ME5411 - ROBOT VISION AND AI

NATIONAL UNIVERSITY OF SINGAPORE

COLLEGE OF DESIGN AND ENGINEERING

Computing Project

Authors:
Hao Yuzhi (ID: A0304179A)
Sun Yining (ID: A0310458J)
Tang Ziyue (ID: A0304036U)

Date: November 18, 2024

Abstract

Contents

1 Task 1

1.1 Task Requirement

In this task, the original image must be displayed on the screen. Then, the image was experimented with contrast enhancement, followed by comments on the results.

1.2 Solution

1.2.1 Display

```
im = imread('charact2.bmp');
figure;
imshow(im);
title('Original Input Image: charact2.bmp');
```

Listing 1: MATLAB code to read and display an image

Firstly, use the command `imread("charact2.bmp")` to read an image file named *charact2.bmp* from the current working directory. This image is loaded into the variable `im` as a matrix, where pixel intensity values are stored.
Secondly, `figure` creates a new figure window to display graphical content. This is necessary to ensure that the displayed image does not cover any existing graphics.
Then, use the command `imshow(im)` renders the loaded image in the figure window, as shown in Figure 1. The function automatically scales the pixel values for appropriate visualization, depending on the image type, for example, grayscale or RGB. In addition, add a title to the figure by `title('...')`, providing context about the displayed content.

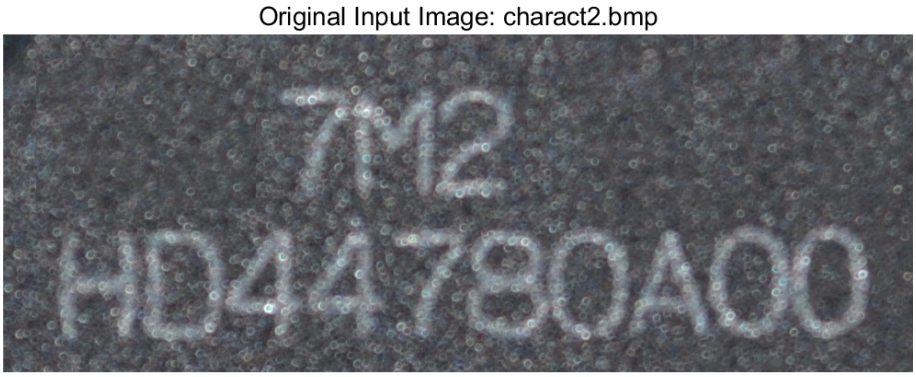

Figure 1: Display the original image on screen

1.2.2 Contrast Enhancement and Comments

```
% Convert from RGB to gray
gray_im = rgb2gray(im);
% Apply contrast enhancement to gray image
gy_imadjust = imadjust(gray_im);
gy_histeq = histeq(gray_im);
gy_adapthisteq = adapthisteq(gray_im);
```

Listing 2: MATLAB code for enhancing the contrast of the image

Firstly, apply the function `rgb2gray` to convert the input RGB image `im` into a grayscale image `gray_im`. This step can remove color information and reduce the image to a single intensity channel, as depicted on the far left of Figure 2. In Figure 3 far left, its histogram is concentrated in the lower intensity range, indicating that the original image has low contrast with dominant dark details.
In this report, three methods are applied to enhance the contrast of the grayscale image:

- `imadjust` [1]

  This applies a contrast adjustment by mapping the intensity values of `gray_im` to a new range. As shown second from the left in Figure 2. While, Figure 3 (the left second) shows that the histogram is stretched over a wider range of intensity values, reflecting the linear mapping applied by the method. However, the overall shape of the histogram remains similar, showing limited enhancement of contrast distribution.

- `histeq` [2]

  This redistributes the intensity values of the input image to equalize the histogram, enhancing the global contrast. As given second from the right of Figure 2. Meanwhile, the second from the right histogram in Figure 3 becomes more evenly distributed, demonstrating that histogram equalization improves global contrast by redistributing the intensity values.

- `adapthisteq` [3]

  This applies adaptive histogram equalization (AHE), which enhances contrast by computing histograms for small tiles of the image. It is particularly effective for local contrast enhancement. The result can be checked in the far right of Figure 2. The last histogram in Figure 3 shows finer intensity level distributions within localized regions, indicating that AHE enhances local contrast, making it particularly suitable for images with complex intensity variations.

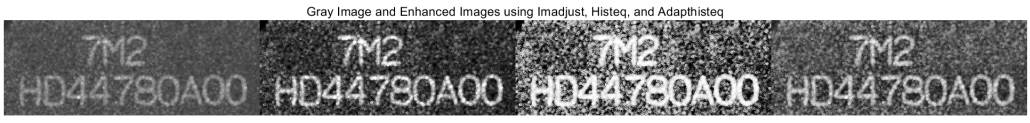

Figure 2: Effects of different contrast enhancement schemes

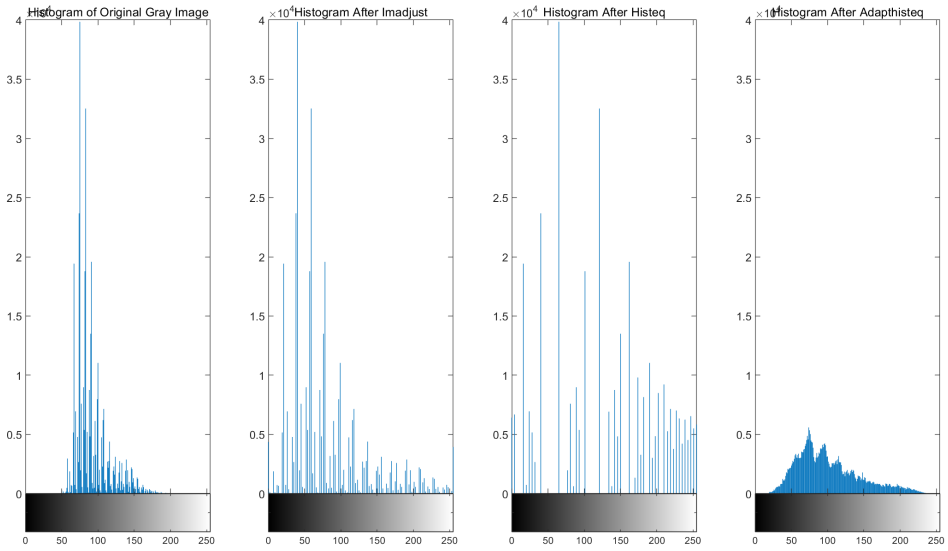

Figure 3: Effects of different contrast enhancement schemes

1.2.3 Reproduction

- `reproduce_imadjust`

  Figure 4 shows that the MATLAB imadjust function and the custom implementation produce the same contrast enhancement results (upper and lower left). On the right, the mapping curve of pixel intensity illustrates the conversion of input grayscale values to adjusted output, clearly aligning the lower limit (1%) and upper limit (99%), confirming the equivalence of the two methods.

Algorithm 1: Reproduce Image Adjustment

Input: $gray\_im$: Input grayscale image, $gy\_frequency$: Grayscale histogram
Output: $result\_im$: Adjusted image

$sorted\_grayscale \leftarrow$ Sort($gray\_im$(:))
$total\_pixels \leftarrow$ Sum($gy\_frequency$)

$lower\_grayscale \leftarrow sorted\_grayscale[\lceil total\_pixels \times 0.01 \rceil]$
$upper\_grayscale \leftarrow sorted\_grayscale[\lfloor total\_pixels \times 0.99 \rfloor]$
$ratio \leftarrow 255/(upper\_grayscale - lower\_grayscale)$

$result\_im[gray\_im \leq lower\_grayscale] \leftarrow 0$
$result\_im[gray\_im \geq upper\_grayscale] \leftarrow 255$
$result\_im[mask] \leftarrow ratio \times (gray\_im[mask] - lower\_grayscale)$, where
$mask = (gray\_im > lower\_grayscale) \wedge (gray\_im < upper\_grayscale)$

$result\_im \leftarrow$ Convert to uint8
return $result\_im$

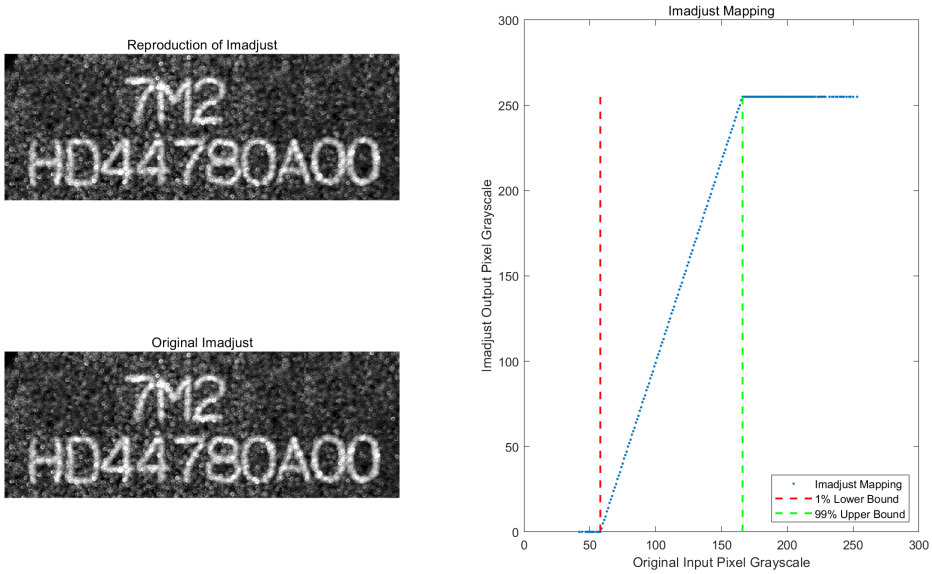

Figure 4: Comparison of custom implementation and MATLAB imadjust

- `reproduce_histeq`

  The provided code implements image enhancement through histogram equalization by redistributing pixel intensities to achieve a more uniform histogram. It calculates the cumulative distribution of grayscale frequencies, constructs a lookup table, and maps input pixel intensities to new values based on the table.

Algorithm 2: Reproduce Histogram Equalization

Input: $gray\_im$: Input grayscale image, $gy\_frequency$: Frequency histogram of grayscale values, $gy\_index\_grayscales$: Index of grayscale values
Output: $result\_im$: Equalized image

Initialize $result\_im$ as a zero matrix of the same size as $gray\_im$
Set $discrete\_gray\_levels \leftarrow 64$
Set $gray\_interval \leftarrow \max(gy\_index\_grayscales)/discrete\_gray\_levels$
$cumulative\_frequency \leftarrow$ Cumulative sum of $gy\_frequency$
$cumulative\_distribution \leftarrow cumulative\_frequency/\max(cumulative\_frequency)$
$look\_up\_table \leftarrow$ Round(Round($cumulative\_distribution \times 255/gray\_interval) \times gray\_interval$)
$result\_im \leftarrow look\_up\_table[gray\_im + 1]$
$result\_im \leftarrow$ Convert to uint8
return $result\_im, look\_up\_table$

Figure 5 demonstrates the effectiveness of the code. The top left image shows the original grayscale input, while the middle left displays the result of MATLAB's histeq function. The

bottom left shows the output from the custom implementation, which closely mirrors MATLAB's result. The mapping curves on the right validate the similarity, with nearly identical transformations between the input and output pixel intensities, confirming that the custom implementation (red) achieves comparable enhancement to MATLAB's tool (blue). However, the custom implementation relies on the cumulative distribution function (CDF) for direct mapping without further optimization of the uniformity of the resulting pixel distribution. This simplification may lead to slight differences in the histogram distribution within certain grayscale intervals compared to MATLAB's more refined approach.

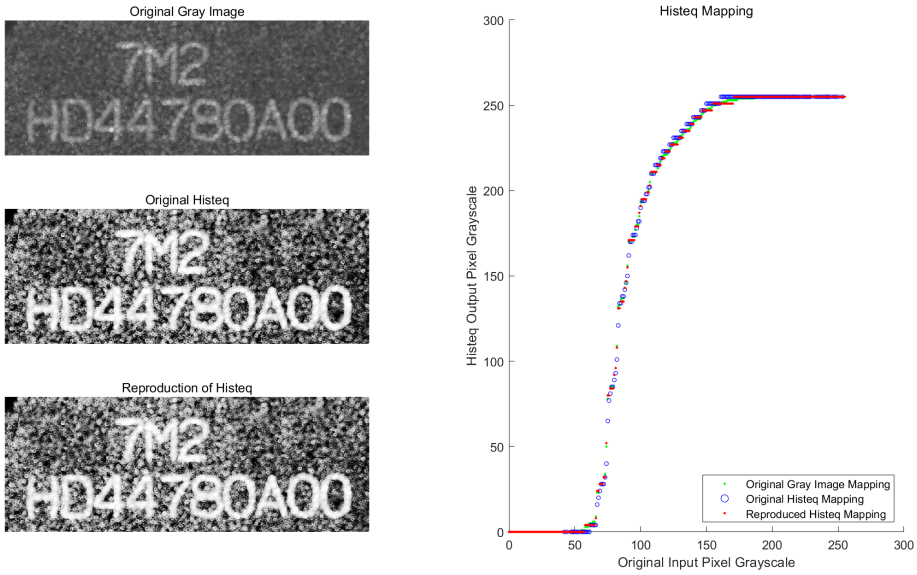

Figure 5: Comparison of custom implementation and MATLAB histeq

- `reproduce_adapthisteq`

  This code implements AHE to enhance image contrast by dividing the image into smaller tiles, performing histogram equalization on each tile, and using bilinear interpolation to smooth transitions between tiles. It incorporates a contrast-limiting mechanism to avoid over-enhancement by clipping excessive frequencies and redistributing them proportionally across valid intensity levels.

