ME5411 Robot Vision and AI

CA – AY24/25 Semester 1 Computing Project Report

Group 4

Full Name	Student ID
Cheng Yuchao	A0304546H
Pan Yongzhou	A0285107E
Xiao Yuxiang	A0304608J

Master of Science in Robotics

COLLEGE OF DESIGN AND ENGINEERING

Nov 18th, 2024

CONTENTS

***Abstract*—This report documents the development and outcomes of tasks in robot vision and artificial intelligence for ME5411 Robot Vision and AI. The robot vision tasks focus on implementing image processing methods, including contrast enhancement, filtering, edge detection, and segmentation, with detailed explanations and pseudo-code provided. For AI part, two character classification methods—Convolutional Neural Networks (CNN) and Multi-class Support Vector Machines (SVM)—are compared in terms of performance and effectiveness. Experimental results highlight the superiority of CNN in accuracy and robustness, supported by data pre-processing and hyperparameter tuning to optimize performance. The findings are critically analyzed, reflecting on the methodologies and future improvement directions.The source code is available at https://github.com/YongzhouPan/ME5411.**

I. Project Overview

This project addresses tasks related to robot vision (Tasks 1–7) and artificial intelligence (Tasks 8–9). The team collaborated effectively to complete the assignment, with responsibilities divided as shown in Table I.

Task Area	Contributor	Contribution
Image Processing	Pan Yongzhou	Implemented image processing methods.
Character Classification	Xiao Yuxiang	Developed and tested CNN.
Character Classification	Cheng Yuchao	Implemented and tested multi-class SVM.
Performance Exploration	Team	Compared performance of CNN and SVM, exploring algorithm performance optimization.

TABLE I: Team Contributions Summary

Image Processing: Pan Yongzhou led the implementation and refinement of image processing tasks, including contrast enhancement, filtering, binarization, edge detection, and segmentation, with most methods developed from scratch without relying on MATLAB's built-in libraries or toolboxes, providing a solid foundation for the robot vision component of the project. **Detailed usage instructions for all custom functions are provided; simply run `help func_name` to view them.**

Character Classification: Xiao Yuxiang developed and tested a Convolutional Neural Network (CNN) for character classification, focusing on model architecture and optimization techniques. Cheng Yuchao implemented and tested a Multi-class Support Vector Machine (SVM), emphasizing feature extraction and training efficiency.

Performance Exploration: The team collaboratively evaluated and compared the performance of CNN and SVM. This included experiments with data pre-processing techniques such as resizing and padding, as well as hyperparameter tuning to optimize performance.

The entire team worked together to compile this report. For implementation details, please refer to the source code included in the submission package or visit the open-source repository.

II. Task 1. Image Display and Contrast Enhancement

A. Task Overview

and converted to grayscale, ensuring a consistent format for subsequent processing. Several contrast enhancement techniques were applied to improve the visibility of image details. Observations and comments on the results are provided to analyze the effectiveness of each method.

B. Algorithm Description

The task began with preprocessing the input image to ensure it was in grayscale. The data image *charact2.bmp* was identified as a color image and subsequently converted to an 8-bit grayscale format for further processing.

Contrast enhancement techniques were then applied to adjust the intensity values of the grayscale image, improving overall contrast and emphasizing specific features. The following methods were implemented:

- **Contrast Stretching**: Enhanced contrast by expanding the range of intensity values through linear or piecewise linear transformations.
- **Brightness Thresholding**: Adjusted brightness by applying intensity thresholds to isolate specific regions, utilizing Contrast Stretching functions for implementation.
- **Gray Level Slicing**: Highlighted a specific range of intensity values while suppressing others to emphasize key features.
- **Dynamic Range Compression**: Used logarithmic transformations to compress the dynamic range, making low-intensity details more visible.
- **Pseudocolor Image Processing**: Mapped grayscale intensity values to a range of colors to enhance visual interpretation.
- **Histogram Equalization**: Redistributed intensity values to achieve a uniform histogram, improving global contrast.

The details of each algorithm are provided as follows:

1) Contrast Stretching

A *contrastStretching* function was developed to support both simple linear and piecewise linear transformations. Piecewise linear contrast stretching extends the intensity range by mapping input values linearly between specified breakpoints, enhancing contrast in targeted regions of the image and making specific features more obvious. The continuous formation of the piecewise linear functions is defined as:

$$s(r) = \begin{cases} \frac{s_1}{r_1} \cdot r, & 0 \le r \le r_1 \\ s_1 + \frac{s_2 - s_1}{r_2 - r_1} \cdot (r - r_1), & r_1 < r \le r_2 \\ s_2 + \frac{255 - s_2}{255 - r_2} \cdot (r - r_2), & r_2 < r \le 255 \end{cases} \quad (1)$$

where:

- r is the input intensity value, and $s(r)$ is the output intensity value.
- r_1 and r_2 are the breakpoints in the input intensity range.

- s_1 and s_2 are the corresponding output intensities at r_1 and r_2, respectively.

This piecewise function maps intensity values from three regions listed in Table II.

Range of r (Input Intensity)	**Mapping of s (Output Intensity)**
$0 \leq r \leq r_1$	Linearly scaled between $[0, s_1]$
$r_1 < r \leq r_2$	Linearly stretched between $[s_1, s_2]$
$r_2 < r \leq 255$	Linearly scaled between $[s_2, 255]$

TABLE II: Piecewise Mapping of Intensity Values

This transformation ensures a mapping across the entire intensity range, enhancing the contrast of targeted regions.

2) Brightness Thresholding

Brightness thresholding was implemented to segment the image by isolating specific intensity ranges. The transformation function assigns output intensity values based on threshold conditions. In our implementation, the functionality was achieved by reusing the `contractStretching` function, where the threshold value was set by making the two breakpoints equal ($r_1 = r_2$). Specifically, the following parameters were used:

- $r_1 = r_2 = 136$: This sets the intensity threshold to 136.
- $s_1 = 0$: Pixels with intensity values less than or equal to r_1 are mapped to 0.
- $s_2 = 255$: Pixels with intensity values greater than r_2 are mapped to 255.

Mathematically, the transformation is expressed as:

$$s = T(r) = \begin{cases} 0, & r \leq r_1 \\ L-1, & r > r_2 \end{cases} \tag{2}$$

where:

- r is the input intensity value, and s is the output intensity value after thresholding.
- r_1 and r_2 are the lower and upper intensity thresholds, respectively.
- $L-1$ is the maximum intensity value (e.g., 255 for an 8-bit image).

This approach leverages the flexibility of the developed *contractStretching* function to perform brightness thresholding by setting the threshold intensity value and mapping all pixels below or equal to the threshold to black and all pixels above the threshold to white.

3) Gray Level Slicing

Gray level slicing was implemented as the *grayLevelSlicing* function to enhance a specific intensity range $[A, B]$ in the input image, either by preserving or diminishing the intensity values outside this range. Two modes were supported in our implementation:

- **Preserve**: The intensity values outside the range $[A, B]$ remain unchanged, retaining their original grayscale levels.
- **Diminish**: The intensity values outside the range $[A, B]$ are set to zero (black), effectively suppressing them.

The transformation function for each mode is expressed mathematically as follows:

For **Preserve Mode**:

$$s(r) = \begin{cases} \text{intensity}, & A \leq r \leq B \\ r, & r < A \text{ or } r > B \end{cases} \tag{3}$$

For **Diminish Mode**:

$$s(r) = \begin{cases} \text{intensity}, & A \leq r \leq B \\ 0, & r < A \text{ or } r > B \end{cases} \tag{4}$$

where:

- r is the input intensity value, and $s(r)$ is the output intensity value.
- A and B are the lower and upper boundaries of the intensity range to be enhanced.
- intensity is the output intensity value assigned to the range $[A, B]$ (default: 255).

In **Preserve**, intensity values outside the specified range $[A, B]$ are left unchanged, maintaining their original grayscale levels. Conversely, in **Diminish**, intensity values outside $[A, B]$ are set to zero, effectively suppressing them.

This method highlights the specified intensity range, enhancing the visibility of features within $[A, B]$ while offering flexibility in how the remaining intensity values are treated.

4) Dynamic Range Compression

Dynamic range compression was implemented to compress the dynamic range of a grayscale image using a logarithmic transformation. The method is defined as:

$$T(r) = c \cdot \log(1 + r) \tag{5}$$

where:

- r is the input pixel intensity value.
- $T(r)$ is the transformed output intensity value.
- c is a scaling constant, defined as $c = \frac{255}{\log(1+R)}$, where R is the maximum intensity value of the input image.

In our implementation, the maximum pixel value R is determined based on the input image type (uint8, uint16, or float). The logarithmic transformation maps the input values nonlinearly, enhancing lower intensity values and compressing higher ones. The final output is scaled back to the 8-bit range $[0, 255]$.

5) Pseudocolor Image Processing

Pseudocolor processing was implemented to enhance visual interpretation by mapping grayscale intensity values to distinct colors. The function segments the image into two visually distinct regions by assigning blue and yellow to pixels based on a specified threshold.

The pseudocolor mapping in our implementation is defined as:

$$T(r) = \begin{cases} \text{blue } [0, 0, 255], & r \leq \text{threshold} \\ \text{yellow } [255, 255, 0], & r > \text{threshold} \end{cases} \tag{6}$$

where:

- r is the input intensity value from the grayscale image.
- $T(r)$ is the RGB color assigned to each pixel.

The resulting pseudocolor image enhances contrast and segmentation, making it easier to distinguish regions of interest in the grayscale image.

6) Histogram Equalization

Histogram equalization was implemented to enhance the contrast of a grayscale image by redistributing its pixel intensity values. This method improves the image by spreading the histogram more evenly across the entire intensity range. In our implementation, the output pixel values are mapped to a specified range $[q_0, q_k]$. The pseudo-code for the algorithm is shown in Algorithm 1.

Algorithm 1: Histogram Equalization

Input: Grayscale image I, output range $[q_0, q_k]$
Output: Equalized image I_{eq}

Compute histogram $H(p)$ for $p = 0, 1, \ldots, 255$;
Normalize to get the PDF:

$$\text{PDF}(p) = \frac{H(p)}{N \times M}$$

Compute the CDF:

$$\text{CDF}(p) = \sum_{i=0}^{p} \text{PDF}(i)$$

foreach *pixel p in I* **do**
    Compute equalized pixel value:

$$q(p) = q_0 + (q_k - q_0) \cdot \text{CDF}(p)$$

    Replace original pixel value with $q(p)$;
return I_{eq}

C. Experiment Results

In this section, we present the results of applying contrast enhancement methods to a grayscale image derived from the original color image.The original image is shown in Figure 1, followed by the enhanced image after applying the contrast enhancement techniques, as shown in Figure 2.

