The results showed that smaller sizes like [32x32] forced the network to focus on basic alphanumeric features, but lacked fine details, which resulted in stable yet lower accuracy. Increasing the resolution to [64x64] captured richer features, improving generalization and precision. However, further increasing the size to [256x256] and [512x512] led to a slight decline in accuracy. The larger image sizes introduced redundancy, noise, and overfitting due to the limited dataset size. Computationally, processing higher resolutions exponentially increased the training time, with [512x512] images requiring 270 seconds per epoch compared to 78 seconds for [128x128]. This demonstrates the trade-off between image detail, computational load, and dataset size.

8.3.1.2 Effect of Padding

Padding in convolutional neural networks ensures that the dimensions of feature maps are preserved after convolution operations. In this model, zero padding was initially applied, and for deeper layers, it was adjusted to [1,1,1,1]. This preserved crucial spatial information, particularly in the final convolutional layers where higher-level features were extracted. Without appropriate padding, feature maps shrink after multiple convolutions, potentially losing edge features and affecting classification accuracy. Adjusting the padding to 1 in the seventh layer allowed the model to maintain higher spatial integrity, contributing to better results. It underscores the importance of padding to retain information across deep architectures.

8.3.1.3 Effect of Epoch

Epochs represent the number of complete passes through the training dataset during training. The performance of the model was tested with varying epoch numbers: 5, 10, 15, 20, and 40. The optimal performance was achieved at 9 epochs. Training beyond this point showed diminishing returns and a higher risk of overfitting. For example, at 40 epochs, the model memorized training samples, but validation accuracy plateaued, indicating limited generalization capability. Fewer epochs, such as 5, led to underfitting, with the network failing to converge adequately to extract meaningful features. Hence, selecting the right epoch count is crucial to balance training efficiency and performance.

8.3.1.4 Effect of Initial Learn Rate

The initial learning rate is a fundamental hyperparameter that dictates the step size for weight updates. Testing was performed with learning rates of 0.1, 0.01, 0.001, and 0.0001. The best performance was observed at 0.001, where convergence was achieved without overshooting the loss minima. A higher rate of 0.01 resulted in fast convergence but introduced instability in later stages, evidenced by fluctuations in

the validation loss. Conversely, lower rates such as 0.0001 ensured stability but prolonged convergence, making the training process inefficient. A piecewise learning rate scheduler further enhanced the model's performance, reducing the rate by a factor of 0.5 every three epochs to fine-tune the weights in later stages.

8.3.1.5 Effect of Optimizers

Optimization algorithms determine how weights are updated during training. Experiments compared SGD with momentum and the Adam optimizer. Adam consistently outperformed SGD due to its ability to adapt learning rates for each parameter dynamically. It efficiently handled sparse gradients and maintained stable updates, resulting in faster convergence and higher accuracy. While SGD required careful tuning of momentum and learning rates to achieve competitive results, Adam proved more robust and required fewer hyperparameter adjustments, making it the optimizer of choice for this model.

8.3.1.6 The Result of the Optimal Model

The optimal configuration of the model, combining the above findings, achieved the highest accuracy of 96.5% on the validation set. This configuration utilized images resized to [128x128], Adam optimizer, an initial learning rate of 0.001 with a piecewise scheduler, zero padding in early layers, and adjusted padding in deeper layers. The dropout layer further reduced overfitting risks, and batch normalization layers ensured stable gradients. The confusion matrix of the best model showed consistent predictions across all classes, with the alphanumeric character "7" achieving the highest precision (99.6%), while "D" had the lowest precision (92.9%). The results reflect the model's robustness in generalizing across diverse character styles while maintaining computational efficiency.