  Figure 6 demonstrates the results of the AHE process. The original grayscale image (top left) exhibits low contrast, with most pixel intensities concentrated in a narrow range. The output from MATLAB's adapthisteq (top center) and the custom implementation (top right) shows significantly improved contrast and enhanced visibility of features. The histograms (bottom row) reveal how pixel intensities are more evenly distributed after AHE. Specifically, the custom function achieves a result similar to MATLAB's adapthisteq, effectively enhancing image contrast and redistributing pixel intensities across the available range. A little differences may arise due to simplifications in clipping and redistribution strategies, but the overall performance demonstrates the function's robustness and alignment with MATLAB's

approach.

Algorithm 3: Reproduce Adaptive Histogram Equalization

Input: $gray\_im$: Input grayscale image, num_tiles: Number of contextual regions (default: $[8, 8]$), $clip\_limit$: Maximum contrast factor (default: 0.01)

Output: $result\_im$: Enhanced image

```
Divide image into num_tiles rectangular regions
Calculate max_frequency ← clip_limit × tile size
foreach tile in gray_im do
    Compute histogram of the tile
    Clip frequencies above max_frequency, redistributing excess values proportionally
    Compute CDF of the tile
    Generate lookup table using CDF and map tile pixels to new values
end
Smooth the borders between tiles by interpolating pixel values
Apply interpolation row-wise and column-wise
Convert result_im to uint8 format
return result_im
```

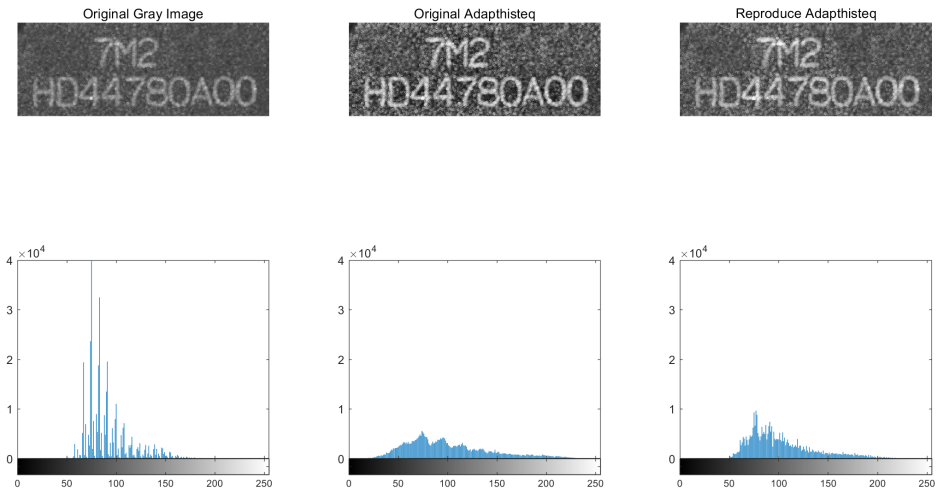

Figure 6: Comparison of custom implementation and MATLAB adapthisteq

1.3 Conclusion

These three methods demonstrate a progression in image enhancement sophistication, from global linear scaling (contrast adjustment) to global intensity redistribution (histogram equaliza-

tion) and localized enhancement with noise suppression (AHE). Each method improves contrast and detail visibility by manipulating pixel intensity distributions, with AHE providing the most localized and adaptive approach. The results highlight the importance of selecting an enhancement technique based on the specific requirements of an image.

2 Task 2

2.1 Task Requirement

In this task, it is asked to implement a 5×5 averaging filter and apply it to an image. Experiment with filters of 4 different sizes and compare the results to analyze the effects of different image smoothing methods.

2.2 Solution

2.2.1 Apply 5×5 Averaging Filter

```
filter_size = 5;
averaging_filter_5 = ones(filter_size) / (filter_size^2);
smoothed_img_5x5 = imfilter(gray_im, averaging_filter_5, 'replicate');
```

Listing 3: MATLAB code to initialize a 5x5 averaging filter and apply it to an image

The first step is to define a filter size for this case. Next, generate a 5×5 matrix filled with ones, then divide each element by the total number of elements to normalize the filter. This ensures the output image does not become brighter or darker.

$$averaging\_filter\_5 = \begin{bmatrix} 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \end{bmatrix}$$

Finally, `imfilter`, a MATLAB function designed for filtering of images, applies the averaging filter `averaging_filter_5` to the grayscale image `gray_im` [4]. The option `'replicate'` is used to handle boundary pixels by replicating edge values, which helps avoid artifacts caused by boundary effects during filtering [4].

2.2.2 Compare Filters with Different Sizes

Using the same method, averaging filters of various sizes (2×2, 10×10, and 50×50) were defined and applied to the grayscale image. The results of these smoothing processes are displayed in Figure 7, showing the effects of different filter sizes on the image quality.
From the comparison:

- Super Small Filter (2×2):

It is almost impossible to tell the difference from the original image with the naked eye. Therefore, a smaller filter is of no practical significance.

- Small Filter (5×5):

  This filter preserves most of the original image details while effectively reducing noise. Text remains sharp and legible, making this filter suitable for applications requiring minimal detail loss.

- Medium Filter (10×10):

  The 10×10 filter produces a stronger smoothing effect, reducing noise more aggressively. However, it also begins to blur fine details, resulting in slight degradation of text clarity. This filter achieves a balance between noise reduction and maintaining image features.

- Large Filter (50×50):

  A large filter will significantly blurs the image, removing nearly all noise but at the cost of canceling text details. While this level of smoothing is useful for detecting large-scale patterns, it is unsuitable for scenarios requiring high detail.

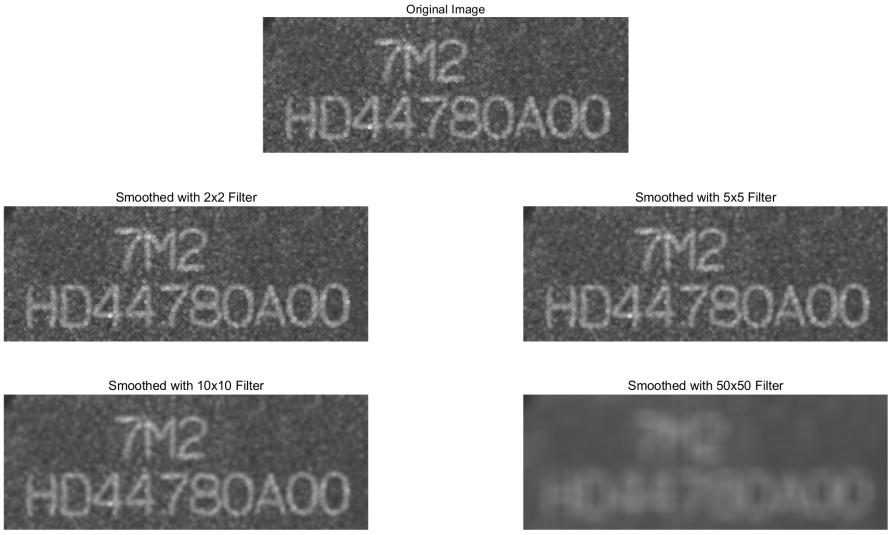

Figure 7: Effects of different filter sizes on the image quality

2.2.3 Reproduction

The custom image filtering function implements a Gaussian blur by creating a custom Gaussian filter kernel, extending the input image boundaries symmetrically to avoid edge effects, and applying the filter kernel to each pixel using a sliding window approach [5]. The extended boundaries

ensure smooth filtering near image edges, and the filter kernel is normalized to maintain intensity consistency across the image.

Algorithm 4: Custom Image Filtering

Input: ori_im: Input image, $sigma\_size$: Size parameter for filter
Output: $result\_im$: Blurred image, $filter\_disk$: Custom filter kernel, mag_filter_disk: Magnified filter for visualization

Create a 2D Gaussian filter using $sigma\_size$
Normalize the filter and adjust for numerical errors
$filter\_disk \leftarrow$ Normalized Gaussian filter
Extend ori_im symmetrically by $sigma\_size$ on all sides
Initialize $result\_im$ as a zero matrix with the same size as ori_im
foreach pixel in ori_im **do**
    Extract the corresponding region from the extended image
    Perform element-wise multiplication with $filter\_disk$
    Compute the sum of the resulting values and assign it to the pixel in $result\_im$
end
Convert $result\_im$ to uint8 format and return
return $result\_im, filter\_disk, mag\_filter\_disk$

Figure 8 illustrates the results of the custom filtering process. The "Original Image" demonstrates high-frequency noise and sharp transitions, while the "DIY Filtered Image" exhibits a noticeable smoothing effect, reducing noise and enhancing continuity in intensity transitions. The "DIY Filter in 3D" and "Magnified DIY Filter in 2D" visualizations represent the custom Gaussian kernel used for filtering, emphasizing its smooth distribution and impact on the blurring process. They also show the huge difference between the custom filter and the disk filter kernel defined by `averaging_filter_5`.

2.3 Conclusion

Filters play a critical role in image processing by enhancing or suppressing specific image features. The 5x5 averaging filter smooths the image by reducing noise and fine details through uniform averaging of pixel intensities, while increasing the filter size further enhances the smoothing effect at the expense of image sharpness. Comparing custom filters and MATLAB's imfilter, the key differences lie in boundary handling and kernel generation. The custom filter uses symmetric extension for boundaries and specifically design kernels, Gaussian distributions, for tailored flexibility. However, they are computationally expensive. In contrast, imfilter is highly optimized, offering multiple predefined padding methods for efficient and robust processing, for instance, symmetric, replicate.

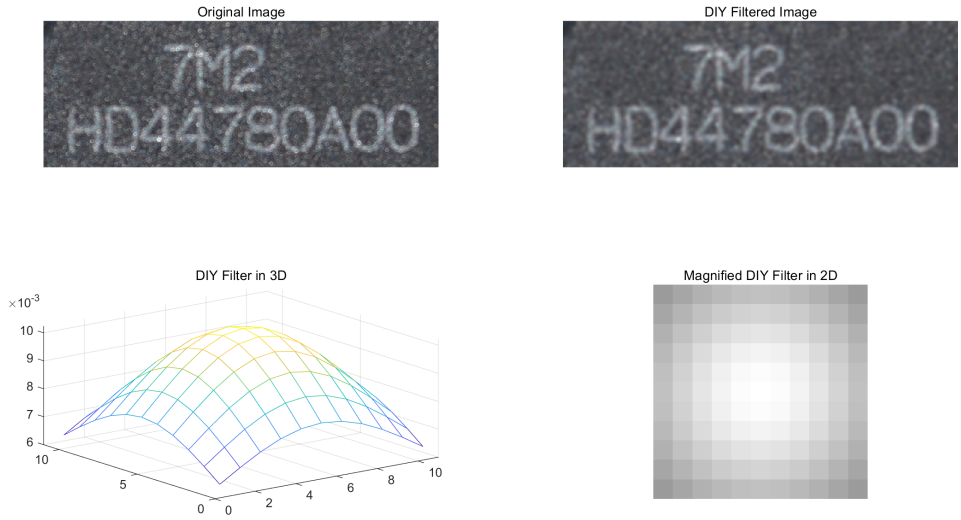

Figure 8: The results of the custom filtering process

3 Task 3

3.1 Task Requirement

The objective of Task 3 is to implement and apply an ideal high-pass filter on an image to enhance high-frequency components, which typically represent edges and details. The critical aspect of Task 3 is to utilize the high-pass filter in the frequency domain, and then convert the filtered image back to the spatial domain to observe the effect of filtering on the image.

3.2 Solution

1. Image Preprocessing:

   Firstly, convert the original image to grayscale using `rgb2gray` function of MATLAB, reducing computational requirements and focusing on the brightness information.

2. Fourier Transform [6, 7]:

   Perform Fourier transform to the grayscale image using `fft2` function to convert the image from the spatial to the frequency domain. The `fftshift` function is then applied to rearrange the frequency components, moving low frequencies to the center and high frequencies to the edges, enabling targeted manipulation of the image's frequency components.

3. High-Pass Filter Design and Application:

   Firstly, create a frequency-coordinate `meshgrid (U, V)` with u and v representing the horizontal and vertical frequency coordinates centered on the image respectively. Then, calculate the Euclidean distance from the center for each point to create a `distance2center`

matrix. Then, determine a high-pass filter mask by computing the cumulative energy ratio and setting an appropriate `target_ratio` (select 0.9 based on the characteristics of this figure) as a threshold to retain high-frequency components. Lastly, construct the filter mask `high_pass_mask` by using double(`distance2center ¿ target_distance`) and multiply it with the frequency-domain image to apply the filtering.

4. Inverse Fourier Transform [8, 9]:

   To convert the filtered image back into the spatial domain, utilize `ifftshift` and `ifft2` functions. Finally, save the real-number part of the result image and ensure that pixel values remain within the displayable range [0,255].