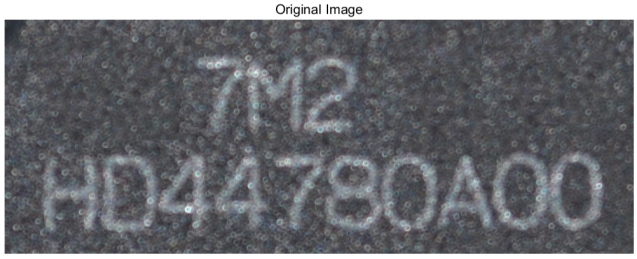

Fig. 1: The original grayscale image.

A series of contrast enhancement methods are applied to the grayscale image from the original. We compare the original image with the enhanced version to evaluate the impact of the applied techniques.

This comparison highlights how the methods improve the image's contrast, making details more visible in both dark and bright regions.

D. Comparison and Discussion

The original image is in color, but contrast enhancement was applied primarily to the grayscale version. Therefore, the first step in processing was to convert the image to grayscale.

Among the contrast enhancement methods, histogram equalization produced the most noticeable effect. It significantly improved the visibility of details, especially by enhancing the boundaries in the image, making them much clearer.

While **Dynamic Range Compression** is effective for images with large intensity ranges (e.g., images captured in extreme lighting conditions), it is **not suitable for this task**, as the given input images already have a limited and well-distributed intensity range. Applying this technique to the current dataset does not produce significant improvements and may even result in the loss of some contrast details in the processed images.

III. Task 2. Averaging Filter

Algorithm 2: Averaging Filter

Input: img_raw (Input grayscale or RGB image), filter_size (Size of square filter)
Output: img_filtered (Filtered image)

filter_mask ← Create matrix of size filter_size × filter_size, each element ← $\frac{1}{\text{filter\_size}^2}$;
if *size(img_raw, 3) == 1 (Grayscale image)* **then**
    img_filtered ← Initialize empty image with same size as img_raw;
    foreach *pixel (i, j) in img_raw* **do**
        Avg ← Mean of img_raw($i - \lfloor \text{filter\_size}/2 \rfloor : i + \lfloor \text{filter\_size}/2 \rfloor, j - \lfloor \text{filter\_size}/2 \rfloor : j + \lfloor \text{filter\_size}/2 \rfloor$);
        img_filtered(i, j) ← Avg;
    end
end
else if *size(img_raw, 3) == 3 (RGB image)* **then**
    img_filtered ← Initialize empty image with same size as img_raw;
    foreach *channel in {R, G, B}* **do**
        foreach *pixel (i, j) in img_raw[channel]* **do**
            Avg ← Mean of img_raw[channel]($i - \lfloor \text{filter\_size}/2 \rfloor : i + \lfloor \text{filter\_size}/2 \rfloor, j - \lfloor \text{filter\_size}/2 \rfloor : j + \lfloor \text{filter\_size}/2 \rfloor$);
            img_filtered[channel](i, j) ← Avg;
        end
    end
end
return img_filtered;

Fig. 2: Comparison of contrast enhancement.

A. Task Overview

In this task, a 5x5 averaging filter was implemented and applied to the image, producing a blurred image with reduced noise. Filters of different sizes were also tested to observe their effects and compare the differences in image smoothing results. A final comparison and commentary on the performance of the respective image smoothing methods are provided.

B. Algorithm Description

The averaging filter smooths the input image by reducing noise while preserving its essential structure. The algorithm is designed to handle both grayscale and RGB images, applying the filter to each channel separately for RGB images.

The algorithm works by calculating the average intensity of each pixel's neighborhood within a square window defined by the filter size. For grayscale images, this involves averaging the intensities of the surrounding pixels in the neighborhood, and for RGB images, the same operation is performed independently for each color channel.

The pseudocode for the averaging filter algorithm is shown in Algorithm 2. It outlines the steps for applying the filter to both grayscale and RGB images.

C. Experiment Results

In this section, we present the results of applying the averaging filter with different filter sizes: 3x3, 5x5, and 7x7. The images show how the blurring effect increases with larger filter sizes. The 7x7 filter produced the most significant blurring, while the 3x3 filter preserved relatively more details.

D. Comparison and Discussion

For the current filter sizes, the averaging filter produces only slight changes in the image. However, it is important to note that MATLAB provides two different functions for convolution: `conv2` and `imfilter`. Below is a comparison of their key differences:

Feature	`conv2`	`imfilter`
Boundary Handling	Zero padding only	Flexible (zero, symmetric, replicate)
Output Size	Controlled by mode (`'same'`, `'full'`, `'valid'`)	Matches input size by default
Use Case	General 2D convolution	Optimized for image filtering tasks

TABLE III: Comparison of `conv2` and `imfilter`

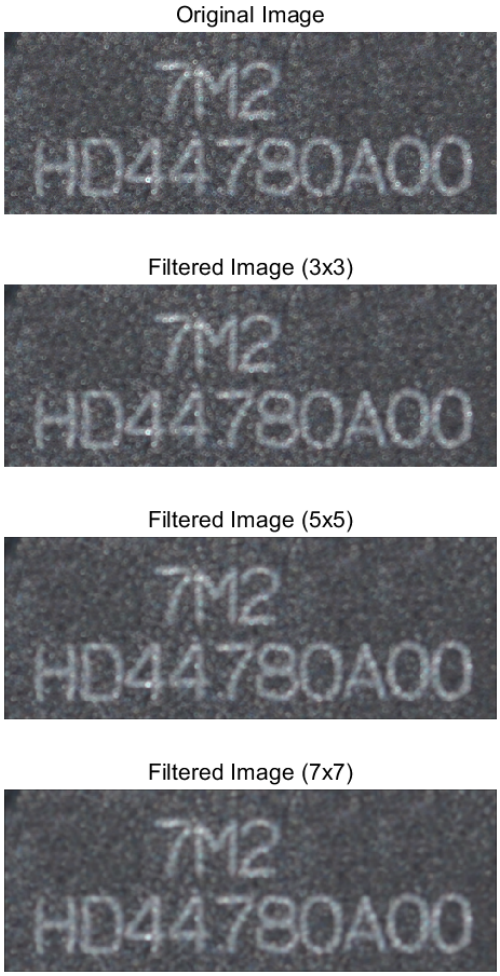

Fig. 3: Comparison of images processed with average filters of different sizes.

IV. Task 3. High-pass Filter

A. Task Overview

In this task, a high-pass filter is implemented and applied to the image in the frequency domain. The objective of the high-pass filter is to emphasize high-frequency components, such as edges and fine details, while suppressing low-frequency components like smooth regions and noise.

B. Algorithm Description

The high-pass filter is first applied in the frequency domain, where the image is transformed to its frequency representation using the Fourier Transform. After applying the filter, the image is transformed back to the spatial domain using the inverse Fourier Transform.

We use three types of high-pass filters in this task: **Ideal**, **Gaussian**, and **Butterworth** filters. Each of these filters is defined mathematically as follows:

1) Ideal High-Pass Filter

The Ideal High-Pass Filter passes all frequencies higher than the cutoff frequency and blocks all lower frequencies. It is defined as:

$$H(u,v) = \begin{cases} 1 & \text{if } D(u,v) > D_0 \\ 0 & \text{if } D(u,v) \leq D_0 \end{cases}$$

where $D(u,v)$ is the distance from the origin (center) in the frequency domain, calculated as:

$$D(u,v) = \sqrt{(u-u_0)^2 + (v-v_0)^2}$$

and D_0 is the cutoff frequency. Here, u and v represent the frequency components in the horizontal and vertical directions, respectively, in the frequency domain, while u_0 and v_0 are the coordinates of the origin (center of the frequency domain).

2) Gaussian High-Pass Filter

: The Gaussian High-Pass Filter gradually suppresses low frequencies, with a smoother transition compared to the ideal filter. It is defined as:

$$H(u,v) = 1 - \exp\left(-\frac{D(u,v)^2}{2D_0^2}\right)$$

where D_0 is the cutoff frequency, and $D(u,v)$ is the distance from the origin as defined previously.

3) Butterworth High-Pass Filter

The Butterworth High-Pass Filter provides a smoother transition between pass and stop bands, with the order n affecting the sharpness of the transition. It is defined as:

$$H(u,v) = \frac{1}{1 + \left(\frac{D_0}{D(u,v)}\right)^{2n}}$$

where D_0 is the cutoff frequency, n is the filter order, and $D(u,v)$ is the distance from the origin as defined earlier.

C. Experiment Result

In this experiment, we applied different high-pass filters to the grayscale image of the original to observe their effects in both the frequency and spatial domains. The filters evaluated include the Ideal, Gaussian, and Butterworth filters. For comparison, we used different cutoff frequencies and filters, as summarized in Table IV:

Filter Type	**Cutoff Frequency** (D_0)
Ideal	0.08, 0.5
Gaussian	0.08
Butterworth	0.08

TABLE IV: Comparison of high-pass filters and their cutoff frequencies.

The results of applying these filters are shown in Figure 4, where we compare the original grayscale image with the results of each filter.