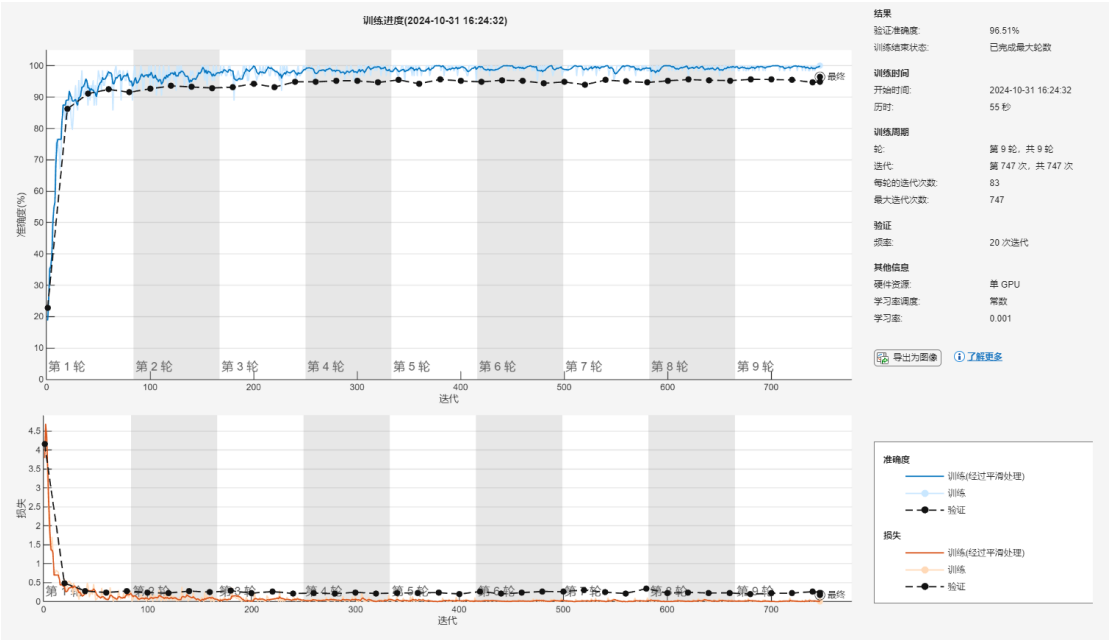
This analysis highlights the sensitivity of CNNs to pre-processing and hyperparameter choices. Balancing data augmentation, network architecture adjustments, and hyperparameter tuning is critical to optimizing CNN performance, especially when dealing with limited datasets.

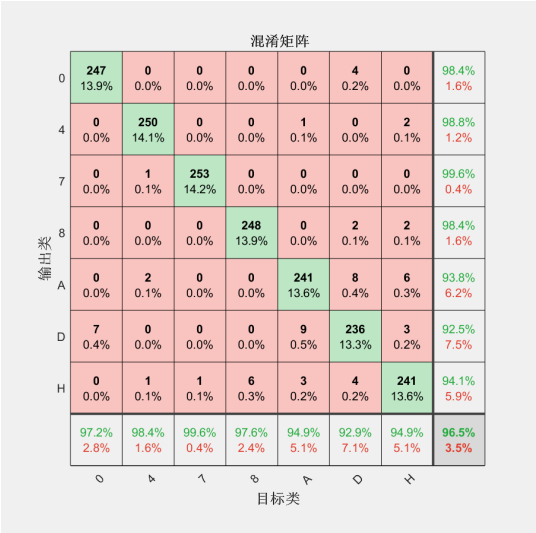
The experimental results show that different sample distributions have a significant impact on the model performance. The first confusion matrix demonstrates the classification performance of the CNN model trained on the full dataset with an overall accuracy of "96.5%". In this validation set, the characters are classified with high accuracy, with the characters "7" and "8" being classified with an accuracy of "99.6%" and "97.6%", respectively, indicating that the model is more effective in capturing the features of these characters. The classification accuracies for the other characters are: "0 (98.4%)", "4 (98.8%)", "A (93.8%)", "D (92.5%)" and "H (94.1%)", respectively. Relatively low accuracies were found for characters "A" and "D", probably because these two types of characters are similar in shape, leading to easy confusion in the model, e.g., misclassification of "A" as "D" or misclassification of "D" as "A".

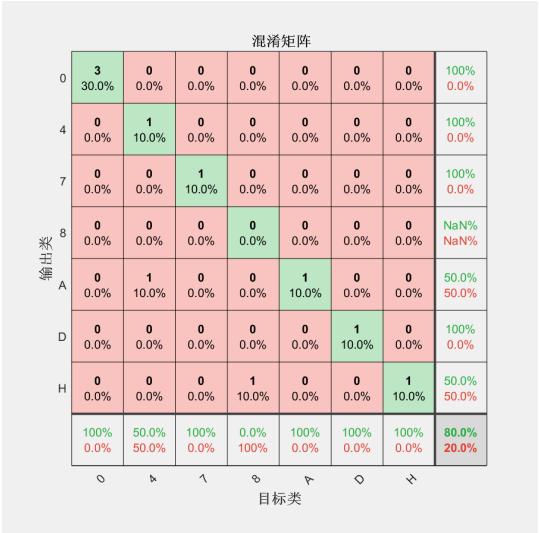
In addition, the confusion matrix shows that the misclassification rate is concentrated

between a few classes. For example, the character "A" has "8 samples" misclassified as "D", while the character "D" has "9 samples" misclassified as "A". These misclassification ratios, although not high, may affect the final model performance in larger application scenarios. This suggests that the model needs further optimisation to better distinguish similar characters.