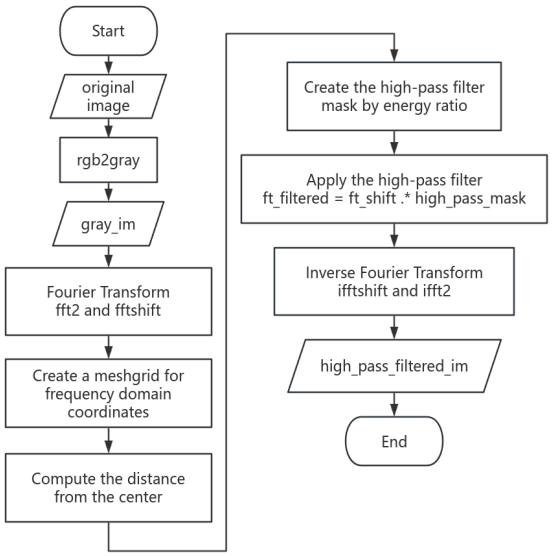

Figure 9: High-pass filter framework

3.3 Result and Analysis

The following Figure 10 displays the `energy_ratio` values. It can be clearly observed that selecting a value range from 0.04 to 0.96 as the `target_ratio` effectively balances the inclusion of significant frequency components while minimizing the impact of noise and irrelevant details. Thus, in this case, `target_ratio` is set to 0.9 to emphasize the most critical frequency components.

The use of `energy_ratio` is particularly beneficial as it provides a quantitative measure of each frequency component's contribution to the overall image structure. Unlike using the raw amplitude values, the energy, represented as the square of the frequency magnitude

$$|F(u,v)|^2$$

, captures the strength and significance of each frequency component in relation to the total image content. This ensures that the filter prioritizes components with meaningful contributions while effectively reducing noise.

energy_ratio

363330x1 double

	1	2
181843	0.0271	
181844	0.0275	
181845	0.0276	
181846	0.0276	
181847	0.0293	
181848	0.0312	
181849	0.9688	
181850	0.9707	
181851	0.9724	
181852	0.9724	
181853	0.9725	
181854	0.9729	
181855	0.9729	
181856	0.9730	

Figure 10: Significant change in `energy_ratio` at 0.9688

The high-pass filter, guided by the `target_ratio`, significantly enhances the edges and details within the image, demonstrating an important role of high-frequency components in image sharpening. By suppressing low-frequency content, blurred regions are effectively reduced, resulting in sharper edges and contours. This process highlights key structural features of the image while retaining only the most relevant information.

Figure 11: High-pass filter result and comparison

3.4 Compare with Task 2

The high-pass filter works in contrast to the low-pass filter. While the low-pass filter preserves low-frequency information, leading to image smoothing and edge blurring (with reduced noise but loss of detail), the high-pass filter is more suitable for detail enhancement and feature emphasis, particularly in applications requiring edge detection or image sharpening.
Additionally, implementing the high-pass filter in the frequency domain enables precise control over the separation and extraction of frequency components. By adjusting the target ratio in the mask, it becomes possible to flexibly control the degree of edge and detail enhancement. When trying to design an optimal high-pass filter mask, we initially attempts to adjust the `target_distance` directly. However, this proved challenging in finding the most effective distance for the ideal high-pass filter. Through further exploration, we finally adopt an energy ratio-based approach, allowing the selection of an appropriate `target_ratio` threshold. This method successfully highlighted image edges and details while avoiding excessive sharpening that could amplify noise. This task provided us with deeper insights into the role of high-pass filtering in frequency-domain image processing and the importance of selecting an appropriate threshold. Through the comparison with low-pass filtering, we gained a clear understanding of the different effects of high and low-frequency components in image.

4 Task 4

4.1 Task Requirement

In Task 4, the aim is to extract a sub-image containing the middle line and ”HD44780A00” from the original image. This extraction is necessary to narrow down the focus to a specific region of interest, allowing for a more detailed analysis of the lower part of the image in subsequent tasks.

4.2 Solution

```
% Find the middle line and the image range below it
middle_line = ceil(h/2:h);
% Apply the range to cut the image
sub_im = im(middle_line, :, :);
sub_gray_im = gray_im(middle_line, :, :);
[sub_h, sub_w, sub_c] = size(sub_im);
```

Listing 4: MATLAB code to extract a sub-image

Firstly, identify the position of the middle line, which locates the row corresponding to the grayscale image’s central line. Then, extract the sub-image containing the middle line. Then display or save the result as needed.

4.3 Results and Analysis

In Task 4, we extract the sub-image by referencing row numbers directly, which is a straightforward and efficient method. Since the size of the input figure is 367×990, the sub-image includes the

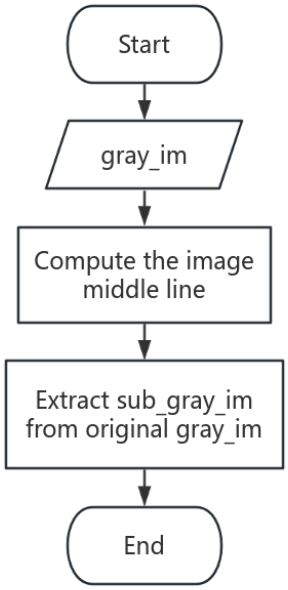

Figure 12: Sub-image extraction framework

middle line should have a hight of $(367+1) \div 2 = 184$. The result sub-image successfully includes the middle line and the “HD44780A00” text as shown in the following Figure 13.

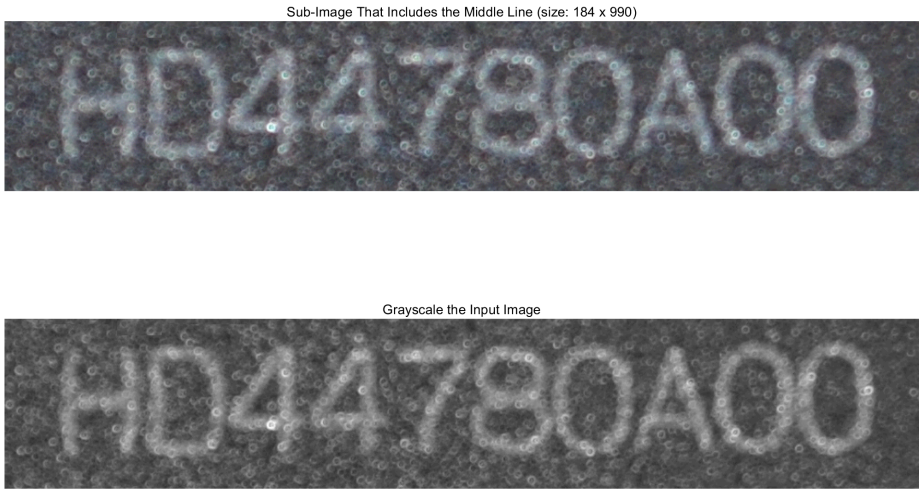

Figure 13: Sub-image extraction result

5 Task 5

5.1 Task Requirement

The objective of Task 5 is to convert the extracted sub-image into a binary image, emphasizing key features of the image by retaining just black and white colors. This conversion enhances the clarity of essential details within the region of interest, making them easier to identify.

5.2 Solution

```
% Image binarization using a tool
binary_im = imbinarize(sub_gray_im);
```

Listing 5: MATLAB code to convert the sub-image into a binary image by a tool.

Firstly, convert the grayscale sub-image `sub_gray_im` to a binary image directly using the `imbinarize` function [10]. This function automatically determines an optimal threshold, which divides pixel values into 0 (black) and 1 (white), separating the foreground (e.g., characters) from the background. The result is displayed on Figure 14 in the middle.

5.2.1 Reproduction

```
% Image binarization using a custom function
rpd_binary_im = reproduce_imbinarize(sub_gray_im);
```

Listing 6: MATLAB code to convert the sub-image into a binary image by a tool.

The provided function implements Otsu's method to binarize an input grayscale image. Otsu's method maximizes the between-class variance to determine the optimal threshold that separates the foreground from the background [11]. The function computes a histogram, cumulative sums, and between-class variance, identifying the threshold that maximizes the variance and using it to binarize the image. Its pseudo code is as follows.

Algorithm 5: Reproduce Image Binarization (Otsu's Method)

Input: ori_im: Input grayscale image
Output: $result\_im$: Binarized image

$counts, index \leftarrow$ imhist(ori_im)
$\omega_1 \leftarrow$ Cumulative sum of normalized counts
$\mu \leftarrow$ Cumulative mean weighted by ω_1
Replace NaN values in μ with 0
$\mu_{total} \leftarrow \mu[end]$
$\sigma_b^2 \leftarrow (\mu - \omega_1 \times \mu_{total})^2 / (\omega_1 \times (1 - \omega_1))$
$threshold \leftarrow$ Index of maximum σ_b^2
$result\_im \leftarrow (ori\_im > threshold)$
return $result\_im$

Otsu's method is a global thresholding technique that separates an image into foreground and background by maximizing the between-class variance. The method calculates weights (ω_1 and ω_2), means (μ_1 and μ_2), and total mean (μ_T) of two pixel classes (foreground and background) based on a threshold. The between-class variance is expressed as:

$$\sigma_B^2 = \omega_1(\mu_1 - \mu_T)^2 + \omega_2(\mu_2 - \mu_T)^2$$

and it is also known that:

$$\omega_2 = 1 - \omega_1$$

$$\mu_T = \omega_1\mu_1 + \omega_2\mu_2$$

The function uses these formulas can simplify σ_B^2 to:

$$\sigma_B^2 = \omega_1(1 - \omega_1)\frac{(\mu - \omega_1\mu_T)^2}{\omega_1(1 - \omega_1)}$$

The derivation of the formula can be found in the appendix, and this algorithm does ensure the effective calculation of the optimal threshold.

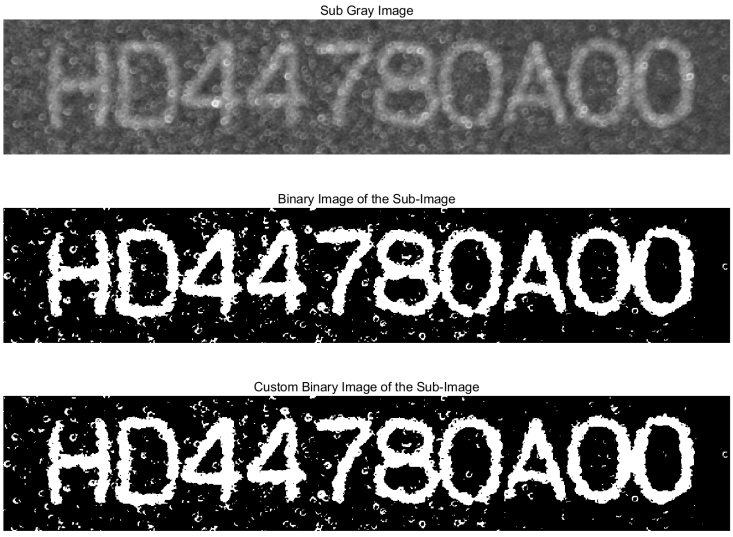

Figure 14: Binary image generated by different functions

5.3 Conclusion

The binary image presents a clear black-and-white contrast, highlighting key features and making them more visible and distinguishable. Compared to the grayscale input, the binary version emphasizes critical details more effectively, allowing for cleaner and more interpretable visualization of essential structures.

The custom binarization approach, based on Otsu's Method, determines the ideal threshold by maximizing the variance between classes. This method successfully separates the foreground (region of interest) from the background, producing a well-segmented image. Figure 14 illustrate the effectiveness of this approach, showcasing the original grayscale input and its binarized output. The computed threshold effectively sketches these characters, achieving the same segmentation result to MATLAB's `imbinarize`.
The custom implementation replicates the functionality of MATLAB's `imbinarize`, using the same principles of Otsu's Method to identify the optimal threshold. Minor differences in implementation details, such as manual handling of NaN values, do not impact the overall output quality. This demonstrates that the custom function achieves equivalent performance, ensuring robust and accurate image binarization.

6 Task 6

6.1 Task Requirement

The objective is to determine the outlines of characters within the input image by employing a sequence of image preprocessing and segmentation techniques.

6.2 Solution

This code performs preprocessing and edge detection on a character image. It involves multiple steps to clean, enhance, and segment the image, ultimately identifying the character outlines.

1. Load and Preprocess the Image:

   The image is loaded and converted to grayscale if it is a color image. This step ensures the image is in a simplified format suitable for processing.

2. Enhance the Image:

   Histogram adjustment is applied to enhance the contrast of the grayscale image, making features like edges and characters more distinct.

3. Denoise the Image:

   A 5×5 median filter is used to remove noise while preserving edges. This improves clarity and reduces irrelevant noise.

4. Binarize the Image:

   The denoised image is converted into a binary format (black and white) using a thresholding method. This reduces complexity and makes subsequent morphological operations more effective.

5. Morphological Operations:

   - An opening operation (erosion followed by dilation) is applied with a disk-shaped structuring element to remove small noise.

- A closing operation (dilation followed by erosion) is applied to fill gaps within characters, ensuring better connectivity of the foreground.

6. Invert the Binary Image:

   The binary image is inverted to make the characters black and the background white, which is better suited for edge detection.

7. Edge Detection:

   The Canny edge detection algorithm is applied to identify the outlines of the characters in the binary image. This step highlights the shapes and boundaries of the characters for further analysis.