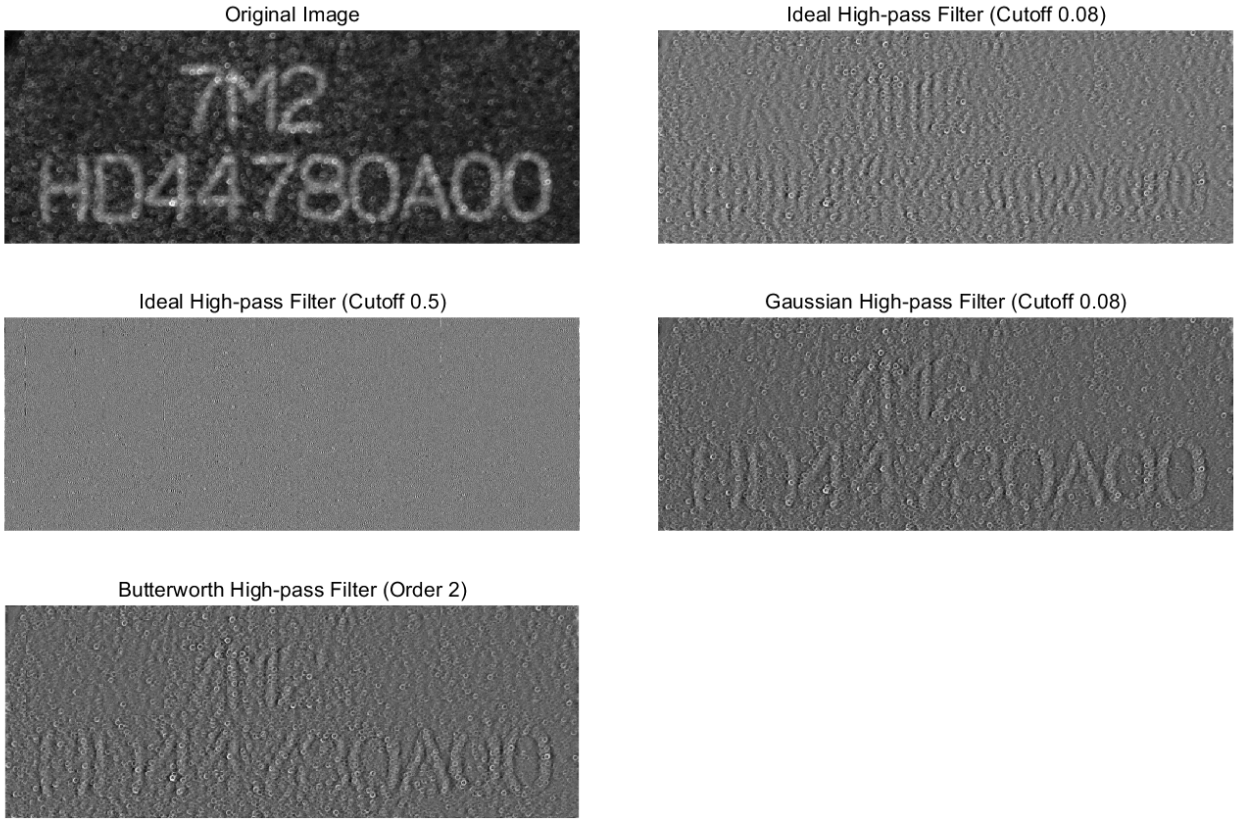

Fig. 4: Comparison of performance of different high-pass filters.

D. Comparison and Discussion

The high-pass filter effectively sharpens edges and enhances fine details in the spatial domain, while suppressing smooth regions. The cutoff frequency plays a crucial role in the filter's performance. If the cutoff is too high, important details such as edges are lost.

In this experiment, the **Ideal** High-pass Filter caused significant loss of edge details, particularly for the characters, as it harshly blocks low frequencies, including those that define the edges.

In contrast, both the **Gaussian** and **Butterworth** High-pass Filter produced better results by offering a smoother transition between the pass and stop bands, preserving more character edges.

However, for our character extraction task, **high-pass filters are not ideal** as they suppress low-frequency components that are essential for distinguishing the characters.

V. Task 4. Sub-image Cropping

A. Task Overview

In this task, we are required to create a sub-image from the original image, focusing on the center line labeled "HD44780A00." This involves cropping the original image to extract a smaller region that includes this specific center line.

B. Method Description

The image cropping process primarily utilizes the MATLAB function `imcrop(I, rect)`, where the rectangle `rect` defines the region to be extracted. In this task, the rectangle's size was manually adjusted to capture the area containing the center line "HD44780A00."

After experimenting and fine-tuning the cropping region, we determined that the rectangle with coordinates $[40, 200, 960, 340]$ produced the optimal result.

C. Experiment Result

The results of the sub-image cropping are shown in Figure 5, containing the center line "HD44780A00" as required.

D. Comparison and Discussion

The cropping method successfully extracted the desired region; however, it was performed manually. Future work could explore automated techniques for identifying and cropping the relevant areas, improving efficiency and consistency.

VI. Task 5. Image Binarization

A. Task Overview

In this task, we convert the sub-image obtained in Task 4 into a binary image. To refine the binary image, we developed and applied morphological operations such as closing,

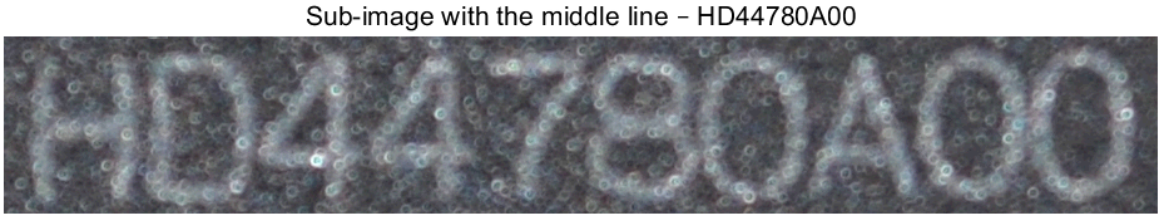

Fig. 5: Cropped sub-image containing the center line "HD44780A00".

opening, erosion, and dilation. Additionally, we implemented four thresholding methods for binarization: global thresholding, double thresholding, Otsu's thresholding, and a custom implementation of Otsu's thresholding developed from scratch.

It is important to note that unlike grayscale images, **binary images contain only two values (0 and 1)**.

B. Algorithm Description

This task utilizes morphological operations, including closing, opening, erosion, and dilation, along with four thresholding methods for binarization: global thresholding, double thresholding, and Otsu's thresholding.

1) Morphology Operation

Morphological operations are fundamental preprocessing methods used for binary or grayscale images. In this task, we developed four basic morphological operations: **erosion**, **dilation**, **opening**, and **closing**, and applied them to refine the binary image and remove small-scale noise.

Erosion: Erosion is a process that reduces the boundaries of the foreground object in an image. In this operation, the structuring element (usually a small matrix) is passed over each pixel, and the output pixel is set to the minimum value of all pixels under the structuring element. It is defined as:

$$\text{erosion}(I) = \min\left(I(x+u, y+v)\right) \quad \forall (u,v) \in SE \tag{7}$$

where I is the input image, SE is the structuring element, and (x, y) are the coordinates of the pixel being processed. The operation effectively removes small foreground structures or noise, shrinking the objects.

Dilation: Dilation is the inverse of erosion. It expands the boundaries of the foreground object by placing the structuring element on each pixel and setting the output pixel to the maximum value of all pixels under the structuring element. It is defined as:

$$\text{dilation}(I) = \max\left(I(x+u, y+v)\right) \quad \forall (u,v) \in SE \tag{8}$$

where I is the input image and SE is the structuring element. Dilation enlarges the objects in the image, helping to fill small holes or gaps in the foreground.

Opening: Opening is a morphological operation that consists of erosion followed by dilation. It is useful for removing small objects or noise from the image. The process works as follows:

- Apply erosion to shrink the foreground.
- Apply dilation to expand the remaining structures back, but the small noise removed by erosion is not restored, thus reducing it.

Closing: Closing is the opposite of opening. It involves dilation followed by erosion, which helps to close small holes and gaps in the foreground. This operation is particularly effective for closing gaps between objects or filling small holes within the objects. The process works as follows:

- Apply dilation to expand the foreground.
- Apply erosion to shrink the expanded regions, closing small holes and gaps.

2) Global Thresholding

Global thresholding is a simple and widely used method for converting a grayscale image into a binary image. In this method, each pixel in the image is compared to a fixed threshold value. If the pixel intensity is greater than the threshold, the pixel is set to one (white); otherwise, it is set to zero (black). This method is effective when there is good contrast between the foreground and background. The algorithm for global thresholding is as follows:

Algorithm 3: Global Thresholding Algorithm

Input: Grayscale image I, threshold T
Output: Binary image I_{binary}

```
foreach pixel (i, j) in I do
    if Intensity(I(i, j)) > T then
        Set I_binary(i, j) ← 1
    end
    else
        Set I_binary(i, j) ← 0
    end
end
return I_binary
```

In addition to the global thresholding, histogram equalization was also applied to the grayscale image, both with and without morphological operations. The results showed an improvement over direct grayscale processing, enhancing the overall effectiveness of the thresholding.

3) Double Thresholding

Double thresholding is a method used for segmenting an image based on two threshold values. The binary image is created by assigning the value 1 (white) to pixels that fall between the two thresholds and 0 (black) to all other pixels. The algorithm is as follows:

Algorithm 4: Double Thresholding Algorithm

Input: Grayscale image I, lower threshold T_{low}, upper threshold T_{high}
Output: Binary image I_{binary}
foreach *pixel* (i, j) *in* I **do**
  if $T_{low} < Intensity(I(i, j)) < T_{high}$ **then**
    Set $I_{\text{binary}}(i, j) \leftarrow 1$
  end
  else
    Set $I_{\text{binary}}(i, j) \leftarrow 0$
  end
end
return I_{binary}

4) Otsu Thresholding

Otsu thresholding is a global thresholding method that automatically determines the optimal threshold for binarizing an image by maximizing the between-class variance. A basic assumption of this method is the image contains two classes of pixels (background and foreground), and the goal is to find the threshold that best separates these two classes.

The algorithm works by iterating over all possible thresholds and calculating the between-class variance for each threshold. The threshold that maximizes the variance is chosen as the optimal threshold. The pseudocode for Otsu thresholding is as follows:

Algorithm 5: Otsu Thresholding Algorithm

Input: Grayscale image I
Output: Binary image I_{binary}
Compute the histogram H for I;
Normalize histogram to get probabilities $P(i)$ for each intensity level i;
Initialize variables: $\sigma_b^{\max} \leftarrow 0, T \leftarrow 0$;
for $t \leftarrow 1$ *to* 255 **do**
  Compute probabilities $P_1(t)$ and $P_2(t)$ for classes separated by threshold T;
  Compute class means $\mu_1(t)$ and $\mu_2(t)$;
  Compute between-class variance $\sigma_b(t) = P_1(t) \cdot P_2(t) \cdot (\mu_1(t) - \mu_2(t))^2$;
  if $\sigma_b(t) > \sigma_b^{max}$ **then**
    Update $\sigma_b^{\max} \leftarrow \sigma_b(t)$ and $T \leftarrow t$;
  end
end
return Thresholded image $I_{\text{binary}} = (I > T)$

In this task, we implemented the Otsu thresholding method from scratch to compute the optimal threshold for binarizing the grayscale image.