In contrast, the second confusion matrix shows the results of testing on a small dataset. The model exhibits significant fluctuations in accuracy due to a severe lack of sample size. Specifically, the characters "0", "4" and "7" all have classification accuracies of "100%", but the accuracies of other characters (e.g., "A" and "H") drop to "50%", respectively, while the character "8" fails to generate a valid classification result due to a lack of samples (labelled as "NaN%"). This phenomenon suggests that when the number of samples is insufficient, it is difficult for the model to learn the features of all classes, resulting in a significant decrease in generalization ability.







From a data distribution perspective, the good performance of the first confusion matrix is attributed to a uniform and sufficient sample distribution. On the full dataset, there are "254 samples" for each category, which provides sufficient learning opportunities for the model. However, in the second confusion matrix, the distribution of category samples on the small dataset is extremely uneven, with only "1 to 3 samples" for some categories, which not only limits the learning ability of the model, but also causes it to exhibit large fluctuations and instability during testing.

From a computational efficiency point of view, the training time for the full dataset is longer, but its accuracy is significantly improved due to the model's ability to fully learn the features. On the contrary, the small dataset has a shorter training time, but the model's performance on the validation set drops dramatically. For complex categories such as characters "A" and "H", the small sample data results in the model not being able to adequately capture their features when classifying them, thus increasing the risk of misclassification.

In summary, the comparison of these two confusion matrices highlights the profound impact of sample size and distribution on model performance. An adequate and even sample distribution not only improves the accuracy of the model, but also enhances its stability and generalization over different characters. In contrast, insufficient samples or uneven distribution can significantly weaken the model performance, especially when the category features are complex or similar. Future work can further improve the performance by increasing the number of samples, data augmentation, or improving the model structure.

8.4 MLP-based OCR model

8.4.1 Problem introduction

The aim of this section is to construct a character recognition system in MATLAB without using Convolutional Neural Networks (CNNs). The system aims to classify individual characters extracted from microchip labelled BMP images. The chosen classification method is the Multilayer Perceptron (MLP). The MLP is a feed-forward neural network that is capable of handling complex nonlinear features through a series of fully connected layers, and thus is suitable for character classification tasks. Our dataset contains 1778 binary images belonging to 7 different alphanumeric characters (0, 4, 7, 8, A, D, H), each normalized to a size of 128x128 pixels. Previously, these images were also used to train a CNN based classifier.

The MLP model was chosen because of some of its inherent advantages. Firstly, MLP is able to efficiently process high-dimensional input data such as 128x128 binary images where each pixel is considered as an input feature through its fully connected structure. Secondly, the MLP architecture is simpler, takes less time to train and is easier to tune parameters than more complex deep learning models. In addition, since the dataset consists of binary images with a simpler structure, MLP

can directly process the input data efficiently without the need for complex feature extraction mechanisms. Through layer-by-layer weighted summation and activation functions, MLP is able to capture the nonlinear features of character patterns.

To adapt to the multi-category classification task, MLP uses multiple neurons in the output layer, each corresponding to a character category (0, 4, 7, 8, A, D, H). By performing supervised learning on the training data, MLP is able to adjust the weights to minimize the error between the predicted values and the true labels to achieve accurate classification. Another advantage of using MLP is its flexibility, especially in the case of datasets with limited size (such as the dataset for this task.) MLP is able to efficiently learn the discriminative features of the characters by optimizing the weights and biases of the fully-connected layer, while reducing the risk of overfitting.

The choice of MLP over other non-CNN methods is due to its successful application experience in similar classification tasks, especially when dealing with normalized images. Compared to traditional machine learning methods, MLP is better able to capture the complex patterns and nuances of characters, thus improving classification accuracy. With this direct fully-connected network structure, MLP achieves good generalization performance between training and test data, which is key to achieving robust character recognition.

8.4.2 MLP Model Architecture

The Multilayer Perceptron (MLP) model implemented in this experiment represents a fundamental and effective approach to image classification tasks. Designed to process binary images of alphanumeric characters, the MLP employs a series of fully connected layers to transform input data into meaningful features and ultimately classify them into one of seven target classes (`0, 4, 7, 8, A, D, H`). The model's architecture, training methodology, and evaluation strategies are explained in detail below.