Algorithm 6: Character Image Preprocessing and Edge Detection

Input: I: Input image file ('charact2.bmp')
Output: $edges$: Detected character edges

Read the input image I
if I **is a color image then**
    Convert I to grayscale: $sub\_gray\_im \leftarrow rgb2gray(I)$
end
Display the original grayscale image
Enhance contrast using histogram adjustment: $im\_enhanced \leftarrow imadjust(sub\_gray\_im)$
Display the enhanced image
Apply a 5×5 median filter: $denoised\_im \leftarrow medfilt2(im\_enhanced)$
Display the denoised image
Convert to binary: $binary\_im \leftarrow imbinarize(denoised\_im)$
Display the binarized image
Perform opening operation to remove noise:
$cleaned\_im\_open \leftarrow imopen(binary\_im, strel('disk', 6))$
Perform closing operation to fill gaps:
$BW\_eroded \leftarrow imclose(cleaned\_im\_open, strel('disk', 3))$
Display the cleaned and filled image
Invert the binary image: $inverted\_im \leftarrow imcomplement(BW\_eroded)$
Apply Canny edge detection: $edges \leftarrow edge(BW\_eroded, 'Canny')$
Display the edge-detected image
return $edges$

6.2.1 Core Function

- `imopen` (Morphological Opening) [12]:

  The `imopen` function performs morphological opening on a binary image. This operation is defined as an erosion followed by a dilation using a specified structuring element (`strel`). Morphological opening is typically used to remove small objects or noise while preserving

the shape and size of larger objects. In this implementation, the structuring element is defined as a disk of radius 6, effectively eliminating small noise around the edges of characters without distorting their structure.

- `imclose` (Morphological Closing) [13]:

  The `imclose` function applies morphological closing, which is a dilation followed by an erosion. This operation is particularly useful for filling small gaps or holes within objects. In the script, a disk-shaped structuring element with a radius of 3 is used to close gaps within the characters, ensuring smoother and more continuous shapes for edge detection and analysis.

- `edge` (Edge Detection) [14]:

  The `edge` function is used for detecting edges in a binary image. Edges are detected where intensity changes significantly, often corresponding to object boundaries. This function supports multiple methods, and in this case, the `Canny` method is employed for its robustness in detecting edges with noise resistance.

  The Canny method is a multi-stage algorithm that identifies edges by maximizing gradients while minimizing false detections due to noise. It involves:

  1. **Gaussian Filtering**: Smooths the image to reduce noise.
  2. **Gradient Computation**: Identifies edge candidates using gradient magnitude and direction.

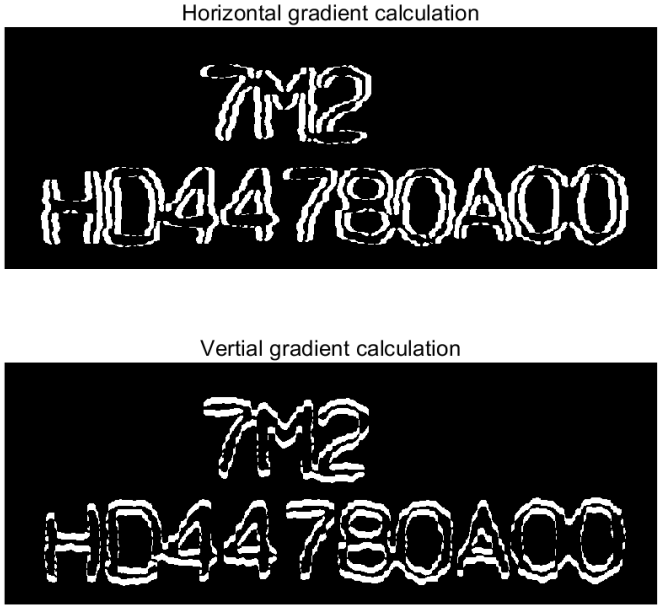

Figure 15: Gradient calculation result for horizontal and vertical

3. **Non-Maximum Suppression (NMS)**: Refines edges by retaining only the local maxima of gradient magnitudes.
   (a) Gradient magnitude:
   $$\text{mag\_gradient} = \sqrt{dx^2 + dy^2}$$
   (b) Gradient direction:
   $$\theta = \arctan 2(dy, dx)$$
   (c) The direction θ is quantized into one of four principal directions:
   - **Horizontal (0°):** Compared with the left and right neighboring pixels.
   - **Vertical (90°):** Compared with the top and bottom neighboring pixels.
   - **Positive Diagonal (45°):** Compared with the bottom-left and top-right neighboring pixels.
   - **Negative Diagonal (135°):** Compared with the top-left and bottom-right neighboring pixels.

   For each pixel, its mag_gradient(i, j) is compared with the magnitudes of its neighboring pixels along the quantized gradient direction.
4. **Hysteresis Thresholding**: Finalizes edges based on user-defined thresholds to determine strong and weak edges. This method is widely recognized for its accuracy and precision in edge detection.
   - **Strong edge pixels:** Pixels with gradient magnitudes greater than a high threshold (highThresh = 0.9).
   - **Weak edge pixels:** Pixels with gradient magnitudes between a low threshold (lowThresh = 0.2) and the high threshold.
   - **Non-edge pixels:** Pixels with gradient magnitudes below the low threshold.

6.3 Result

Through the above steps, whose images can be seen in Figure 16, the edges of the characters in the image are obtained, shown in Figure 17.

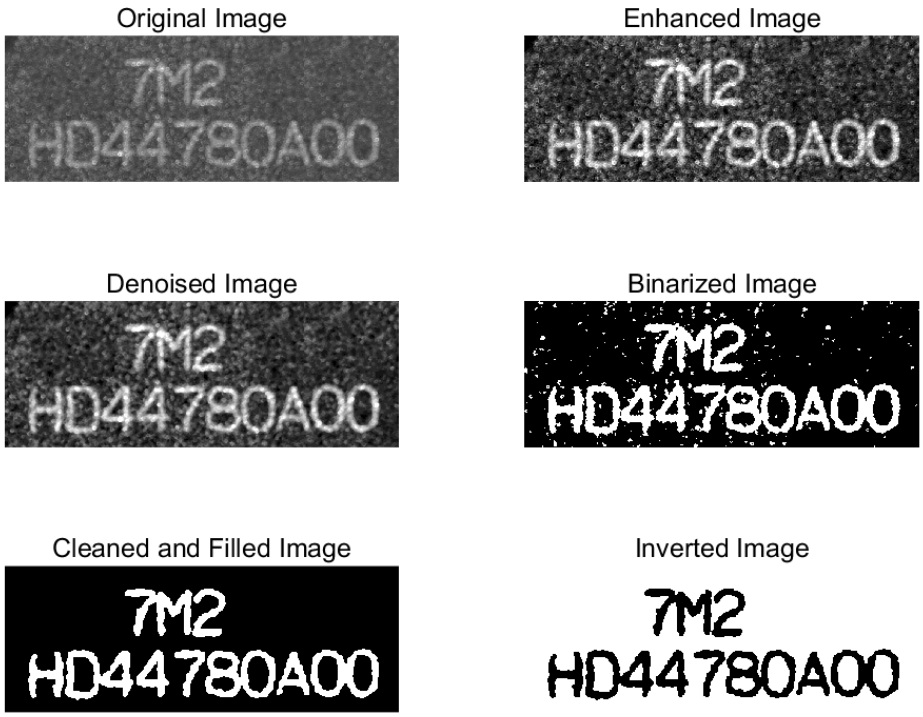

Figure 16: Image processing of determine the outlines of characters

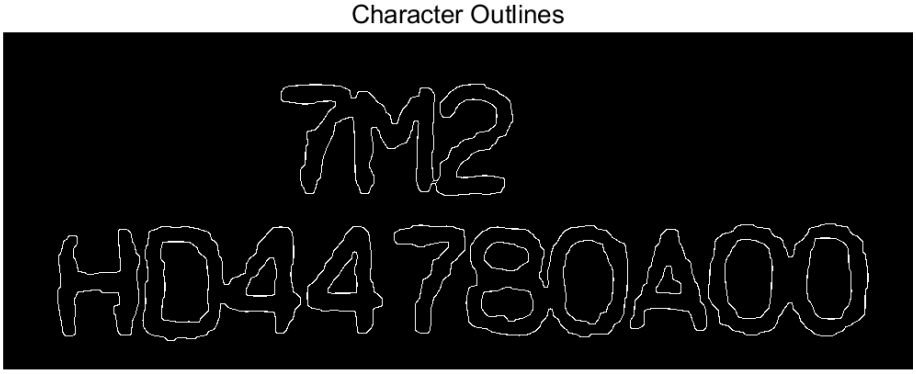

Figure 17: Characters Outlines

7 Task 7

7.1 Task Requirement

The goal is to segment the input image to distinctly separate and label individual characters. The process should ensure that each character is isolated with minimal overlap or ambiguity, enabling clear identification and differentiation of all characters in the image.

7.2 Solution

1. **Label Connected Components**:

   All connected regions in the binary image are identified and labeled as distinct objects. The labeled regions are visualized with a unique color for each component.

2. **Divide the Image into Top and Bottom Halves**:

   The image is split into two halves along its midline. Each half is processed separately to calculate character widths and segment them appropriately.

3. **Calculate Average Character Width**:

   For each half of the image, the average width of the characters is estimated by analyzing the distribution of white pixels in columns.

4. **Split Wide Contours**:

   For contours with excessive width, calculate the nearest integer multiple of the average width and split them into multiple parts. Splitting wide contours helps better identify and process characters.

5. **Filter Out Small Contours**:

   Small regions with an area below a predefined threshold are removed, eliminating noise and irrelevant details.

6. **Relabel Filtered Components**:

   After filtering, the remaining regions are relabeled to create a clean segmentation. The new labeled regions are visualized in a final labeled image.

7.3 Result

Figure 19 illustrates the results of character segmentation and labeling, where each character has been distinctly outlined and assigned a unique label, effectively separating the characters from the background for clear identification.

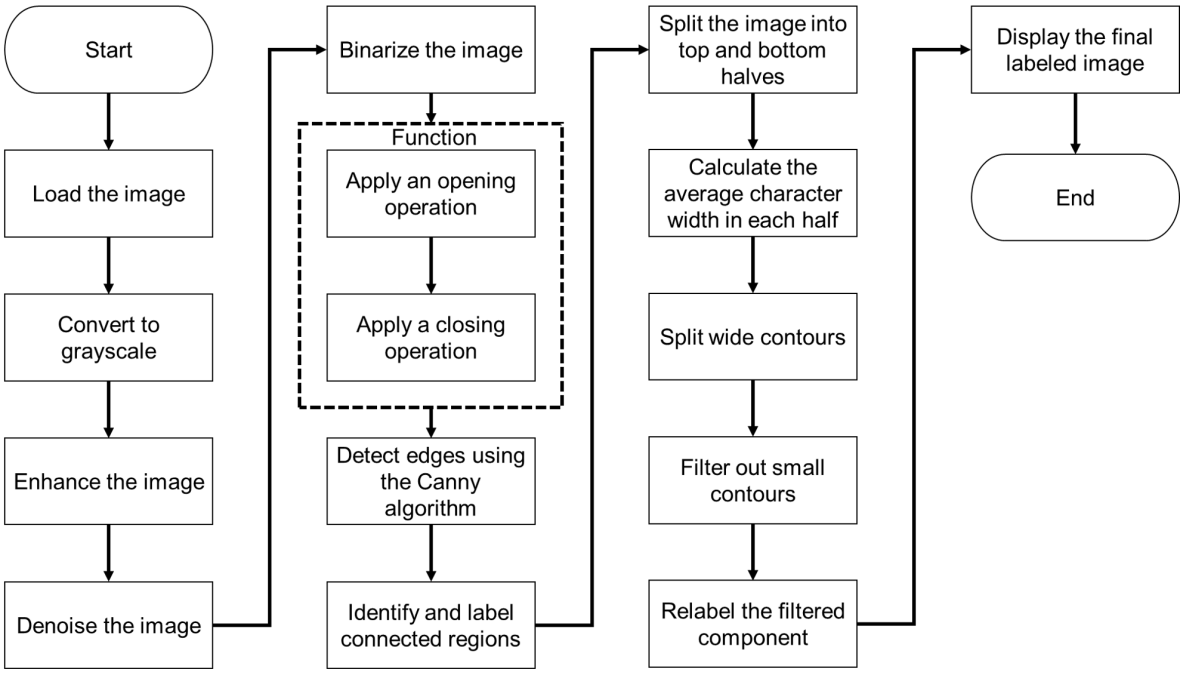

Figure 18: Total framework for task 6 and 7

Figure 19: Segmented and labeled characters with clear outlines

8 Task 8 and Task 9

8.1 Task Requirement

The part combines Task 8 and Task 9, which involves classifying characters in a provided dataset (p_dataset_26.zip) using two different approaches and comparing their performance. The dataset should be divided into a training set (75%) and a validation set (25%). The first task requires designing and implementing a Convolutional Neural Network (CNN) to classify the characters in "Image 1," training it on the training set and evaluating its performance on the validation set [15]. The second task involves developing a classification system using a non-CNN-based method (or a combination of such methods) covered in Part 2 of the course, and similarly training and evaluating it. Then compare the results from the two approaches, analyzing their effectiveness and efficiency, and providing an explanation for any observed differences in performance.

8.2 Task 8.0: Preprocessing

Before conducting the character classification in Task 8, it is essential to extract individual characters from the original image. The image preprocessing follows the same procedures as in Task 7.

During the character extraction process, regions are first filtered based on predefined minimum and maximum bounding box dimensions, which ensures that the extracted regions adhere to character-like features while minimizing interference from non-target areas. Regions that are too small are likely noise or non-character elements, while excessively large regions often result from merged characters. To address this, a character-splitting algorithm is applied to separate connected characters effectively. Next, individual character regions are cropped from the processed image using the detected bounding boxes. The size and position of these bounding boxes are automatically calculated by the image processing algorithm to preserve the integrity and recognizability of each character as much as possible.

To visually verify the results, bounding boxes for each detected character are overlaid on the original image. This provides a straightforward way to inspect the accuracy of the extraction process.