5) Local Thresholding

Local thresholding is used to binarize an image based on local pixel intensity values rather than a global threshold. This method applies a local threshold to each pixel using the mean intensity of a neighborhood defined by a square filter of a specified size, making it effective for images with uneven illumination. The procedure is as follows:

- Apply a square averaging filter of size `filter_size` to the grayscale image.
- For each pixel, the local mean of its neighborhood is computed.
- The pixel value is then compared to the local mean to determine the binary output.

C. Experiment Results

In this task, various thresholding methods were applied to binarize the sub-image. The performance of each method was assessed and compared based on their ability to segment the desired features effectively. Morphological opening operation was applied to the processed binary images for noise reduction.

Figure 6 illustrates the effect of the four thresholding methods on the sub-image, comparing results before and after the application of morphological operations.

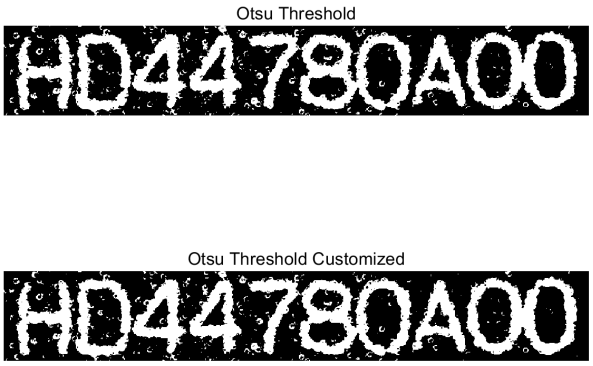

Fig. 7: Comparison of Otsu thresholding using MATLAB's `graythresh` and the custom implementation.

Figure 7 compares the results of Otsu thresholding performed using MATLAB's built-in `graythresh` function and the custom implementation developed from scratch. Both methods yielded similar results, validating the effectiveness of the custom implementation.

D. Comparison and Discussion

Unlike grayscale images, binary images contain only two values (0 and 1). The morphological operations developed for this task were specifically designed to refine the binary results, but they are only applicable to grayscale images. As a result, they cannot be directly used for binary images.

Fig. 6: Comparison of binary thresholding results with and without applying morphological opening to the input grayscale image.

Some improvements should be made in the future for better generalization of the developed functions.

From Figure 6, it is evident that regardless of the binary thresholding method used, the performance improved after applying morphological opening. Both Otsu's method and global thresholding combined with histogram equalization produced excellent results. However, while Otsu's method effectively identifies the optimal threshold, global thresholding paired with histogram equalization demonstrated superior noise reduction (fewer small holes), resulting in better performance for the current application.

The background noise observed in the local thresholding method exhibits a speckled pattern, which occurs because the local thresholding is highly sensitive to local variations in intensity. This sensitivity causes small inconsistencies in the background.

VII. Task 6. Edge Detection

A. Task Overview

In this section, we focus on extracting the outlines of characters in the image and compare the results of edge extraction from both the original grayscale image and the binary image. We used three thresholding methods for binary image segmentation and three operators for edge extraction.

B. Algorithm Description

Three operators are utilized in this section for edge extraction: Sobel, Canny, and Laplacian.

1) Sobel Operator

The Sobel Operator detects edges by calculating the gradient magnitude, highlighting areas where intensity changes, typically corresponding to edges.

Algorithm 6: Sobel Operator

Input: Grayscale image I
Output: Edge-detected image I_{edge}

[rows, cols] = size(img);
Initialize *grad_x* and *grad_y* with the same size as I;
foreach *pixel* (i, j) *in* I, *where* $2 \leq i \leq rows - 1$ *and* $2 \leq j \leq cols - 1$ **do**
  $\text{grad\_x}(i, j) \leftarrow$ $\text{abs}(I(i-1, j+1) - I(i-1, j-1) + 2 \cdot I(i, j+1) - 2 \cdot I(i, j-1) + I(i+1, j+1) - I(i+1, j-1))$;
  $\text{grad\_y}(i, j) \leftarrow$ $\text{abs}(I(i-1, j-1) - I(i+1, j-1) + 2 \cdot I(i-1, j) - 2 \cdot I(i+1, j) + I(i-1, j+1) - I(i+1, j+1))$;
end
$I_{\text{edge}}(i, j) \leftarrow \text{grad\_x}(i, j) + \text{grad\_y}(i, j)$;
return I_{edge}

The operator works by applying Sobel kernels in both the horizontal and vertical directions to compute gradients. The process of the Sobel Operator is expressed in Algorithm 6.

2) Canny Operator

Canny Operator is a multi-stage edge detection process designed to identify edges with high precision. It is particularly effective at detecting both wide and thin edges, which are often challenging for simpler methods to capture. The steps of the edge detection algorithm are as follows:

1) Apply a Gaussian filter to smooth the image, reducing noise and preventing false edges.
2) Compute the gradient of the image to detect regions with rapid intensity changes, indicating potential edges.
3) Apply non-maximum suppression to thin out the edges, preserving only the most prominent edge pixels.
4) Perform edge tracing by hysteresis, where weak edges are discarded unless they are connected to strong edges.

Canny Operator is typically more computationally intensive due to its multi-step process but provides highly accurate edge detection.

3) Laplacian Operator

Laplacian Operator detects edges by calculating the second derivative of image intensity, highlighting regions with rapid intensity changes. This method is sensitive to noise but effective in detecting edges in smooth areas.

The steps of the Laplacian Operator edge detection process are as follows:

1) Convolve the image with a Laplacian kernel to compute the second derivative.
2) Identify areas of rapid intensity change, which correspond to edges.

The Laplacian operator can be applied using a simple kernel. In our case, the following kernel is selected to calculate the second derivative at each pixel:

$$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

C. Experiment Results

We compare the results of outines extraction from both the original grayscale image and the binary image, with the latter yielding significantly better outcomes. When applied directly to the original grayscale image, all three operators performed poorly, failing to clearly highlight the edges.

Of the three operators, Sobel delivered the most consistent and accurate edge detection results, while Canny, though effective, introduced a higher degree of noise. The Laplacian operator, despite its edge detection capabilities, failed to produce distinct results due to its sensitivity to noise.

VIII. Task 7. Image Segmentation

A. Task Overview

In this task, we aim to segment the image to separate and label the different characters as clearly as possible. To enhance the segmentation process, we apply various methods developed in earlier tasks, including morphological operations, contrast enhancement, image binarization, and average filtering.

The goal is to segment the image into distinct regions representing individual characters. After segmentation, each connected component (corresponding to a character) will be labeled with a unique identifier.

B. Algorithm Description

This section outlines the series of steps used to refine and segment the binary image.

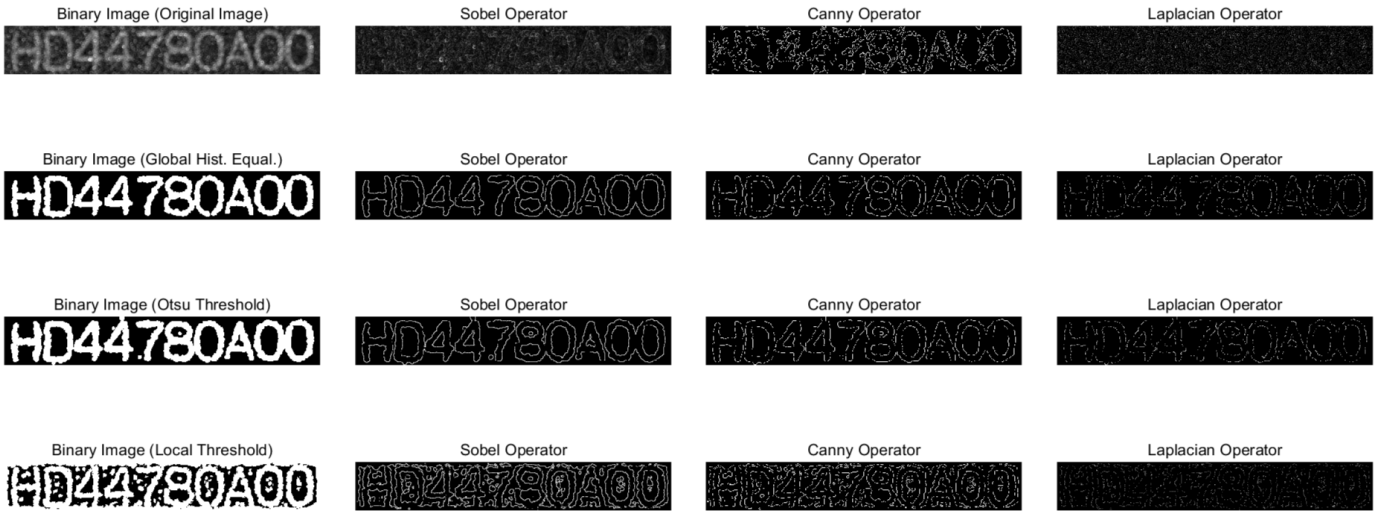

Fig. 8: Comparison of edge detection performance with different thresholding methods and operators.

1) Image Preprocessing

Image preprocessing plays a important role in ensuring the quality and accuracy of the segmentation process. It involves several operations to convert and refine the binary image input for effective segmentation. In our approach, the preprocessing steps include morphological opening, histogram equalization, global thresholding, and 5x5 average filtering. The steps are as follows:

1) **Morphological Opening**: The grayscale image from the raw undergoes an opening operation to remove small objects and noise.
2) **Histogram Equalization**: The processed image is enhanced by redistributing the pixel intensities over a defined range to improve contrast.
3) **Global Thresholding**: A binary thresholding is applied to convert the image into a binary form based on a specified threshold.
4) **Average Filtering**: An average filter (5x5) is applied to smooth the binary image and remove small-scale noise.

Figure 9 illustrates the entire pipeline of the segmentation system.