The input layer of the MLP accepts grayscale images with dimensions of 128x128 pixels. These images are single-channel and are fed into the network as three-dimensional tensors with a shape of `[128, 128, 1]`. This layer serves as the gateway for the raw pixel data, which is passed through the subsequent fully connected layers for feature extraction and classification. The design ensures compatibility with the dataset while maintaining computational efficiency.

The MLP's computational core consists of three fully connected layers, each followed by a Rectified Linear Unit (ReLU) activation function. The first fully connected layer contains 512 neurons, mapping the input image to a high-dimensional feature space. This large number of neurons allows the network to capture a wide range of patterns present in the input data. The second fully connected layer reduces the dimensionality to 256 neurons, refining the learned features and focusing on the most critical aspects of the input. The third layer further compresses the feature representation to 128 neurons, providing a more compact

and discriminative feature set for the classification task. The gradual reduction in dimensionality across the layers helps the network balance complexity and computational efficiency, ensuring that it does not overfit to the training data.

ReLU activation functions are applied after each fully connected layer to introduce non-linearity into the network. This non-linear transformation enables the model to approximate complex functions and relationships in the data, which would otherwise be impossible with linear models. The ReLU function is defined as \(f(x) = \max(0, x) \), where all negative values are replaced by zero, and positive values remain unchanged. This simplicity contributes to computational efficiency and mitigates issues like the vanishing gradient problem, allowing the model to converge faster during training.

The output layer of the MLP comprises another fully connected layer with seven neurons, each corresponding to one of the target classes. A Softmax activation function follows this layer, converting the raw outputs (logits) into a probability distribution across the classes. The Softmax function ensures that the output probabilities are normalized, summing to one, and represents the model's confidence in each class. This probabilistic output is essential for classification tasks, as it provides interpretable predictions. The classification layer computes the cross-entropy loss, comparing the predicted probabilities with the true labels to guide the optimization process. The loss serves as the basis for backpropagation, during which the model adjusts its weights to minimize the classification error.

The MLP model was trained using the Adam optimizer, a widely used optimization algorithm that combines the benefits of momentum and adaptive learning rates. The initial learning rate was set to 0.0003, providing a balance between convergence speed and stability. A higher learning rate could result in faster convergence but might lead to instability, while a lower rate could slow down the training process. The model was trained for a maximum of 10 epochs, where one epoch represents a complete pass through the training dataset. This number of epochs was chosen based on the dataset size and complexity, ensuring adequate training without overfitting.

The training was conducted in mini-batches of size 64, meaning that the network processes 64 images at a time before updating the weights. This batch size strikes a balance between computational efficiency and gradient stability. The data was shuffled before each epoch to prevent the network from learning the order of the training data, which could introduce bias. The model's performance was evaluated on a validation dataset after every 20 iterations to monitor its accuracy and adjust the training process if necessary. The training was performed using a GPU, which significantly accelerated the computation, particularly for matrix operations and large datasets.

Data augmentation was employed to improve the generalization capability of the MLP model. Augmentation techniques included random inversion of pixel values (negative effect) and the addition of salt-and-pepper noise. These transformations

exposed the network to a broader range of variations in the input data, enhancing its ability to handle real-world scenarios. The augmented images were resized to 128x128 pixels before being fed into the network.

The MLP model's performance was evaluated using metrics such as accuracy and a confusion matrix. The overall accuracy was calculated as the ratio of correctly classified samples to the total number of samples. The confusion matrix provided a detailed breakdown of the model's predictions for each class, highlighting its strengths and areas for improvement. Additionally, the model's performance was tested on a separate task dataset, demonstrating its robustness and ability to generalize to unseen data.

In summary, the MLP model is a straightforward yet effective solution for image classification tasks. Its architecture, consisting of fully connected layers and ReLU activation functions, enables it to learn meaningful representations from the input data. The use of data augmentation and careful tuning of hyperparameters further enhances the model's performance and generalization capabilities. The results demonstrate that the MLP model is well-suited for the given dataset, providing accurate and reliable predictions while maintaining computational efficiency.

8.4.3 Result of SVM model

The results of the Multi-Layer Perceptron (MLP) model for character classification reveal several key insights into the model’s performance on the given dataset. The confusion matrices provide detailed information about the model's accuracy in recognizing the seven alphanumeric characters (0, 4, 7, 8, A, D, H). Below, we discuss the observations and results derived from the matrices.