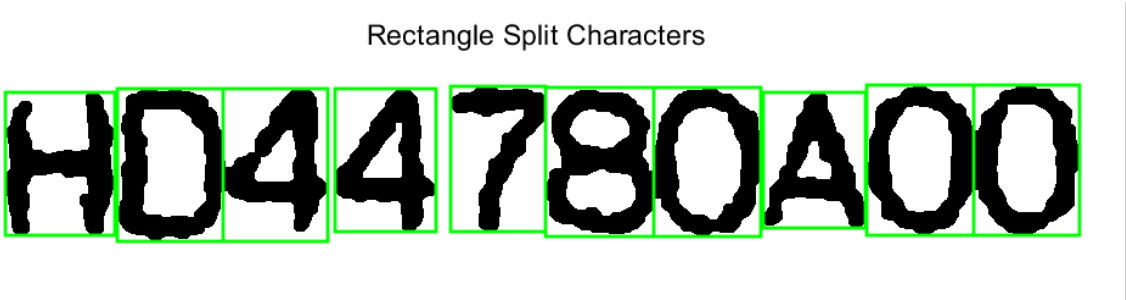

Figure 20: Rectangle split characters result

Finally, each segmented character is saved in a designated folder (‘characters/data‘), ensuring standardized input for subsequent classification or feature extraction tasks.

Figure 21: Each segmented characters

8.3 Task 8.1: CNN

8.3.1 Introduction

The provided pseudocode algorithms detail the processes involved in training and using a CNN for image-based tasks, especially for characters recognition from images. Each algorithm serves a specific purpose within the larger framework of CNN operations.

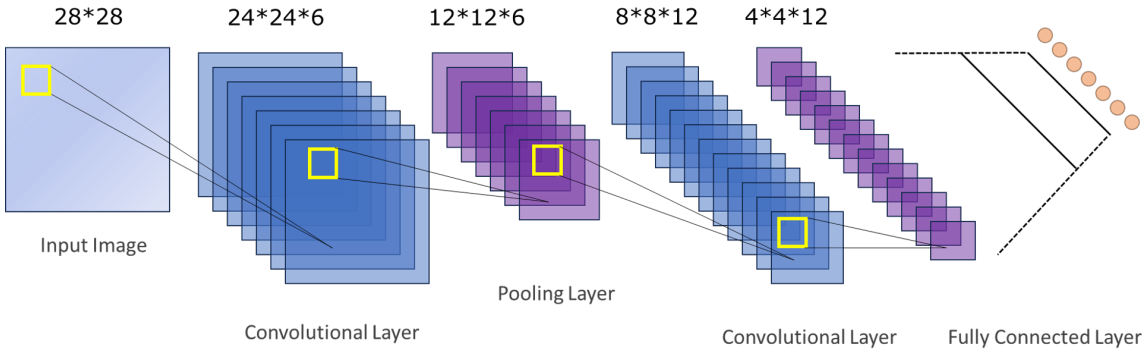

Figure 22: CNN network architecture diagram

8.3.2 Algorithms and Processing Steps

1. **Training a CNN Model**:

   - **Functionality**:
     This algorithm contains the entire training process of a CNN model. It begins with initializing the architecture, including weights and biases, and progresses through multiple epochs where it iteratively updates the model parameters based on the training

data. Each epoch consists of a forward pass, loss computation, backpropagation, and updating weights and biases using computed gradients. Additionally, the model's performance is evaluated on a validation set after each epoch to monitor improvements and avoid overfitting. Finally, the trained model is evaluated on a test set to determine its accuracy.

- **Purpose**:

The primary goal of this algorithm is to iteratively refine the CNN's parameters, weights and biases, to minimize the loss function, which measures the difference between the predicted outputs and the actual labels. By continuously improving these parameters, the CNN learns to make more accurate predictions, ultimately enhancing its performance on test set, which is not seen before.

Algorithm 7: Training a CNN Model

Input: X: Training data, y: True labels, X_{val}: Validation set, y_{val}: Validation labels, X_{test}: Test set, y_{test}: Test labels
Output: $model$: Trained CNN model

Initialize CNN architecture
Initialize weights and biases
for each epoch do
    for each batch of training data do
        Perform forward pass: $output \leftarrow cnnff(batch\_X)$
        Compute loss: $loss \leftarrow LossFunction(output, batch\_y)$
        Perform backpropagation: $gradients \leftarrow cnnbp(output, batch\_y)$
        Update weights and biases: $model \leftarrow cnnapplygrads(model, gradients)$
    end
    Evaluate the model on the validation set: $val\_accuracy \leftarrow Evaluate(model, X_{val}, y_{val})$
    Display validation accuracy for the current epoch
end
Evaluate the final model on the test set: $test\_accuracy \leftarrow Evaluate(model, X_{test}, y_{test})$
Display test accuracy
return $model$

Algorithm 8: Applying Gradients to Update Weights and Biases

Input: $model$: CNN model, $gradients$: Computed gradients, $learning\_rate$: Learning rate
Output: $model$: Updated CNN model

```
for each layer in the network do
    if layer is convolutional or fully connected then
        for each weight in the layer do
            Update weight:
            weight ← weight − learning_rate · weight_gradient
        end
        for each bias in the layer do
            Update bias:
            bias ← bias − learning_rate · bias_gradient
        end
    end
end
return model
```

2. **Backpropagation for CNN (cnnbp)**:

   - **Functionality**:

     This algorithm details the backpropagation process used to compute the gradients necessary for updating the weights and biases of the CNN. It starts by calculating the error between the predicted outputs and the true labels. The error is then propagated backward through the network: first through each fully connected layer and then through each convolutional layer, including any pooling layers. During this process, gradients of the loss with respect to each parameter are calculated. These gradients are used to update the parameters in a subsequent step.

   - **Purpose**:

     Backpropagation is critical for training neural networks as it enables the model to learn from the errors it makes. By computing how much each parameter contributed to the error, it allows the network to adjust its parameters to reduce the overall loss, thereby improving the model's prediction accuracy.

Algorithm 9: Backpropagation for CNN (cnnbp)

Input: $predicted\_output$: Output from forward pass, $true\_labels$: Ground truth labels, $model$: CNN model

Output: $gradients$: Computed gradients for weights and biases

```
Calculate the output error:
error ← predicted_output − true_labels
for each fully connected layer (in reverse order) do
    Compute gradient of the loss with respect to weights and biases
    Calculate gradient of the activation function
    Propagate error to the previous layer
end
for each convolutional layer (in reverse order) do
    if pooling was applied then
        Upsample error from pooling layer
    end
    Compute gradient of the activation function
    Calculate gradient of the convolution operation with respect to filter weights
    Propagate error to the previous layer
end
Store computed gradients for weights and biases
return gradients
```

3. **Forward Pass for CNN (cnnff)**:

   - **Functionality**:

     This algorithm describes the forward propagation phase of a CNN, where input data is processed through multiple layers of the network to produce a final output. The data passes through convolutional layers where it is convolved with filters and subjected to activation functions (e.g., ReLU) and possibly pooling operations. The output of these layers is then flattened and passed through fully connected layers that further process the data by applying weights, biases, and additional activations (e.g., softmax for classification tasks). The final output is the predicted probabilities or scores.

   - **Purpose**:

     The forward pass is essential for both training and using a CNN. During training, it provides the initial predictions that are necessary for calculating the loss. When using the trained model, the forward pass computes the predictions that determine the model's output on new, unseen data.

Algorithm 10: Forward Pass for CNN (cnnff)

Input: $data$: Input data, $model$: CNN model
Output: $output$: Predicted probabilities or scores

Set input layer with data:
$input \leftarrow data$
for each convolutional layer in the network do
  for each filter in the layer do
    Perform convolution operation:
    $convolved\_output \leftarrow Convolve(filter, input)$
    Apply activation function (e.g., ReLU):
    $activated\_output \leftarrow Activation(convolved\_output)$
    if pooling is applied then
      Apply pooling operation (e.g., max pooling):
      $pooled\_output \leftarrow Pool(activated\_output)$
    end
  end
  Set $input \leftarrow pooled\_output$ for the next layer
end
Flatten the output from the last pooling layer:
$flattened\_output \leftarrow Flatten(pooled\_output)$
for each fully connected layer in the network do
  Perform matrix multiplication with weights:
  $weighted\_sum \leftarrow Weights \cdot flattened\_output$
  Add bias:
  $weighted\_sum \leftarrow weighted\_sum + Bias$
  Apply activation function (e.g., ReLU or softmax for the final layer):
  $flattened\_output \leftarrow Activation(weighted\_sum)$
end
Return the final output:
$output \leftarrow flattened\_output$

8.3.3 Result and Analysis

The performance of the CNN model can be comprehensively analyzed through various metrics depicted in the provided figures below. Each of these metrics offers insights into different aspects of the model's ability to classify images into predefined classes (0, 4, 7, 8, A, D, H).

- **Performance Metrics for Each Class**:

  Accuracy: The model shows consistently high accuracy across all classes, nearly perfect, indicating its effectiveness in correctly identifying class labels.

Precision: Uniformly high precision suggests that the model's predictions are reliable, minimizing the risk of false positives, which is crucial in critical applications.

Recall: While slightly varied, recall remains high across classes, demonstrating the model's ability to capture the majority of relevant instances.

F1 Score: The F1 scores are consistently high across all classes, showcasing the model's balanced and robust performance.

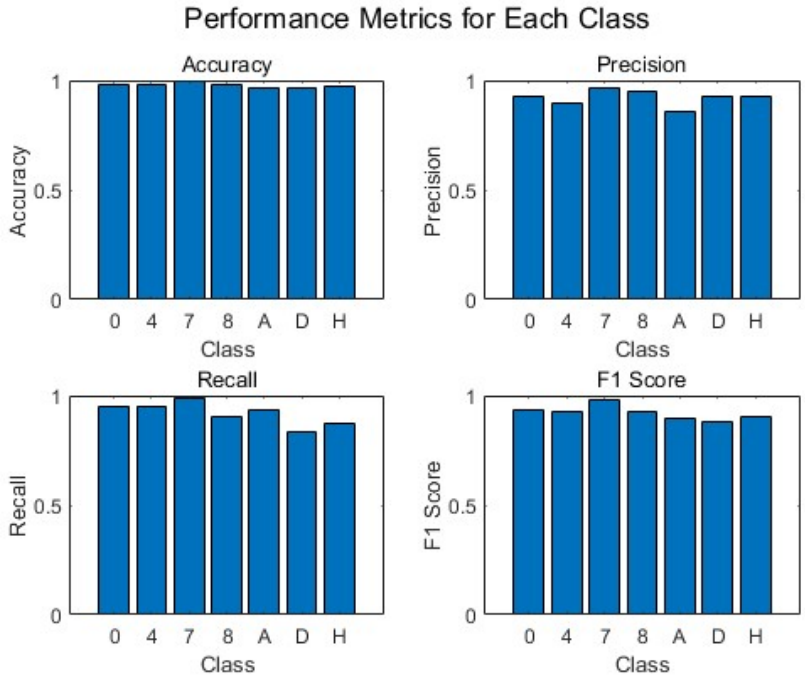

Figure 23: Performance metrics

- **Confusion Matrix**:

  The confusion matrix further provides granularity into the model's performance, showing the exact number of predictions for each class compared to the actual labels:

  Classes such as '4', '7', and 'A' show high numbers of true positives (diagonal elements), with relatively few misclassifications, indicating clear class separability by the model. Some confusion is observed between classes '0', 'D', and 'O', which may be due to visual similarities in these characters.

- **Sample Predictions**:

  Sample predictions for each class reveal practical insights into the model's performance:

  Correct predictions across a variety of classes demonstrate the model's effectiveness in handling different styles and deformations within the characters. Misclassifications seen in the prediction examples point to potential areas of improvement, particularly in distinguishing between characters with similar shapes or strokes.

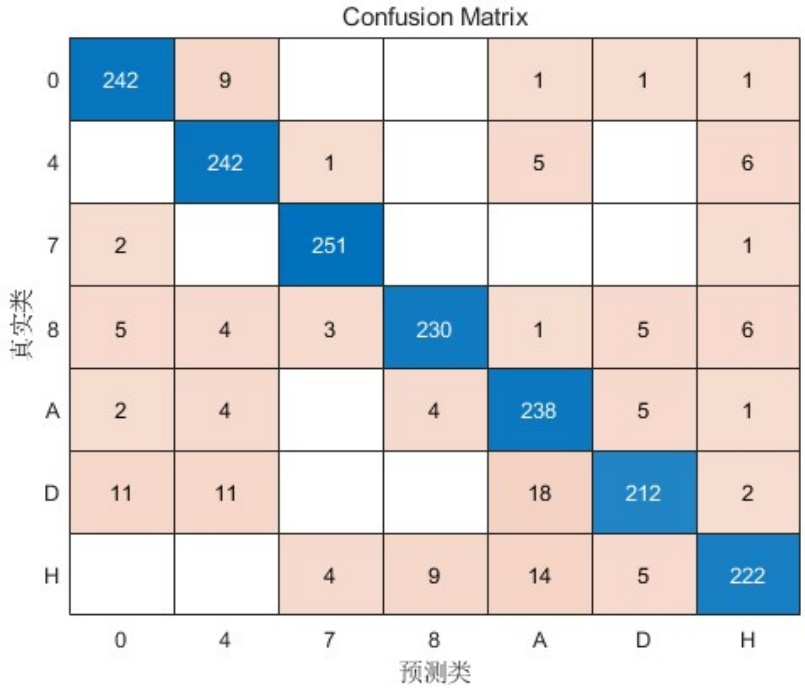

Figure 24: Confusion matrix

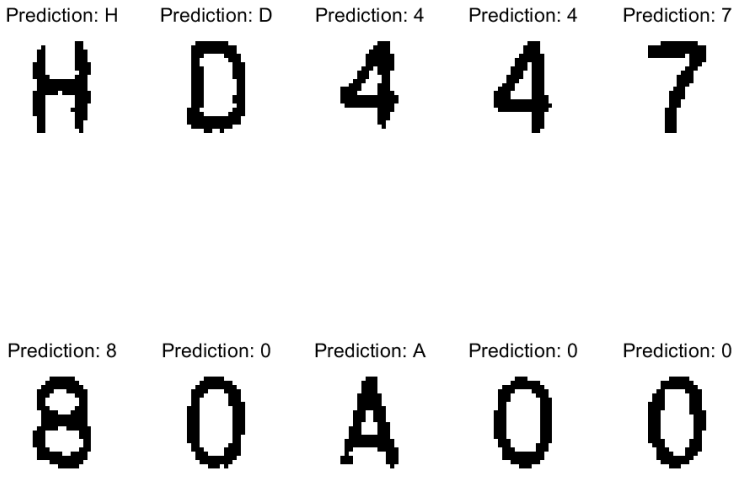

Figure 25: Character Recognition

- **Training Convergence**: Mean Squared Error during Training: The plot of the mean squared error (MSE) over training epochs illustrates a sharp decline in error as training progresses, leveling off as the model converges. This indicates that the model's learning process effectively minimizes the error, stabilizing as it reaches an optimal state. The relatively quick decrease in MSE suggests that the model's architecture and learning rate are well-suited to the dataset and the task.