2) Image Segmentation

The image segmentation algorithm extracts individual characters from a binary image. It handles small regions by removing them based on a minimum area constraint. The algorithm assumes that there will be at most two connected components in the input binary image. The steps of the algorithm are outlined as follows:

Algorithm 7: Image Segmentation Algorithm

Input: Binary image img_bin;
Width threshold $width\_threshold$;
Minimum area min_area
Output: List of segmented images $imgs\_segmented$

Use $bwconncomp(img\_bin)$ to extract connected components;
Get BoundingBox and Area for each component;
$cnt \leftarrow 1$;
foreach *connected component* **do**
  Get the BoundingBox and Area of the component;
  if *Area is less than* min_area **then**
    continue
  end
  Crop the image using BoundingBox;
  if *width of the character exceeds* $width\_threshold$ **then**
    Split the character in half and store both parts;
  end
  Increment cnt
end
return $imgs\_segmented$

C. Experiment Results

In Figure 10, we present the step-by-step processing outputs of the cropped sub-image containing the center line "HD44780A00." The image demonstrates the sequence of operations applied to refine the image, including morphological

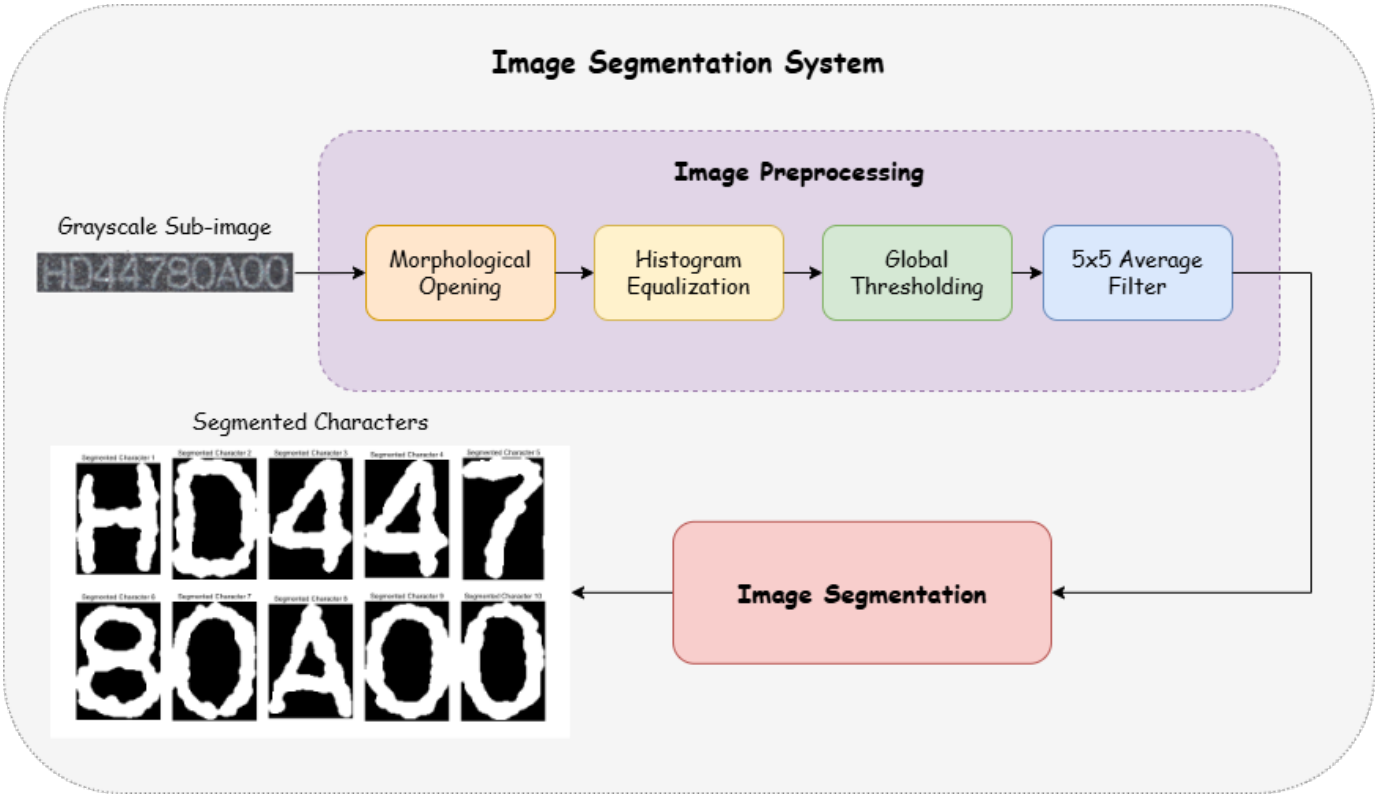

Fig. 9: Pipeline for the image segmentation system.

opening, histogram equalization, global thresholding, and 5x5 average filtering to prepare the image for segmentation.

Figure 11 presents the final segmented output. After the segmentation process, individual characters in the sub-image have been separated and labeled clearly.

D. Comparison and Discussion

In our algorithm, the segmentation process relies on the assumption that there are **at most two connected components**. This assumption **heavily depends on the quality of the preprocessing**. If the preprocessing is not effective, cases may arise where three characters are connected, making it impossible to split them into three distinct parts, and instead, they will be split into two.

Future work should address scenarios involving more connected characters, and could explore integrating OCR (Optical Character Recognition) to accurately categorize and label the segmented characters.

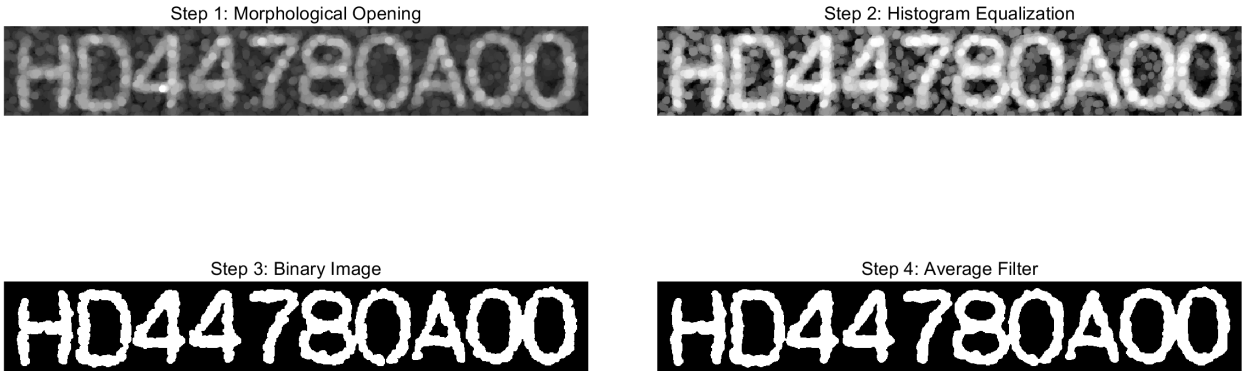

Fig. 10: Step-by-step processing of the sub-image.

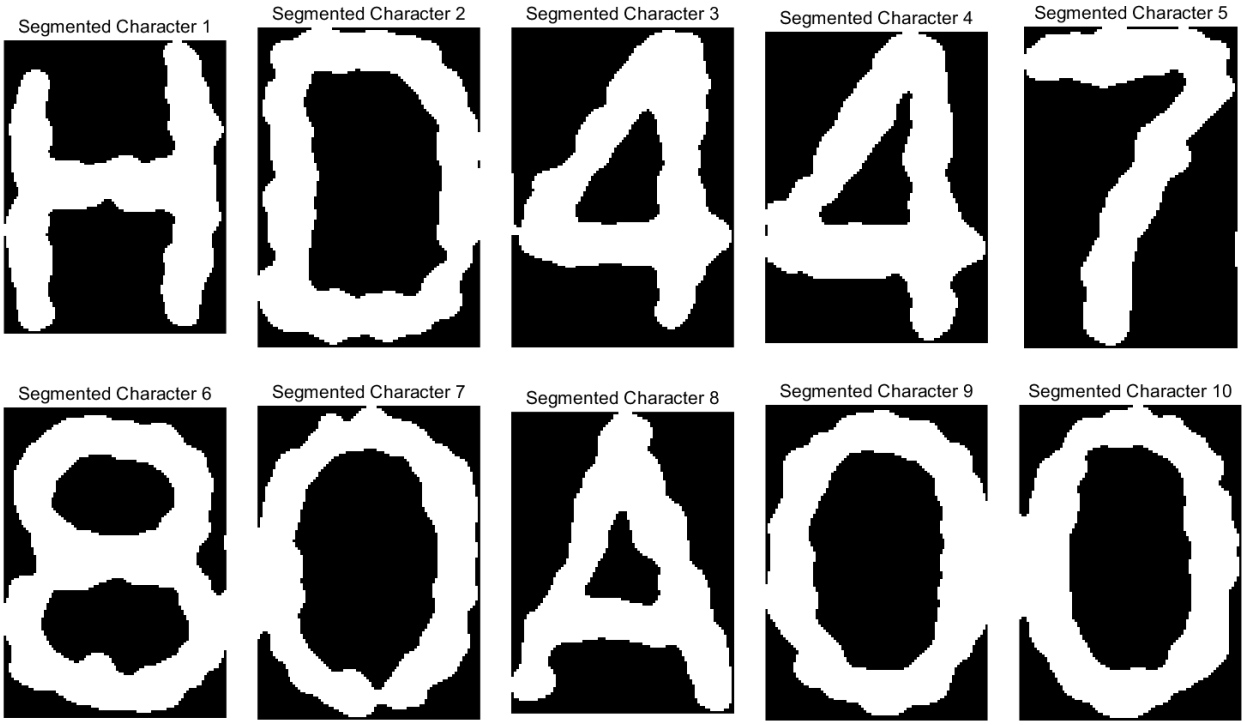

Fig. 11: Final segmented output.

IX. Task 8-I. Convolutional Neural Networks (CNN)

A. Introduction

Convolutional Neural Networks (CNN) are a cornerstone in the field of deep learning, particularly excelling in image processing tasks. Their unique ability to automatically learn and extract spatial hierarchies of features from images makes them highly effective for applications such as image classification, object detection, and medical imaging [1] [2].

Unlike traditional methods that rely heavily on handcrafted features, CNNs learn these features directly from raw data through layers of convolution, pooling, and activation functions. For example, they can classify images, recognize faces, or detect objects like traffic signs. CNNs are also widely applied in tasks like segmentation, where they identify and separate objects within an image, and in medical diagnosis, where they assist in analyzing complex patterns in medical scans.