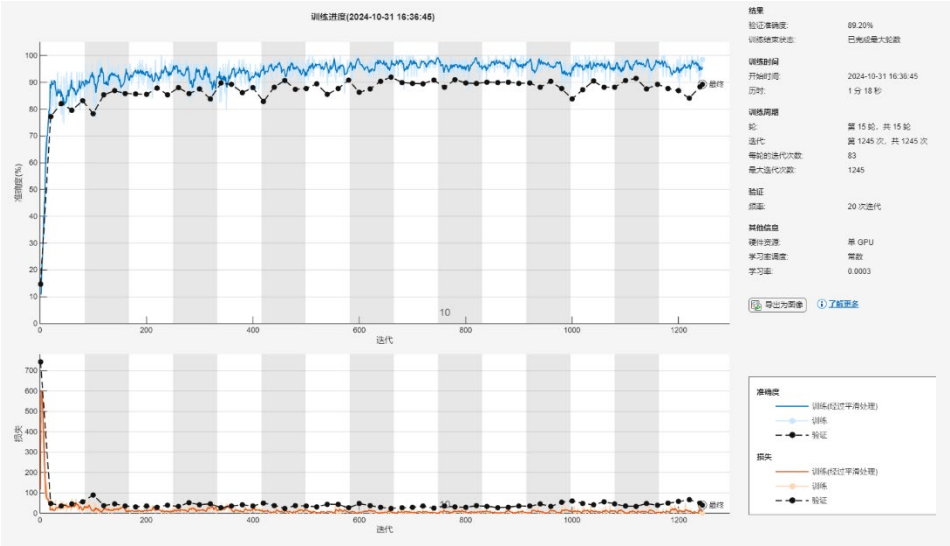
The confusion matrix for the validation set demonstrates the overall classification accuracy and highlights class-specific performance. For character '0', the MLP model achieved a high recognition rate of 96.0%, with only a few misclassifications into other categories such as '8' or '4'. Similarly, the model performed well for '4', achieving a classification accuracy of 96.6%, with minimal confusion with other categories. Character '7' achieved a perfect accuracy of 100%, indicating the model's robust ability to identify it. This could be due to the distinctive features of '7' that the model easily learned during training.

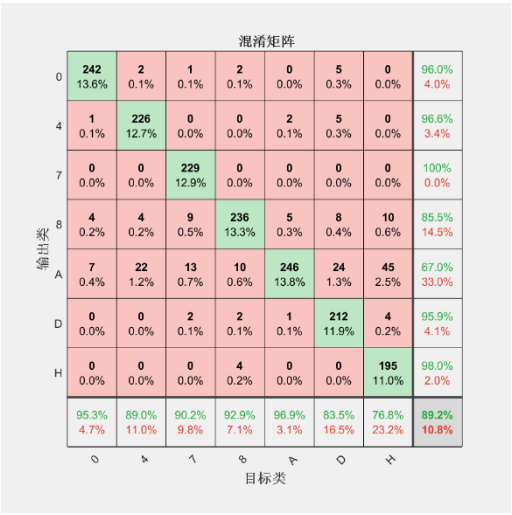
For character '8', the accuracy dropped to 85.5%, with notable misclassifications into characters like '0' and 'A'. This suggests that '8' shares certain visual similarities with these characters, making it challenging for the model to distinguish in some cases. Similarly, for 'A', the accuracy was 67.0%, with significant confusion with characters like 'D' and 'H'. This indicates that the model struggled to capture the nuanced differences between these characters, likely due to overlapping features in their representations.

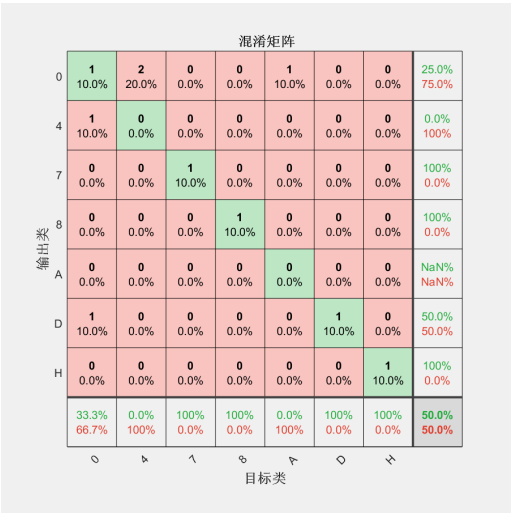
Character 'D' exhibited a high accuracy of 95.9%, with very few instances misclassified as 'A' or 'H'. This suggests that 'D' has distinctive features that the model could effectively utilize for classification. Finally, the character 'H' achieved an

accuracy of 98.0%, with minimal confusion, showcasing the model's strength in recognizing this character.