  The analyzed CNN model displays excellent capability in recognizing and classifying various alphanumeric characters with high precision and accuracy. While it shows robust generalization across the classes tested, the slight misclassifications observed offer opportunities for further refinement of the model, possibly through enhanced preprocessing, data augmentation, or a more nuanced architectural tuning. The model's training dynamics also underscore its efficiency in learning from the data, making it a promising solution for tasks requiring high levels of accuracy in image classification.

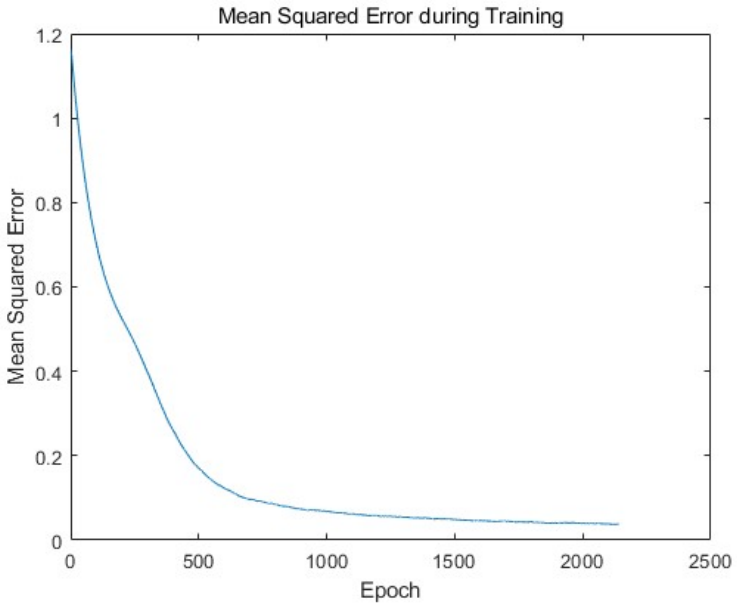

Figure 26: Mean Squared Error

- **Average Evaluation Results**:

```
Average Accuracy: 98.36
Average Precision: 94.43
Average Recall: 94.26
Average F1 Score: 94.25
Error rate: 5.74%
```

Figure 27: Average Evaluation Results

8.4 Task 8.2

8.4.1 Introduction

The goal of Task8_2 is to design a non-CNN-based classification system for character recognition and classification. In this section, we employed three different methods to build Support Vector Machine (SVM) models, which were then trained and tested for classification [16]. Various evaluation metrics were applied to assess the models' performance. Additionally, by fine-tuning the model parameters, we improved the classification accuracy, successfully classifying characters from the preprocessed character2 dataset.

8.4.2 Algorithms and Processing Steps

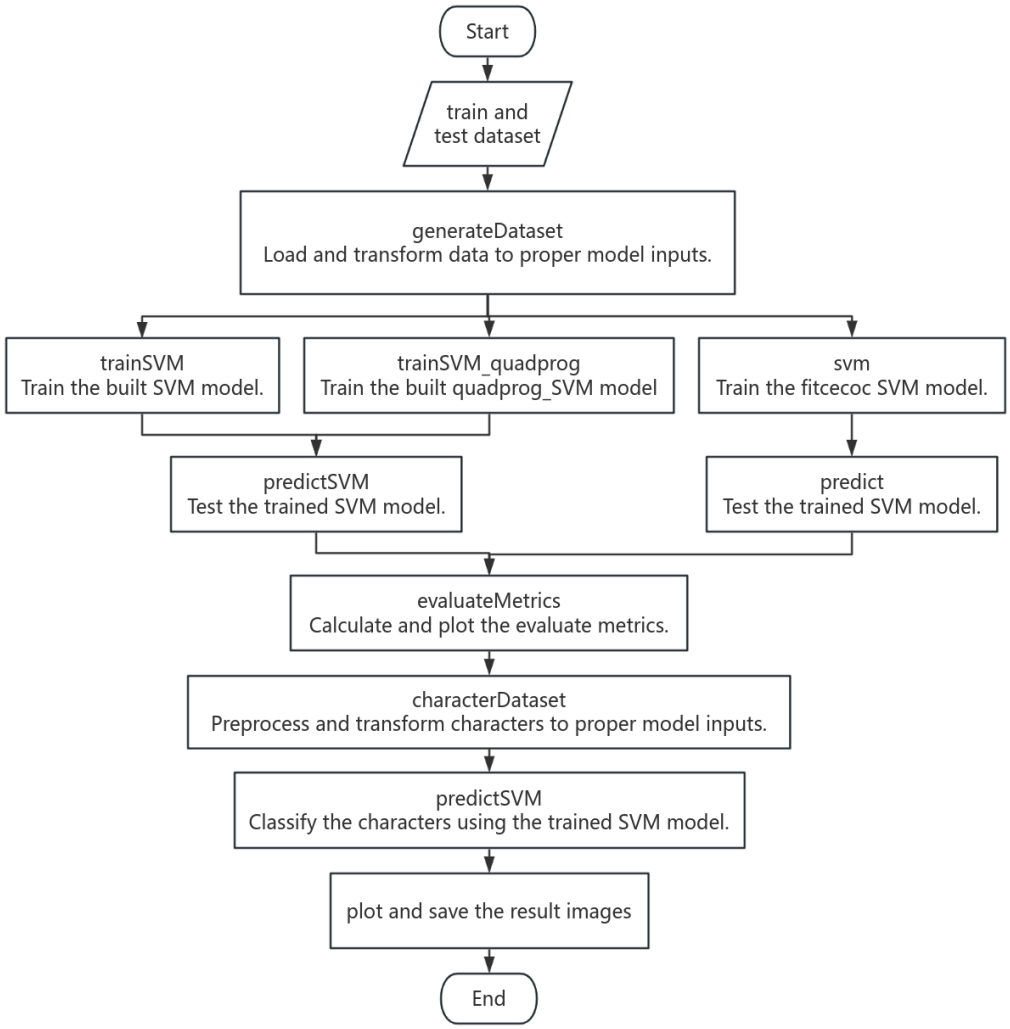

Figure 28: SVM framework

1. `GenerateDataset`:

   The generateDataset function is used to process the training and testing images, along with their labels, and convert them into a suitable format for model input. Initially, convert the images from RGB to grayscale using rgb2gray, which helps reduce computational complexity and focus on luminance information. Next, resize the images to a fixed resolution of 28x28 pixels using the imresize, ensuring consistency across the dataset and standardizing the input for the SVM model. The resized images are then flattened into one-dimensional feature vectors, with each image assigned a label that represents its character class. Finally, generate the dataset containing feature matrix (X) and label vector (Y), which provides structured input for training the SVM model.

2. `Train_SVM`:

   The trainSVM function is designed to train the SVM classifier. Since SVM is primarily used for binary classification tasks, we apply the One-vs-All strategy. Specifically, an SVM model is trained for each character class, treating the current class as positive and all other classes as negative. This approach enables each model to focus on recognizing a specific character class, and the results are subsequently combined to form a comprehensive classification system.

   The function takes the following input parameters: the feature matrix XTrain, the label vector YTrain, and the total number of classes numClasses. Each row of XTrain represents the features of a sample, while YTrain provides the corresponding labels. The function outputs are the weight vectors w and bias b for each class's SVM model. The process begins by setting the label of the target class to 1, while all other labels are set to -1. Firstly, compute the Gram matrix K = XTrain * XTrain', where each element K(i, j) represents the inner product of the i-th and j-th samples, capturing their similarity in feature space. Additionally, the similarity matrix H = (binaryY * binaryY') .* K is calculated, reflecting the relationship between the target class and the others. Then initialize the Lagrange multiplier vector alpha to zero. During each iteration, the error for each sample is computed as f_i = sum(alpha .* binaryY .* K(:, i)), and the gradient is used to update alpha with the rule alpha(i) = alpha(i) + C * (1 - binaryY(i) * f_i), ensuring that alpha remains within the bounds of [0, C]. The iteration continues until the change in alpha between successive rounds is below a threshold (1e-4), at which point the model is considered to have converged. Once the model converges, calculate the weight vector w = XTrain' * (alpha .* binaryY), representing the normal vector to the decision hyperplane. Furthermore, support vectors are identified by finding non-zero values in alpha, and the bias b = mean(binaryY(supportVectors) - XTrain(supportVectors, :) * w) is computed, representing the distance from the hyperplane to the origin. Finally, save the weight vector w and bias b for each class, making them available for use during the prediction phase.

3. `Predict_SVM`:

   The predictSVM function classifies new test data XTest using the trained SVM models and returns the predicted labels YPred. First, calculate the score for each sample with the formula scores(:, class) = XTest * w + b using the weight vector w and bias b from each model. This score represents the degree of match between the test samples and each class. Finally, select the class with the highest score as the final predicted label YPred.

4. `Evaluate_Metrics`:

   The evaluateMetrics function computes various evaluation metrics to assess models performance. Firstly, calculate the confusion matrix confMat = confusionmat(YTrue, YPred) to show the comparison between true labels and predicted labels of each class. The confusion matrix provides an intuitive view of the model's error distribution, helping to identify which classes are most prone to misclassification. Next, compute the accuracy, precision, recall, and F1 score for each class. Accuracy is the most commonly used metric, representing the proportion of correctly predicted samples. Precision measures the proportion of correctly predicted positive samples among those predicted as positive, while recall measures the proportion of actual positive samples correctly predicted. These two metrics are particularly important when dealing with imbalanced datasets. The F1 score is the harmonic mean of precision and recall, offering a comprehensive evaluation of the model's performance in terms of both accuracy and recall. Finally, plot the confusion matrix and histograms of the evaluation metrics to clearly observe the model's performance across all classes.

5. `CharacterDataset`:

   The characterDataset function is similar to generateDataset but processes the character2 data for classification. There are two main differences between this function and generateDataset. Firstly, there is no label processing here, as this step is part of the testing phase and does not require calculating accuracy or other evaluation metrics. Secondly, add padding to the images using img = padarray(img, [topBottomPadding, leftRightPadding], 255, 'both'). Since the character2 data differs from the training and testing images, this padding helps make the test data more compatible with the SVM model, ensuring it aligns with the format used in the training and testing phases.

6. `Main`:

   The main function is the core of this character classification system. It coordinates various steps, including dataset processing, SVM training, SVM prediction, and performance evaluation. Firstly, use the generateDataset function to load the training and test datasets, which will serve as inputs for the SVM model training. Next, train the SVM classifier using the trainSVM function and predict the test data using the trained model. The prediction results are then evaluated using the evaluateMetrics function, which calculates accuracy, precision, recall, and F1 score. Finally, process the character2 character images with the characterDataset function, classify the characters, and display the images.

7. `TrainSVM_quadprog`:

   The trainSVM_quadprog function offers an alternative approach to train the SVM model, using quadratic programming for optimization of the loss function. Compared to the iterative approach used in trainSVM, this method provides more precise optimization of the SVM model. The MATLAB quadprog function is used here alpha = quadprog(H, f, A, a, B, b), which is advantageous in terms of both accuracy and convergence speed. By setting inequality and equality constraints (A, a, B, b), we ensure that the alpha values meet the requirements of non-negativity and that the sum of labels equals zero. Then the weight vector w and bias b are computed, resulting in a more accurate SVM model.

8. `svm`:

   This file differs from the previous implementations by directly using MATLAB built-in fitcecoc function to construct a multi-class SVM classifier with a kernel function [17]. The classification is performed with the following setup: t = templateSVM('KernelFunction', 'gaussian', 'KernelScale', 'auto', 'BoxConstraint', 1); SVMModel = fitcecoc(XTrain, YTrain, 'Learners', t). The fitcecoc function automatically decomposes the multi-class classification task into multiple binary classification problems, eliminating the need to manually build several binary models. The classifier is then applied to the test dataset with YPred = predict(SVMModel, XTest) to generate classified labels for each character. This approach simplifies multi-class prediction implementation while maintaining high accuracy. For performance evaluation, we also use evaluate_Metrics function, allowing us to compare the performance of this method with others.