A key strength of CNNs is their ability to process visual data by applying convolutional filters that focus on localized patterns, such as edges or textures, in the early layers. Deeper layers capture more complex structures, eventually leading to high-level representations. Pooling layers reduce the spatial dimensions of feature maps, improving computational efficiency and reducing overfitting. Fully connected layers integrate all features and serve as the final step for tasks like classification.

While newer architectures have emerged, CNNs remain highly relevant due to their efficiency and versatility in image-related tasks. For scenarios where large datasets are available, they provide a robust framework for learning complex patterns [3]. Considering the nature of our task in 8.1, where extracting meaningful visual features is crucial, we opted for a CNN-based approach to ensure reliable and accurate results.

B. Network Architecture

Our CNN is designed to extract and classify features from input images effectively. The network architecture consists of multiple layers, each with specific roles in the feature extraction and classification process. The details of our CNN architecture are as follows:

- **Input Layer**: The network accepts input images of size $64 \times 64 \times 3$, corresponding to a 64x64 resolution RGB image.
- **Convolutional Layers**: The network includes three convolutional layers, each with a kernel size of 3×3, designed to progressively extract features:
  - The first convolutional layer has 32 filters.
  - The second convolutional layer has 64 filters.
  - The third convolutional layer has 128 filters.

  Each convolutional layer is followed by a ReLU activation function to introduce non-linearity.
- **Batch Normalization Layers**: Batch normalization is applied after each convolutional layer to stabilize learning and speed up convergence.
- **Max Pooling Layers**: Each convolutional layer is followed by a 2×2 max pooling layer with a stride of 2, reducing the spatial dimensions of the feature maps and improving computational efficiency.
- **Fully Connected Layers**: After the final convolutional and pooling layers, the feature maps are flattened and passed through two fully connected layers:
  - The first fully connected layer has 128 neurons.
  - The second fully connected layer outputs the final class probabilities.
- **Softmax Layer and Classification Layer**: The softmax activation function is applied to the output of the final fully connected layer to compute the probability distribution over the target classes. The classification layer then determines the predicted class based on the highest probability.

The architecture of our CNN is illustrated in Figure 12. This architecture effectively captures hierarchical features from the input images and leverages them for accurate classification.

C. Cross-Entropy Loss

In our CNN, we employ the cross-entropy loss function to optimize the model for multi-class classification tasks. This loss function is particularly suitable as it evaluates the performance of the model by comparing the predicted probability distribution with the true labels. The goal of the model is to minimize this loss, thereby improving the accuracy of predictions.

For a classification problem with N samples and C classes, the cross-entropy loss is defined as follows:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} y_{i,j} \log(\hat{y}_{i,j}),$$

where:

- $y_{i,j}$ is the true label for the i-th sample and j-th class, represented as a one-hot encoded vector.
- $\hat{y}_{i,j}$ is the predicted probability of the i-th sample belonging to the j-th class, obtained from the softmax function:

$$\hat{y}_{i,j} = \frac{\exp(z_{i,j})}{\sum_{k=1}^{C} \exp(z_{i,k})},$$

where $z_{i,j}$ is the output (logit) of the model for the j-th class.

D. Training Process

The training process of our CNN is designed to ensure efficient learning and stable convergence. The algorithm diagram illustrates the general training process of our CNN. The process begins with the initialization of key parameters such as the batch size, learning rate, number of epochs, and the Adam optimizer. During each epoch, the model processes batches of training data through a forward pass, computes the loss using the cross-entropy function, and performs backpropagation to update the model's weights using the Adam optimizer. Every n iterations, the model is evaluated on the validation set to monitor its performance and prevent overfitting. The training continues until the maximum number of iterations is reached,

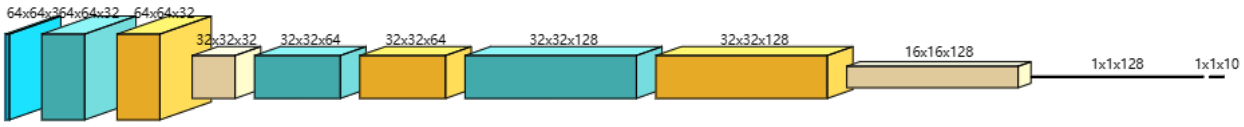

Fig. 12: CNN Architecture

Algorithm 8: Training Process of CNN

Input: Training dataset, Validation dataset
Output: Trained CNN model
Initialize: CNN architecture, batch size, learning rate, epochs, optimizer = Adam;
For each epoch: **for** *each batch in training data* **do**
    Perform forward pass through CNN;
    Compute loss using cross-entropy;
    Perform backpropagation;
    Update weights using Adam optimizer;
end
Evaluate on validation set: Every n iterations;
Check stopping criterion: If maximum iterations reached, stop;
Return: Trained CNN model;

at which point the model is returned as the trained CNN. This algorithm effectively outlines the essential steps involved in training our CNN model. Key parameters and techniques are configured as follows:

- **Batch Size**: The batch size is set to 32, which balances computational efficiency and stability. This size is particularly suitable for small to medium-sized datasets, allowing effective utilization of limited computational resources while maintaining robust gradient estimates.
- **Learning Rate and Optimizer**: The initial learning rate is set to 2×10^{-5}, a conservative value that ensures gradual convergence without large oscillations in the early stages of training. We employ the Adam optimizer, known for its adaptive learning rate and momentum properties, which enhance training stability and accelerate convergence. This combination helps the model achieve an optimal solution efficiently.
- **Epochs and Iterations**: Training is conducted over 5 epochs, with the MaxEpochs parameter computed to ensure the total number of iterations reaches the target of 1000. This setup ensures the model undergoes sufficient updates to learn effectively from the data.
- **Validation and Evaluation**: The validation frequency is set to every 30 iterations, providing periodic assessments of the model's performance on a separate validation set. This allows for early detection of overfitting and helps track the generalization ability of the model.
- **Data Shuffling**: To improve the generalization capability, the training data is shuffled at the start of each epoch. This prevents the model from learning patterns specific to the order of the training data.

X. Task 8-II. Support Vector Machines (SVM)

A. Introduction

Support Vector Machines (SVM) are powerful supervised learning algorithms widely used for classification and regression tasks [4]. In image processing, SVM plays a crucial role due to its ability to handle high-dimensional data effectively, making it suitable for various applications.

SVM is used primarily for **image classification** by separating data points into distinct classes based on their characteristics [5]. For example, it can identify objects, recognize faces, or classify handwritten digits by learning the relationship between image features and their corresponding labels. SVM is also used in **object detection**, often combined with feature extraction techniques such as the Histogram of Oriented Gradients (HOG), to detect specific objects such as pedestrians in images. Furthermore, SVM aids in **texture analysis** by categorizing image patterns and is a valuable tool in **medical image analysis**, where it helps classify medical scans and detect anomalies.

The core principle of SVM is to find the optimal hyperplane that separates data points of different classes while maximizing the margin between them [6]. For linearly separable data, the hyperplane is defined by a set of support vectors—data points closest to the decision boundary. In cases where data are not linearly separable, SVM employs **kernel functions** to map the

data to a higher-dimensional space, making them linearly separable. Common kernel functions include linear, polynomial, and radial basis function (RBF) kernels. By incorporating soft margins, SVM also handles noisy datasets by allowing some misclassifications to prevent overfitting.

Although SVM has been partially overshadowed by deep learning, its efficiency in small-scale, high-dimensional tasks and its interpretability make it a reliable choice for image processing tasks. Its ability to generalize well with limited data and its straightforward decision boundaries remain key advantages in practical applications. Given that our dataset is small, and the task is relatively clear, we decided to use the SVM approach in task 8.2.

B. Code structure and Features

In our code, the type of SVM used is Multi-class SVM and is implemented through the *fitcecoc* function.

Support Vector Machines (SVMs) are inherently a binary classification algorithm, but in real-world problems, many tasks (e.g. image classification) need to deal with multiple classes. Error-Correcting Output Codes (ECOC) is a way to convert a multi-class problem into multiple binary classification problems, and MATLAB's *fitcecoc* function is based on this to implement a multi-class classification support vector machine.

The ECOC method solves multi-class problems by dividing them into a set of binary classification problems. The core idea is the following.

1) Map the K categories into a coding matrix M of size K × L, where L is the number of binary classifiers.
   - Each row in the matrix represents a category.
   - Each column corresponds to a binary classifier.
2) Each bi-classifier groups the categories according to one of the columns of the coding matrix and determines to which group the samples belong.
3) The final classification result is determined by the combined vote or confidence of all classifiers.

Example of a coding matrix:

Suppose there are 4 categories *A*, *B*, *C*, *D*, the coding matrix *M* can be defined as follows:

$$M = \begin{bmatrix} +1 & -1 & +1 \\ +1 & +1 & -1 \\ -1 & +1 & +1 \\ -1 & -1 & -1 \end{bmatrix} \tag{9}$$

- The first column corresponds to the first binary classifier, the classification task is to distinguish between {A, B} and {C, D}.
- The second column corresponds to the second binary classifier, the classification task is to distinguish between {A, C} and {B, D}.
- The third column corresponds to the third dichotomous classifier, and the classification task is to distinguish between {A} and {B, C, D}.

Voting decision:

For testing, each sample is compared to the rows of the coding matrix by the predicted values of all binary classifiers to find the closest category.

C. Default steps

MATLAB implements the ECOC multi-class SVM using *fitcecoc* with the following steps by default:

(1) Decomposition strategy

- The default strategy is One-vs-One:
- Train a binary classifier for any two categories of data.
- If the number of categories is *K*, then you need to train $C(K,2) = \frac{K(K-1)}{2}$ classifiers.
- For example, for 4 categories A, B, C, and D, 6 binary classifiers need to be trained: A vs B, A vs C, A vs D, B vs C, B vs D, and C vs D.

(2) Nuclear Functions

- The default kernel function is a linear kernel:

$$K(x,z) = x \cdot z \tag{10}$$

- User can customize kernel functions (e.g. RBF kernel, polynomial kernel) via *templateSVM*:

t = templateSVM('KernelFunction', 'rbf');

SVMModel = fitcecoc(train_data, imdsTrain.Labels, 'Learners', t);

(3) Classification decision

For testing, for each sample, the results of all the binary classifiers are combined and compared with the rows of the coding matrix to do the final classification using the following strategy:

- Voting strategy: select the category with the highest number of predictions.
- Confidence strategy: use the confidence score of the classifier output to select the closest category.