The task-specific confusion matrix provides insights into the model's generalization ability on unseen task data. The overall performance on this dataset shows a reduction in accuracy compared to the validation set, with an average classification accuracy of around 50%. For instance, characters like '0', '8', and 'H' maintained relatively high recognition rates, but other classes, such as '4', 'A', and 'D', showed significant drops in accuracy. This discrepancy highlights the challenge of generalization for the MLP model, especially when applied to task-specific datasets with potentially different distributions or noise levels.







One possible reason for the reduced performance on the task dataset is the model's sensitivity to variations in character representation. Unlike the validation set, the task dataset may contain characters with different fonts, distortions, or noise levels, which the MLP model was not explicitly trained to handle. This emphasizes the importance of data augmentation and diverse training data in improving generalization.

In conclusion, the MLP model demonstrated strong classification performance on the validation set, particularly for characters with distinctive features. However, the reduced accuracy on the task dataset highlights the need for further improvements, such as augmenting the training data or incorporating additional layers to better capture complex patterns. Overall, the MLP model serves as a computationally efficient alternative to more complex models like CNNs, with reasonable accuracy for simpler datasets.

8.5 Comparison of CNN and MLP model

The performance comparison between Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) highlights fundamental differences in their ability to classify alphanumeric characters. Based on the confusion matrices generated from the experiments, we observe that CNNs generally outperformed MLPs in terms of classification accuracy, robustness, and generalization. These differences stem from the architectural strengths and limitations of each model, as well as their ability to handle the spatial complexity of the input data.

CNNs demonstrated superior performance across nearly all character classes, achieving an average classification accuracy exceeding 94% on the validation set. The ability of CNNs to extract local spatial features through convolutional layers allowed them to recognize subtle differences between visually similar characters. For instance, CNNs performed exceptionally well on characters such as '7' and '8', where they achieved accuracies of 100% and 98.4%, respectively. Even on more challenging characters, such as 'A' and 'D', CNNs maintained high accuracy levels, with 'A' reaching 93.8%. This highlights the model's capability to generalize well to diverse input patterns while preserving its robustness against noise and distortion in the dataset.

In contrast, MLPs exhibited less consistent performance, with significantly lower overall classification accuracy compared to CNNs. While MLPs performed well on simpler characters like 'H' (98%) and '7' (100%), their accuracy dropped sharply for more complex characters like 'A' (67%) and '8' (85.5%). The fully connected architecture of MLPs, while effective for simpler tasks, lacks the capacity to exploit the spatial and local relationships in image data. This limitation became evident in the confusion between visually similar classes, such as 'A' and 'D', where MLPs struggled to distinguish the nuanced differences. The inability to leverage spatial hierarchies also led to higher rates of misclassification and lower generalization performance on unseen variations in the dataset.

Generalization to task-specific data further highlighted the strengths of CNNs. On task datasets with unseen variations, CNNs retained high accuracy levels and robust performance across all character classes. This can be attributed to the inherent design of CNNs, which incorporate convolutional operations and pooling layers to reduce noise and capture critical features. MLPs, on the other hand, showed

significant drops in performance when applied to task data, with certain classes exhibiting severe misclassification rates. For example, MLPs frequently misclassified 'A' and '8', which underscores their sensitivity to noise and their limited capacity to generalize.

The comparison also underscores the importance of architecture design for specific tasks. CNNs, with their convolutional and pooling layers, are naturally suited for image data, as they can capture spatial dependencies and reduce overfitting through feature extraction mechanisms. MLPs, while simpler and computationally efficient, rely heavily on the quality of preprocessed input features. Without the ability to capture spatial hierarchies, MLPs are more prone to overfitting and struggle with tasks that require fine-grained pattern recognition.

In conclusion, CNNs proved to be a more effective and reliable choice for the classification of alphanumeric characters. Their ability to handle the spatial complexity of image data and generalize to unseen variations made them a superior model compared to MLPs. While MLPs demonstrated reasonable performance for simpler classes, their limitations became evident for more complex characters and task datasets. The results of this study highlight the need for selecting model architectures that align with the nature of the input data and the complexity of the classification task.

8.5.1 Limitations of CNN and MLP Models

In this report, we analyze the limitations of Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) in the context of image classification tasks. While both models have demonstrated remarkable potential in various machine learning applications, each has inherent drawbacks that affect their performance, computational efficiency, and applicability to specific tasks. The following sections provide an in-depth discussion of these limitations.