8.4.3 Results Analysis and Comparison

In this report, we applied three different methods to classify characters and compared their performance. These methods include: the SVM model based on the trainSVM function, the SVM model based on the trainSVM_quadprog function, and the SVM model directly using MATLAB's built-in library function. For each method, we evaluate the results through the confusion matrix, histograms of accuracy, precision, recall, and F1 score, average classification performance, and the classification results for the target character set. The following sections analyze the results of each method in detail.

1. **SVM Model Based on the `trainSVM` Function**:

   The model trained using the custom trainSVM function iteratively optimizes the SVM decision boundary. The results for this method are as follows:

   Confusion Matrix: The confusion matrix for the test set is shown below. It provides a clear visualization of how each character category is classified and the distribution of misclassifications.

   Evaluation Parameter Histograms: The histograms in Figure 30 of precision, recall, and F1 score below show the performance across different categories. Some categories demonstrate high precision, but relatively lower recall or F1 scores, indicating a certain bias in the model towards specific categories.

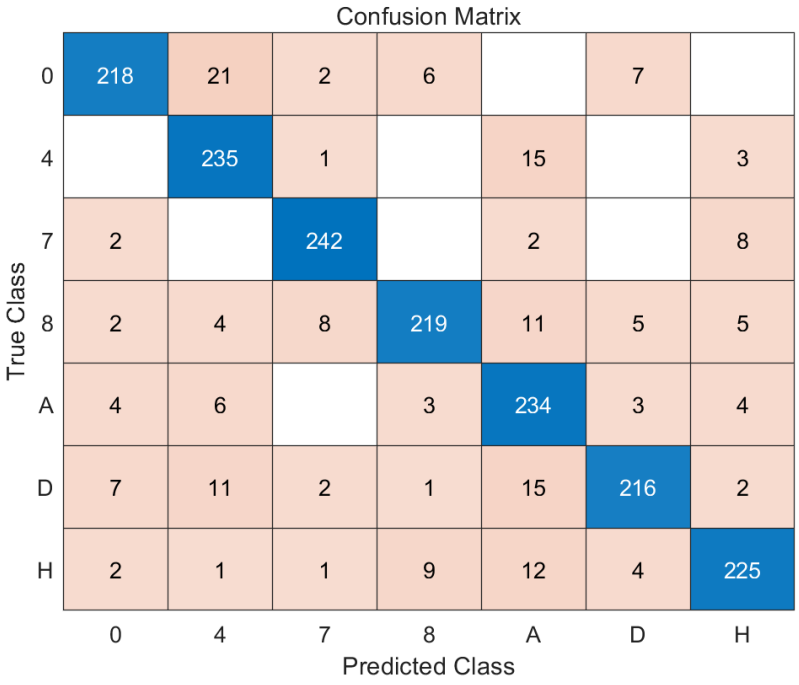

Figure 29: Confusion metrix

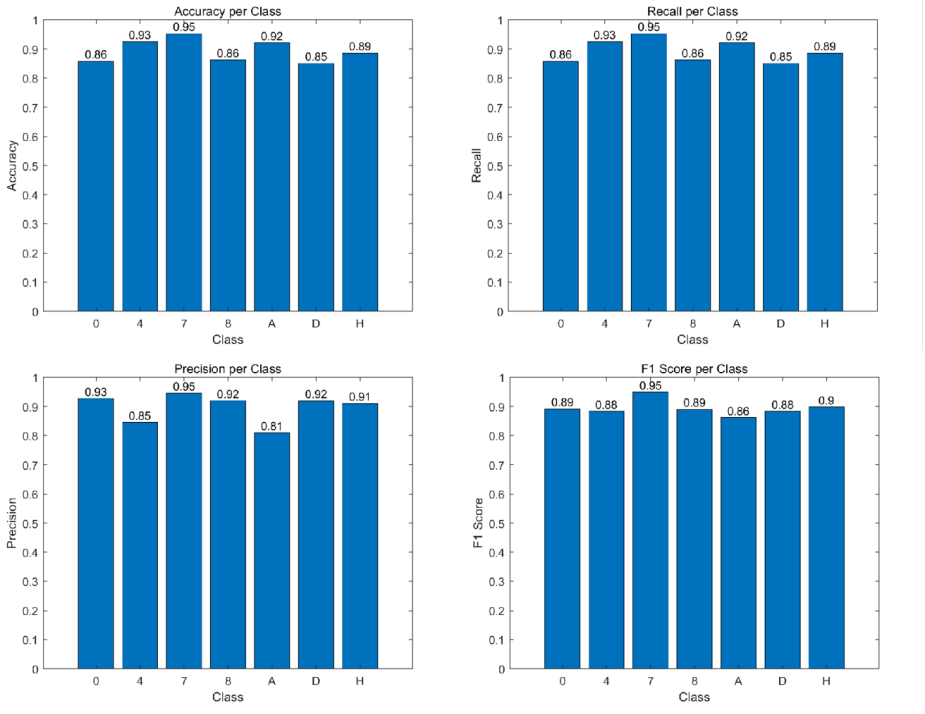

Figure 30: Performance metrix

Average Evaluation Results: The average precision, recall, and F1 score for this model are provided in Figure 31. While the method effectively distinguishes most character categories, its performance is not as strong as other approaches due to the simplified optimization process.

```
>> main
Average Accuracy: 0.89
Average Precision: 0.90
Average Recall: 0.89
Average F1 Score: 0.89
```

Figure 31: Average evaluation result

Classification Results: The images below display the model's classification results for 10 target characters, comparing the predicted labels with the actual ones. These results show how the model performs in practical character recognition. The classification is divided into two scenarios: with and without padding, highlighting the sensitivity of the SVM model to data variations.

- **No padding**:

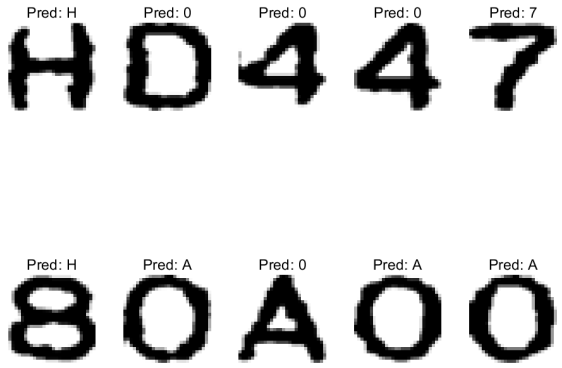

Figure 32: Classification results without padding

- **Padding**:

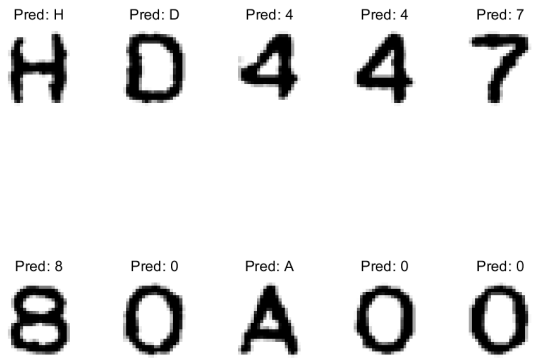

Figure 33: Classification results with padding

2. **SVM Model Based on the `trainSVM_quadprog` Function**:

   By using the quadprog function for quadratic programming optimization, this method achieves more accurate classification by finding the optimal solution. The results are as follows:

   Confusion Matrix: The confusion matrix below shows that the model based on quadprog offers better classification performance, with a noticeable reduction in misclassifications for some categories.

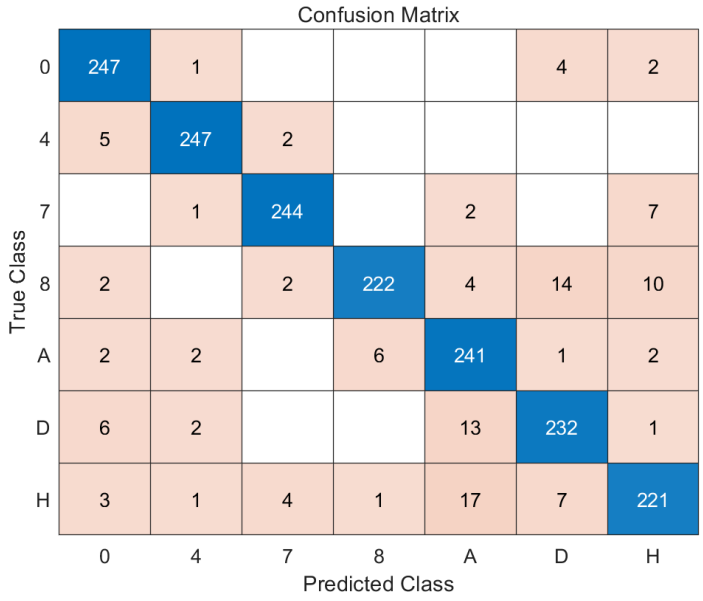

Figure 34: Confusion metrix

Evaluation Parameter Histograms: The histograms of precision, recall, and F1 score show a more balanced performance across categories, reducing classification bias compared to the previous method.

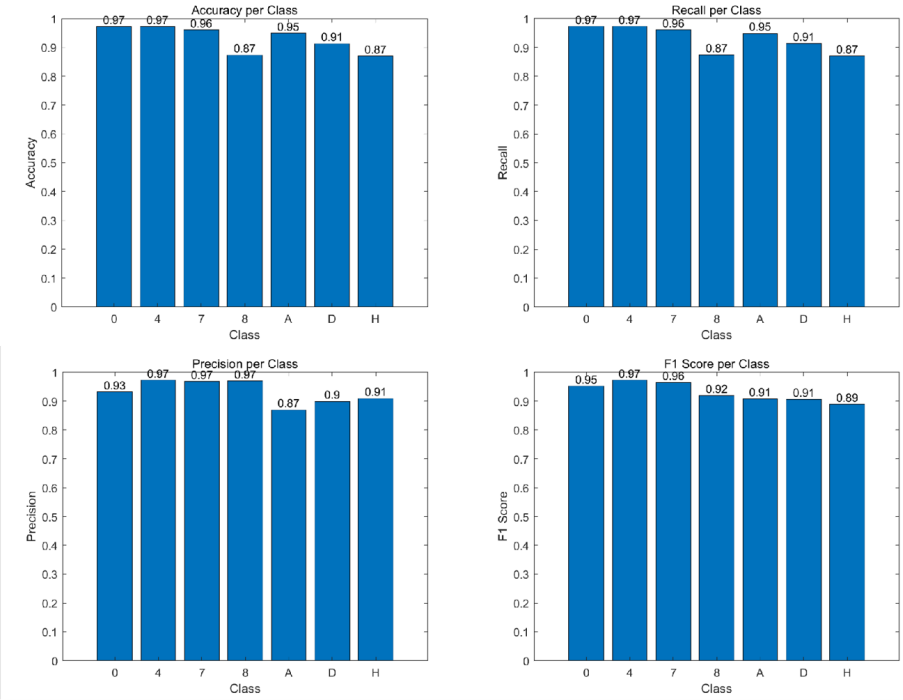

Figure 35: Performance metrix

Average Evaluation Results: The average precision, recall, and F1 score for this method surpass those of the trainSVM model, reflecting the improvement brought by quadratic programming optimization.

```
>> main
Average Accuracy: 0.93
Average Precision: 0.93
Average Recall: 0.93
Average F1 Score: 0.93
```

Figure 36: Average evaluation result

Classification Results:

- **No padding**:

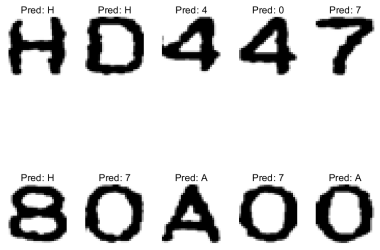

Figure 37: Classification results without padding

- **Padding**:

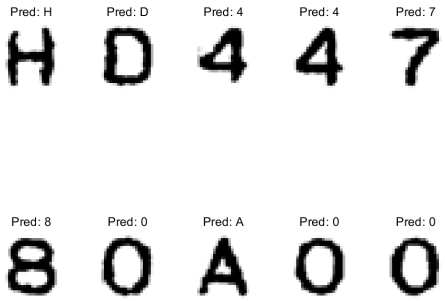

Figure 38: Classification results with padding

3. **SVM Model Using MATLAB's Built-In Library Function**:

   The SVM model constructed using the MATLAB `fitcecoc` function leverages efficient library optimizations to perform multi-class classification directly. The results for this method are as follows:

   Confusion Matrix: The confusion matrix for the model trained using the built-in library function demonstrates high classification accuracy across most categories.

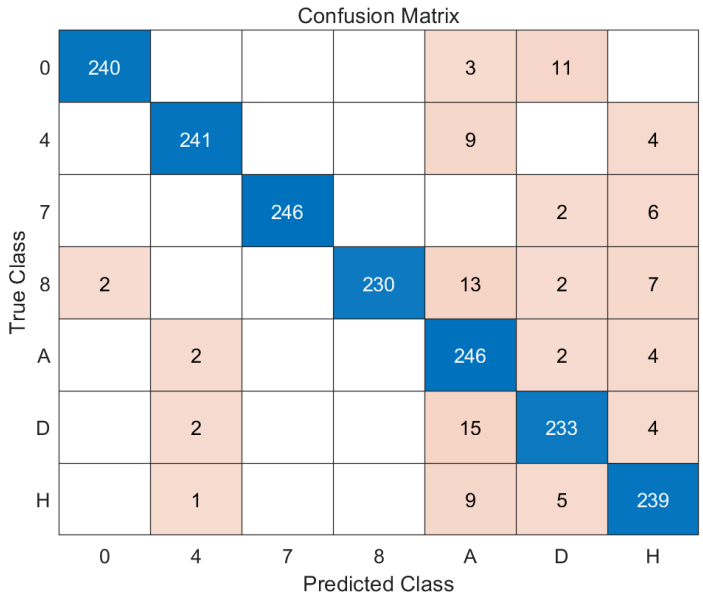

Figure 39: Confusion metrix

Evaluation Parameter Histograms: The histograms of precision, recall, and F1 score further confirm the model's superior performance, showing stability and accuracy across all categories.