The structure of our code is shown in Algorithm 9.

D. Cross validation of the Model

Cross-validation is a robust statistical method for assessing the generalization performance of a machine learning model. It splits the dataset into multiple subsets (folds) to evaluate the model's performance on unseen data, thereby estimating its ability to generalize to independent datasets. This technique is particularly useful for avoiding overfitting, ensuring the model performs well not only on the training data but also on new, unobserved data.

In our validation, we choose the K-Fold Cross-Validation, which is one of the most commonly used cross-validation techniques. The process is shown in Algorithm 10:

Cross-validation is a cornerstone of model evaluation in machine learning, ensuring that the selected model is both accurate and generalizable.

E. Summary

ECOC-based multi-class SVM is a classic approach to multi-classification tasks, decomposing complex multi-class problems into tractable bi-classification problems and integrating the results through coding matrices and voting strategies. MATLAB's *fitcecoc* provides an efficient implementation suitable for multi-class tasks such as image classification, but on large-scale datasets it may need to be considered to optimize the computational efficiency or combine with other methods (e.g. neural networks) to improve performance.

Algorithm 9: Test Set Evaluation and Visualization for ECOC Multi-class SVM

Input: character_folder (Folder with test images), SVMModel (Trained ECOC SVM model)
Output: test_accuracy (Accuracy on the test set), Visualization of predictions

```
/* Step 1: Load and preprocess the
   test dataset                          */
```
imdsTest ← Load dataset with labels manually assigned;
test_data ← Reshape and normalize images in imdsTest;
```
/* Step 2: Predict and calculate test
   accuracy                              */
```
test_pred ← Predict(test_data, SVMModel);
test_accuracy ← $\frac{\text{Correct predictions}}{\text{Total test samples}}$;
```
/* Step 3: Visualize predictions         */
```
Initialize a 2x5 grid for displaying test images;
foreach *test image I with index i* **do**
    Display I in subplot i;
    True_Label ← imdsTest.Labels(i);
    Pred_Label ← test_pred(i);
    if *True_Label == Pred_Label* **then**
        Set title color to green;
    else
        Set title color to red;
    end
    Display title: "True: True_Label, Pred: Pred_Label";
end
Add overall test accuracy as a title for the figure;
return test_accuracy;

Algorithm 10: K-Fold Cross-Validation

Input: Dataset $\mathcal{D}$ with n samples, number of folds K, machine learning model $\mathcal{M}$
Output: Average performance metric $\bar{M}$

Initialize: Divide $\mathcal{D}$ into K folds: $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K$; Initialize metrics = [];
for $k \leftarrow 1$ *to* K **do**
    Set: $\mathcal{D}_{\text{train}} = \bigcup_{i \neq k} \mathcal{D}_i$, $\mathcal{D}_{\text{val}} = \mathcal{D}_k$;
    Train model $\mathcal{M}$ on $\mathcal{D}_{\text{train}}$;
    Evaluate $\mathcal{M}$ on $\mathcal{D}_{\text{val}}$ to obtain performance metric M_k;
    Append M_k to metrics;
end
Compute: $\bar{M} = \frac{1}{K} \sum_{k=1}^{K} M_k$;
Return: $\bar{M}$;

XI. Task 8-III. Comparison between CNN and SVM

A. Training and classification result of CNN

The training of the CNN was conducted using MATLAB's **trainNetwork** function. During the training process, the accuracy and loss were monitored and are shown in Figure 13, illustrating the model's performance over time. After 1000 iterations of training, the model achieved an impressive accuracy of 96.23% on the validation set. This demonstrates the effectiveness of our CNN architecture in learning the relevant features from the dataset and generalizing well to unseen data. The steady improvement in accuracy and the reduction in loss during training further highlight the network's successful optimization process. For the preprocessed characters, as shown in Figure 14, CNN achieved a recognition accuracy of 100%, which indicates that the model was able to correctly classify each character, reflecting its effective performance in identifying the relevant features of the input data. This result suggests that CNN successfully learned the key patterns and characteristics during training, enabling it to make accurate predictions on the given dataset.

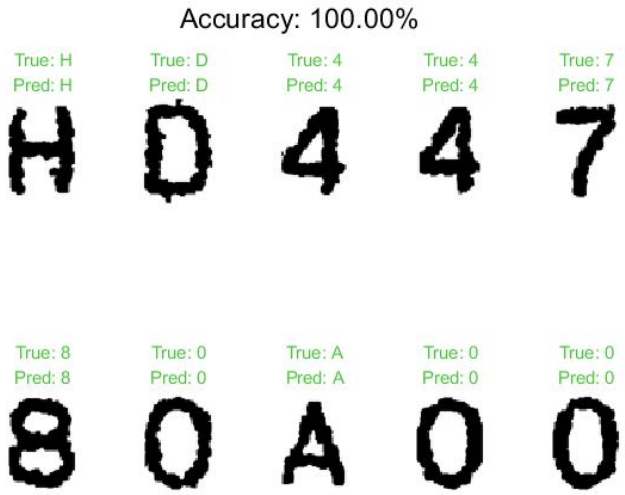

Fig. 14: The classification results based on CNN

B. Training and classification result of SVM

For the support vector machine (SVM) based character recognition methods, the recognition accuracy for the test set reached 93.9%. In addition, we used the *crossval* function to cross-validate the model during our testing. The results of the *crossval* function were also applied to calculate the Out-of-Sample Loss of the model, *oosLoss*, which is a scalar that represents the prediction error of the model and is used to measure the generalization ability of the model. A lower value of this represents a better predictive ability of the model. For our SVM model, the *oosLoss* is 0.0116 which is a quite low value. And for the preprocessed characters, the recognition accuracy reached 90%, which means that 9 out of 10 characters were recognized correctly, and the only character that was not recognized correctly was the character D, who was incorrectly recognized as a 0. The final results have been illustrated in Figure 15. The probable reason for this could be that the

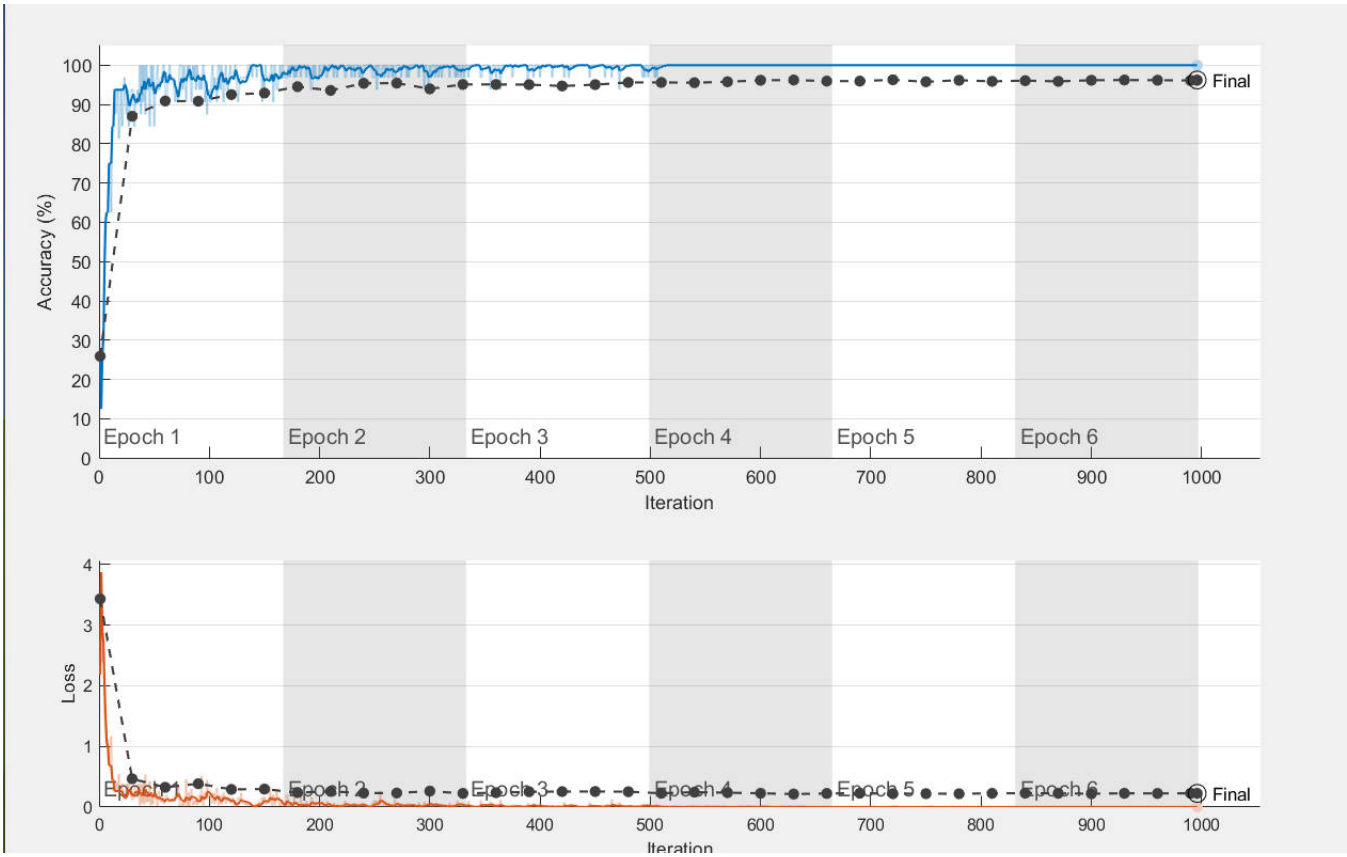

Fig. 13: The training results based on CNN

outlines of the characters D and 0 are very similar in the style of the microchip label fonts. At the same time, it is possible that some features that could be used for recognition by a support vector machine were lost during the preprocessing process. If the extracted features (e.g., edges, contours, shapes, etc.) are not enough to distinguish the two, the SVM classifier may confuse them.