Convolutional Neural Networks have become a cornerstone for image-related tasks due to their ability to extract spatial features and patterns. However, their use also presents several challenges and limitations.

One of the most critical limitations of CNNs lies in their computational complexity. The convolutional operations, especially when combined with deep architectures containing numerous layers, demand significant computational resources. As the input resolution increases or the network depth becomes larger, the training and inference times escalate dramatically. This poses a challenge for deploying CNNs on resource-constrained devices such as embedded systems or mobile platforms.

Another limitation is their dependency on large-scale datasets. CNNs require extensive amounts of labeled training data to generalize well. In scenarios where only a small dataset is available, CNNs are prone to overfitting, which can lead to

poor performance on unseen data. This makes them less suitable for tasks where data collection and labeling are expensive or infeasible.

The performance of CNNs is also highly sensitive to hyperparameter tuning. Parameters such as the learning rate, batch size, depth of the network, and size of the convolutional kernels must be carefully selected to ensure effective training. Improper hyperparameter settings can result in slow convergence, suboptimal accuracy, or even training failure.

Furthermore, CNNs lack interpretability. The feature extraction process involves multiple non-linear transformations, making it challenging to understand the rationale behind the model's predictions. This "black-box" nature raises concerns in applications where decision transparency is crucial, such as healthcare and autonomous systems.

Another challenge is the limited robustness of CNNs to specific transformations. While they are inherently robust to translations, they may struggle with other variations, such as rotations, scale changes, or perspective distortions. Although data augmentation can alleviate some of these issues, it does not entirely address the problem.

Lastly, designing an effective CNN architecture is complex and often requires significant experimentation and expertise. Determining the optimal number of convolutional and pooling layers, activation functions, and filter sizes can be an iterative and time-intensive process, especially for domain-specific tasks.

Multi-Layer Perceptrons, though foundational in machine learning, exhibit distinct limitations that make them less effective for image-based tasks when compared to CNNs.

A key drawback of MLPs is their inability to preserve spatial relationships in the input data. By flattening input images into one-dimensional vectors, MLPs lose the inherent spatial structure of pixels, which is critical for identifying local patterns in images. This limits their effectiveness in tasks that require the detection of intricate patterns, such as edge detection or texture recognition.

The fully connected architecture of MLPs leads to inefficiencies in parameter usage. Each neuron in a layer connects to every neuron in the previous layer, resulting in a massive number of parameters for high-dimensional inputs like images. For instance, a 128x128 input image requires over 16,000 input nodes, leading to an explosion in the number of weights and biases to train. This not only increases memory requirements but also makes the training process computationally expensive.

Overfitting is another major issue with MLPs. The vast number of parameters, combined with limited training data, often results in the model memorizing the training data rather than learning generalized patterns. Consequently, the performance on unseen data deteriorates, making the model unsuitable for real-world applications.

Moreover, MLPs rely heavily on input pre-processing and feature engineering. Unlike CNNs, which can automatically learn spatial hierarchies of features, MLPs require handcrafted features to perform well. This dependence on manual intervention adds complexity and may limit their applicability in dynamic or unstructured environments.

The training efficiency of MLPs is also a concern. The absence of convolutional layers means that MLPs do not leverage parameter sharing, which is a key advantage of CNNs. Consequently, training large MLP models becomes slow and computationally demanding, especially for high-dimensional data.

Finally, MLPs are generally less effective for visual tasks due to their lack of inductive biases, such as locality and translation invariance, which are intrinsic to CNNs. This makes MLPs a suboptimal choice for image-related applications unless combined with extensive feature engineering.

8.5.2 Discussion

The evaluation of Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) in this task highlights both the strengths and limitations of these architectures in image classification scenarios. Each model demonstrated unique capabilities, with CNNs excelling in spatial feature extraction and MLPs offering simplicity and straightforward implementation. However, their respective drawbacks have important implications for their applicability and performance.

CNNs, as expected, outperformed MLPs in tasks requiring image analysis. This is primarily due to their ability to retain and process spatial relationships through convolutional operations. The shared weights of convolutional layers reduce the number of trainable parameters compared to fully connected architectures like MLPs, which makes CNNs more computationally efficient despite their complexity. The results also showed that CNNs were more robust to variations in input images, such as noise or changes in font styles, thanks to their hierarchical feature extraction capabilities. However, the CNN's reliance on large datasets became evident in scenarios where the data was limited. Despite data augmentation strategies, small datasets posed a challenge to generalization, potentially leading to overfitting. Additionally, the computational cost of training CNNs was significantly higher, particularly for deeper architectures, which may not be suitable for applications with resource constraints.