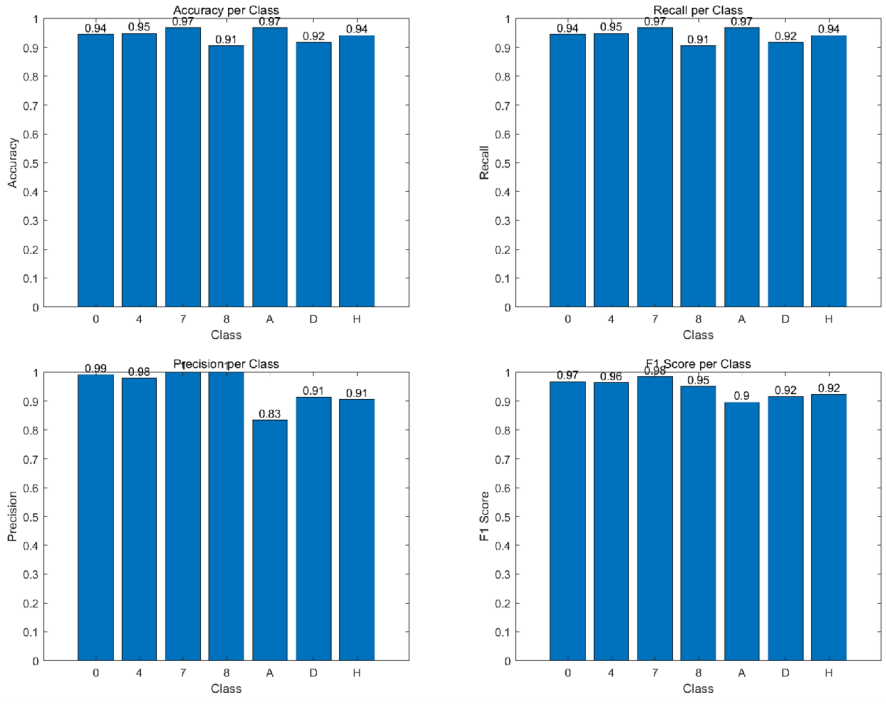

Figure 40: Performance metrix

Average Evaluation Results: The average precision, recall, and F1 score achieved by this method are the highest among all three approaches, indicating the stability and efficiency of the library function method.

```
>> svm
Average Accuracy: 0.94
Average Precision: 0.95
Average Recall: 0.94
Average F1 Score: 0.94
```

Figure 41: Average evaluation result

Classification Results:

- **No padding**:

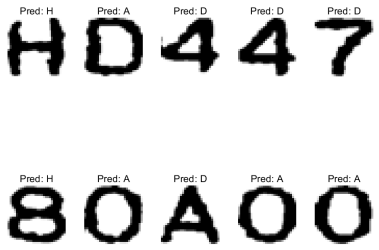

Figure 42: Classification results without padding

- **Padding**:

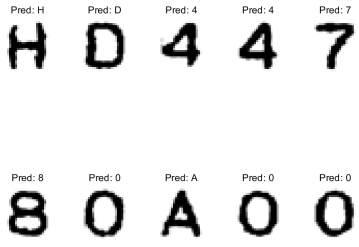

Figure 43: Classification results with padding

8.4.4 Conclusion

Through the comparison and analysis of the three methods, it is evident that the SVM model using the fitcsvm library function performs the best in terms of classification performance and efficiency. Its average evaluation metrics are significantly higher than those of the custom training methods. However, both trainSVM and trainSVM_quadprog methods, which involve manually building the SVM model, also achieved good results.

Table 1: Comparison of SVM Model Performances

Model	Accuracy	Precision	Recall	F1 Score
TrainSVM	0.89	0.90	0.89	0.89
TrainSVM_quadprog	0.93	0.93	0.93	0.93
Fitcecoc_svm	0.94	0.95	0.94	0.94

To further improve the generalization capability of the SVM classification model for characters of different sizes and positions, we applied padding to the input images. Prior to padding, the images to be classified had significant differences from the training and testing images, which could negatively affect the SVM model's classification performance due to disparities in pixel distribution. After experimenting with different padding methods around the image edges, and adjusting the dataset to a consistent 28x28 pixel size, we observed the following changes:

- **Before Padding**:

  When the image size was adjusted without padding, the model struggled to differentiate similar structured characters (e.g., ”D” and ”0”). This inconsistency caused higher misclassification rates, particularly for characters with minimal structural differences, negatively impacting classification accuracy.

- **After Padding**:

  By padding the image edges to match the shape and size of the training and testing data, the classification accuracy improved significantly. The SVM model was better able to capture the key features of each character category.

  The SVM model proved highly sensitive to data processing, as its core principle is to find an optimal hyperplane that maximizes the margin between different category samples. During training, SVM calculates the support vectors and their classification boundaries based on the feature distribution in the input data, making the model highly dependent on the data's features and distribution. If there are large variations in feature scale or distribution, the model's performance will fluctuate. For example, when images are not padded or standardized, characters might differ significantly in shape or size, leading to difficulties in distinguishing similar categories.

In addition to image preprocessing, hyperparameter tuning further improved the SVM model's performance. The regularization parameter C of SVM controls the balance between the classification boundary and error tolerance. A higher C value penalizes errors more, making the model more strict in classifying categories and improving accuracy. However, it also makes the model

more sensitive to outliers and may lead to overfitting. Therefore, we carefully tuned the C value to ensure a balanced performance in both training and testing.
In conclusion, we successfully implemented three SVM models based on the One-vs-All strategy for multi-class character classification, achieving high classification accuracy. We also recognized the SVM model's high sensitivity to data characteristics, highlighting the importance of proper hyperparameter tuning and data preprocessing to optimize performance.

8.5 Conclusion

In Task 8, the focus was on experimenting with data preprocessing techniques such as padding or resizing the input images, along with hyperparameter tuning. The results underscored the importance of these preprocessing steps and parameter adjustments for achieving optimal performance in character classification tasks.
The comparative analysis between the CNN and the SVM highlighted distinct advantages and limitations:

- **CNN**:

  Advantages:

  CNNs demonstrated robust performance in handling image data due to their ability to automatically detect and prioritize the most informative features through multiple layers of processing. This automated feature extraction minimizes the need for manual feature engineering and is particularly effective for image classification tasks where spatial hierarchies are crucial.

  Sensitivity:

  CNNs were found to be highly sensitive to the quality and nature of input data preprocessing. Proper image resizing and normalization significantly improved the performance by ensuring that the network could learn relevant patterns effectively.

  Hyperparameter Tuning:

  The adjustment of network architecture details, such as the number of layers, layer sizes, and learning rates, played a critical role in optimizing the model's performance. The results indicated that CNNs could be finely tuned to enhance classification accuracy and adapt to new or varied datasets effectively.

- **SVM**:

  Advantages:

  SVMs excelled in classification tasks with a clear margin of separation between classes. They were particularly effective when used with a kernel that transforms the input data into a high-dimensional space, making it easier to find a linear separation.

  Limitations:

  SVMs require careful selection of the kernel type and tuning of parameters like the regularization parameter (C) and the kernel parameters. These choices significantly influence the model's ability to generalize and perform well on unseen data.

Sensitivity to Hyperparameters:

SVMs demonstrated a high sensitivity to hyperparameter settings. Optimal performance was contingent on the correct setting of these parameters, which required extensive experimentation and validation.

To sum up, both CNNs and SVMs were sensitive to how the data was preprocessed. Padding, resizing, and normalizing the input images were crucial steps that significantly impacted both models' ability to learn and generalize.
Hyperparameter tuning emerged as a vital process for both models. For CNNs, architectural parameters and learning rates needed careful adjustment, while for SVMs, the choice of kernel and regularization parameters was critical.
The experimental findings also emphasized the need for a balanced approach that considers both the characteristics of the data and the specific strengths and weaknesses of each model to achieve optimal performance.
These insights from Task 8 provide a comprehensive understanding of how different machine learning models respond to variations in data preprocessing and hyperparameter settings, offering valuable guidelines for future projects involving image classification tasks.
In consideration of the constraints imposed by the limited space available within the main text of this document, a comprehensive comparison of additional parameters relevant to the CNN has been meticulously compiled and stored separately. For a detailed examination of these comparative results, which include an array of diverse configurations and their respective impacts on the CNN's performance, readers are directed to the accompanying file repository. This repository, located within the **"Output/Task8_1_cnn_output/cnn_output"** folder, contains structured data and analytical commentary that elucidate the variations observed under different parameter settings.

References

[1] MathWorks, **imadjust**, https://www.mathworks.com/help/images/ref/imadjust.html?s_tid=doc_ta, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/imadjust.html?s_tid=doc_ta pages 5

[2] ——, **histeq**, https://www.mathworks.com/help/images/ref/histeq.html?s_tid=doc_ta#d126e132956, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/histeq.html?s_tid=doc_ta#d126e132956 pages 5

[3] ——, **adapthisteq**, https://www.mathworks.com/help/images/ref/adapthisteq.html?s_tid=doc_ta, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/adapthisteq.html?s_tid=doc_ta pages 5

[4] ——, **imfilter**, https://www.mathworks.com/help/images/ref/imfilter.html, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/imfilter.html pages 10

[5] ——, **mvnpdf**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/stats/mvnpdf.html?searchHighlight=mvnpdf&s_tid=srchtitle_support_results_1_mvnpdf pages 11

[6] ——, **fft2**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/matlab/ref/fft2.html?s_tid=doc_ta pages 13

[7] ——, **fftshift**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/matlab/ref/fftshift.html?s_tid=doc_ta pages 13

[8] ——, **ifftshift**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/matlab/ref/ifftshift.html?s_tid=doc_ta pages 14

[9] ——, **ifft2**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/matlab/ref/ifft2.html?s_tid=doc_ta pages 14

[10] ——, **imbinarize**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/imbinarize.html?searchHighlight=imbinarize&s_tid=srchtitle_support_results_1_imbinarize pages 18

[11] N. Otsu, "A threshold selection method from gray-level histograms," **IEEE Transactions on Systems, Man, and Cybernetics**, vol. 9, no. 1, pp. 62–66, 1979. pages 18

[12] MathWorks, **imopen**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/imopen.html?searchHighlight=imopen&s_tid=srchtitle_support_results_1_imopen pages 21

[13] ——, **imclose**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/imclose.html?searchHighlight=imclose&s_tid=srchtitle_support_results_1_imclose pages 22

[14] ——, **edge**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/images/ref/edge.html?searchHighlight=edge&s_tid=srchtitle_support_results_1_edge pages 22

[15] “Deep learning in MATLAB,” MathWorks - Documentation Center, available at https://www.mathworks.com/help/deeplearning/ug/deep-learning-in-matlab.html. pages 27

[16] “Support vector machines in MATLAB,” MathWorks - Documentation Center, available at https://www.mathworks.com/help/stats/support-vector-machines-for-binary-classification.html. pages 36

[17] MathWorks, **fitcecoc**, n.d., mATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/stats/fitcecoc.html pages 39

A Appendices

Otsu's method

The derivation process of the Otsu's method used in the code is provided below:

$$\sigma_B^2 = w_1(\mu_1 - \mu_T)^2 + w_2(\mu_2 - \mu_T)^2$$

& $w_2 = 1 - w_1$

& $\mu_T = w_1\mu_1 + w_2\mu_2$

Known w_1 , μ_1 , μ_T

$$\therefore \sigma_B^2 = w_1(\mu_1 - w_1\mu_1 - w_2\mu_2)^2 + w_2(\mu_2 - w_1\mu_1 - w_2\mu_2)^2$$

$$= w_1(\mu_1\cdot(1-w_1) - w_2\mu_2)^2 + w_2(\mu_2\cdot(1-w_2) - w_1\mu_1)^2$$

$$= w_1(w_2\mu_1 - w_2\mu_2)^2 + w_2(w_1\mu_2 - w_1\mu_1)^2$$

$$= w_1w_2(\mu_1-\mu_2)^2 + w_1w_2(\mu_2-\mu_1)^2$$

Since $(\mu_1-\mu_2)^2 = (\mu_2-\mu_1)^2$ and both of them already account for the differences between the two classes, there is no need to add factor of 2 in this overall variance.

$$= w_1w_2(\mu_1-\mu_2)^2$$

$$= \frac{(w_1w_2\mu_1 - w_1w_2\mu_2)^2}{w_1\cdot w_2}$$

$$= \frac{(w_1w_2\mu_1 - w_1\mu_T + w_1^2\mu_1)^2}{w_1\cdot w_2}$$

$$= \frac{(\mu_1(w_1(1-w_1)+w_1^2) - w_1\mu_T)^2}{w_1\cdot w_2}$$

$$= \frac{(w_1\mu_1 - w_1\mu_T)^2}{w_1\cdot w_2}$$

Figure 44: The derivation process of the Otsu's method