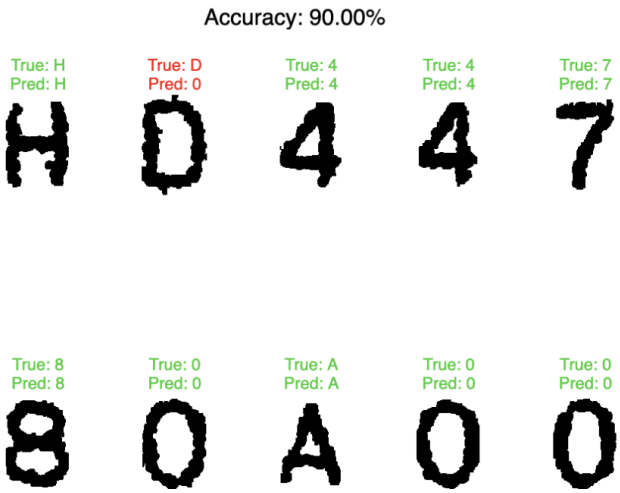

Fig. 15: The classification results based on SVM

C. Comparison between CNN and SVM

We compared the performance of Convolutional Neural Networks (CNN) and Support Vector Machines (SVM) for the same character recognition task. Based on the results provided, a detailed comparison of the two methods is presented below, highlighting their respective strengths and weaknesses:

1) Accuracy

CNN:

- Reached a validation accuracy of 96.23% after 1000 training iterations, showcasing strong generalization capabilities.
- Achieved a 100% recognition accuracy for preprocessed characters, demonstrating its ability to correctly classify all characters in the dataset.
- Advantages: CNN leverages its deep architecture to automatically extract multi-level features, such as edges, textures, and shapes, making it highly effective for complex pattern recognition tasks.

SVM:

- Attained a test set recognition accuracy of 93.9%, which is slightly lower than CNN's performance.
- For preprocessed characters, the recognition accuracy was 90%, with one notable misclassification: the character "D" was confused with "0."
- Limitations: SVM heavily relies on manually extracted features. If preprocessing fails to preserve

critical distinguishing features, the model's classification performance can be negatively impacted.

2) Generalization Capability
CNN:
- Demonstrated a steady improvement in accuracy and reduction in loss during training, indicating a successful optimization process.
- By leveraging large-scale data, CNN can learn complex nonlinear patterns, resulting in superior generalization to unseen data.

SVM:
- Cross-validation using the *crossval* function yielded an out-of-sample loss (*oosLoss*) of 0.0116, indicating good predictive ability.
- However, compared to CNN, SVM's generalization capability is limited by the quality of the extracted features. For instance, the similarity between "D" and "0" caused confusion, likely due to insufficient distinguishing features in the preprocessing step.

3) Feature Extraction and Adaptability
CNN:
- Excels in automatic feature extraction, eliminating the need for manual intervention.
- Particularly well-suited for character recognition tasks, as it effectively captures spatial correlations such as strokes and shapes.
- Advantages: Minimizes reliance on feature engineering, providing better adaptability to diverse datasets.

SVM:
- Relies on manually designed features such as edges, contours, and textures.
- Feature loss during preprocessing can lead to reduced performance, as seen in the misclassification of "D" as "0."
- Limitations: Less adaptable to complex or poorly preprocessed datasets compared to CNN.

4) Model Complexity and Computational Cost
CNN:
- Requires significant computational resources for training, particularly for deep architectures, and often necessitates GPU acceleration.
- Performs best with large datasets, which may not always be available.
- Disadvantages: High hardware and data requirements.

SVM:
- Computationally efficient, especially for small datasets, and has lower hardware requirements.
- Advantages: Faster training times and suitability for limited data scenarios.

D. Summary

In conclusion, CNN demonstrated superior recognition accuracy and generalization capability, particularly excelling in visual tasks like character recognition. Its ability to automatically learn features reduces the reliance on manual feature engineering, making it highly effective for complex datasets. On the other hand, SVM, while less accurate in this task, remains computationally efficient and well-suited for smaller datasets with carefully designed features. Although SVM's performance was limited by feature extraction quality, it can still be a viable option in scenarios with constrained resources. The results underscore CNN's advantages in tasks requiring robust pattern recognition and adaptability, though SVM offers a practical alternative for simpler applications.

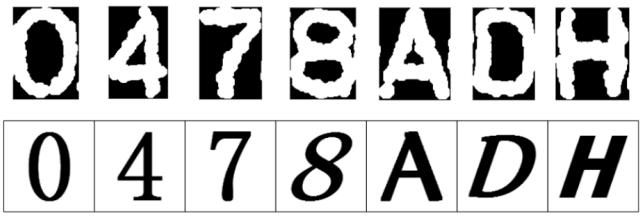

Fig. 16: Comparison of images from original characters and train set

XII. Task 9. Improvement and optimization

A. Data-processing

1) Pre-process stage

In the process of character recognition, we noticed that the boundaries of the segmented characters are at the edges of the images, while most of the characters in the training set leave gaps with the edges of the images. (shown in Figure 16) Therefore, we decided to preprocess the data of the processed characters before feeding them into the CNN and SVM models. Specifically, we initially invert the pixel values of the images. Subsequently, character scaling was implemented using MATLAB functions to maintain the original image size and automatically fill the extra pixels with zeros. The results demonstrate that characters after the above processing are more easily recognized and classified.

2) Feature extraction stage

Based on the previous task, we also do edge detection on the image to be recognized, Edge Detection plays an important role in character recognition preprocessing, especially in helping to extract the contours of the characters, enhance the features and reduce the interfering information, which is well suited for character image recognition work. In addition, we believe that skeletonization, horizontal/vertical projection may also be helpful for feature extraction of characters.

3) Data augmentation

Data augmentation is a critical preprocessing step to expand the training dataset and improve the generalization ability of neural networks. It creates varied versions of the original images, simulating different scenarios the model might encounter, and helps avoid overfitting. Since there are only 763 images in the training set for each character, data augmentation

is pretty beneficial to expand the training dataset. Below are common data augmentation techniques for character recognition:

1) **Translation:**
   Translation shifts the character image slightly in the horizontal or vertical direction without altering the character itself. This augmentation accounts for misalignments or varying positions in real-world data.
2) **Rotation:**
   Rotation involves turning the character image by a small random angle, simulating different orientations. This process makes the model robust to variations in orientation caused by scanning or user.
3) **Scaling:**
   Scaling changes the size of the character while keeping it proportional. It helps the model recognize characters at different scales.It ensures robustness to size variations in the input data.
4) **Adding Noise:**
   Adding noise (e.g., Gaussian noise, salt-and-pepper noise) mimics imperfections like scanning artifacts or image distortions. This way Increases model robustness to noisy or degraded inputs.The Gaussian noise models random variation in intensity values, while the Salt-and-pepper noise mimics pixel corruption.

B. Hyper-parameters Tuning

In this experiment, several key hyperparameters were considered, including batch size, learning rate, and number of epochs. The batch size controls the number of samples processed in each training iteration. For our model, a batch size of 32 was selected to balance computational efficiency and gradient estimation stability. The number of epochs represents how many times the entire training dataset is passed through the model. We set this to 6, ensuring the model trains sufficiently while maintaining the target of 1000 iterations.

Among these hyperparameters, the learning rate was subject to detailed analysis due to its significant impact on the training process. The learning rate determines the magnitude of weight updates during optimization, directly influencing the model's convergence behavior. We tested multiple values, including 1×10^{-5}, 2×10^{-5}, 1×10^{-4}, 2×10^{-4} and 1×10^{-3}. As the results shown in Figure 17, A higher learning rate (1×10^{-3}) accelerated convergence but led to instability, with erratic loss values and reduced validation accuracy. On the other hand, a lower learning rate (1×10^{-5}) provided stable training but was too slow to achieve optimal performance within the iteration limit. The chosen learning rate of 1×10^{-4} offered a balance between stability and convergence speed, allowing the model to achieve a validation accuracy of 96.23%. This analysis underscores the importance of tuning the learning rate to maximize model performance while maintaining training efficiency.

C. Summary

To enhance character recognition accuracy, the data processing workflow involved pre-processing, feature extraction, and data augmentation. During pre-processing, inconsistencies between segmented character images and the training set were addressed by inverting pixel values and scaling characters to maintain a uniform image size, with zero-padding for gaps. This improved model compatibility and recognition accuracy. Feature extraction utilized edge detection to emphasize character contours, reduce noise, and enhance key features, while techniques like skeletonization and horizontal/vertical projections were identified as complementary methods. Data augmentation was crucial to expand the training dataset, employing translation, rotation, scaling, and noise addition to simulate real-world scenarios and boost the model's robustness against positional, orientational, and quality variations.

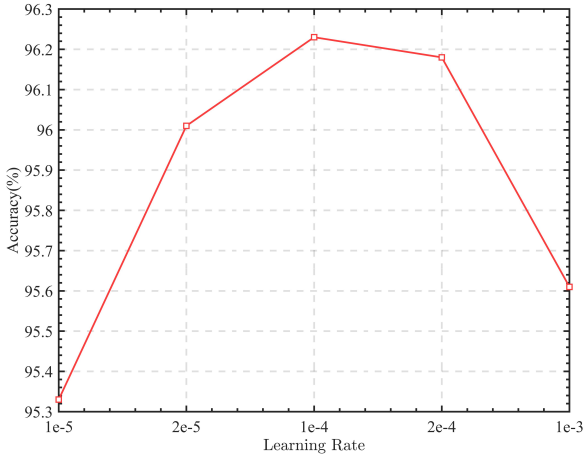

Fig. 17: Accuracy via different learning rate

Hyper-parameter tuning further optimized the model's performance. A batch size of 32 balanced computational efficiency and gradient stability, while six training epochs ensured sufficient exposure to the dataset. A detailed analysis of learning rates revealed that 1×10^{-4} provided an optimal balance, enabling stable convergence and achieving a high validation accuracy of 96.23%. Higher learning rates caused instability, while lower ones resulted in slower training. This systematic combination of data processing, augmentation, and fine-tuned parameters significantly enhanced the model's accuracy and robustness for character recognition tasks.

REFERENCES

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, May 2017. [Online]. Available: https://doi.org/10.1145/3065386
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
[4] C. Cortes and V. Vapnik, "Support-vector networks," vol. 20, no. 3, pp. 273–297. [Online]. Available: http://link.springer.com/10.1007/BF00994018
[5] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A practical guide to support vector classification."
[6] D. Decoste, "Training invariant support vector machines."