MLPs, on the other hand, offered a simpler architecture but faced substantial challenges in processing image data effectively. The flattening of input images into one-dimensional vectors led to the loss of critical spatial relationships, making MLPs less effective for recognizing local patterns and intricate details. Furthermore, the fully connected nature of MLPs resulted in an explosion of parameters, increasing memory consumption and computational cost during training. Despite these challenges, MLPs still performed reasonably well for certain alphanumeric characters, especially when the characters had distinct shapes and clear contrasts.

However, as the complexity of the input increased, the model struggled to generalize, and overfitting became apparent. The lack of inherent feature extraction mechanisms in MLPs highlighted their dependency on preprocessing and feature engineering, which can be time-intensive and domain-specific.

From a broader perspective, this comparison underscores the importance of selecting an appropriate model architecture based on the characteristics of the dataset and the computational resources available. CNNs are highly suited for tasks involving high-dimensional spatial data, such as image classification, but require careful tuning of hyperparameters, extensive datasets, and significant computational power. In contrast, MLPs are better suited for structured data or tasks where feature engineering can compensate for their lack of spatial awareness.

The discussion also emphasizes the importance of addressing the limitations of these models to improve their effectiveness. For CNNs, techniques such as transfer learning and regularization can help mitigate the need for large datasets, while optimizing the network architecture can reduce computational demands. For MLPs, integrating convolutional layers or using hybrid architectures may allow the model to capture spatial features more effectively, bridging the gap between simplicity and performance.

In conclusion, while CNNs demonstrated superior performance in this task, their limitations in terms of computational complexity and data dependency cannot be overlooked. MLPs, though less effective for image classification, remain a viable option for tasks where simplicity and lower computational requirements are prioritized. The findings from this discussion highlight the need to carefully weigh the trade-offs between model complexity, performance, and resource constraints when selecting an architecture for a given application.

8.6 Conclusion

The main objective of this procedure is to design and train a Convolutional Neural Network (CNN) based model for classifying binarized character images and to validate its performance with a test set and a task dataset. The procedure shows good results in data processing, model design and training optimization. Firstly, random inverse colour and pretzel noise are added to the training data by means of a self-defined data augmentation function (randomAugmentation), and this data augmentation increases the diversity of the data, which helps to improve the model's generalization ability and makes the model more robust in the face of noise and image deformation. In terms of model design, a classical multi-layer convolutional neural network architecture is used, including a convolutional layer, a batch normalization layer, a ReLU activation function, a pooling layer and a fully connected layer. The use of a small convolutional kernel (3×3) enables the model to effectively extract local features, the layer-by-layer stacking of convolutional layers enhances feature representation, and the inclusion of batch normalization and Dropout layers

effectively improves training stability and prevents overfitting. For training optimization, the program chooses the Adam optimizer with appropriate initial learning rate (0.001) and small batch size (64) to enable the model to converge quickly and stably during the training process, and further reduces the risk of overfitting by introducing a validation set to monitor the model performance in real time. The high accuracy of the procedure on the test set demonstrates the reliability and validity of the model in classification tasks. However, there is still room for improvement in the data enhancement strategy of this procedure, e.g., more diverse enhancement methods such as rotation, scaling and translation can be introduced to further extend the generalization capability of the model. In addition, the selection of hyperparameters for the model training process can be tuned more carefully, such as dynamically adjusting the learning rate or adopting a learning rate decay strategy to further optimize the model performance. Overall, this procedure achieves satisfactory results in the character classification task through reasonable architectural design and training strategy, which lays a good foundation for subsequent improvement and application.

References:

[1] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, 2012

[2] P. C. Y. Chen, ""“ME5411 Robot Vision and AI Part II: Deep Learning for Computer-Vision in Robotics," in ME5411 Slides AY2425 Sem1," *National University of Singapore,* 2022-2024.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE*, 1998.

[4] G. Deng; L.W. Cahill, “An adaptive Gaussian filter for noise reduction and edge detection”, *Proceedings of the IEEE*, pp. 7803-1487, 1993.

[5] N. Kanopoulos; N. Vasanthavada; R.L. Baker, “Design of an image edge detection filter using the Sobel operator”, *IEEE Journal of Solid-State Circuits*, 1998

[6] Laith Alzubaidi, “Review of deep learning: concepts, CNN architectures, challenges, applications, future directions”, *Journal of Big Data*, 2021