NOVEMBER 18, 2024

ME 5411 ROBOT VISION AND AI: CA – AY24/25 SEMESTER 1

PROJECT GROUP 5:

GOH JIAN WEI NIGEL	STUDENT ID: A0304764A
LI HONGYI	STUDENT ID: A0285183X
WANG HEXIAN	STUDENT ID: A0304376E

Table of Contents

1. Introduction

This report describes the application of various image processing techniques taught in ME5411 on a BMP image of a label on a microchip (figure 1 below). The following sections from tasks 1 to 7 will detail the effects of implementing these techniques on the image and the conclusions drawn from applying them.

Following this, the report will cover the design of a classification system using both a Convolutional Neural Network (CNN) and non-CNN methods. It will report the effectiveness of both approaches and conclusions derived from them.

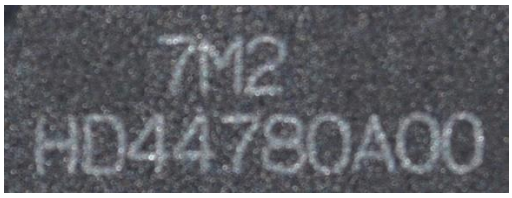


Figure 1: Label on a Microchip

2. Task 1: Contrast Enhancement

2.1. Introduction

The objective of task 1 was to print the original microchip label image on the screen and to experiment with different contrast enhancement techniques. The following list of contrast enhancement techniques were used:

- Contrast Stretching
- Brightness Thresholding
- Histogram Equalization

2.2. Contrast Stretching

Contrast stretching is a technique used in image processing to enhance the contrast of an image by expanding the range of intensity levels. The goal is to make dark areas darker and light areas lighter, thereby improving the overall visibility of features in the image.

Contrast stretching can be described in Figure 2 below where the thresholds are set to r1 and r2 respectively. Pixels in the region of r2 to L-1 are specified as 255 (white) while pixels in the region of 0 to r1 are specified as 0 (black). Pixels in the "window" between the two regions are in grayscale

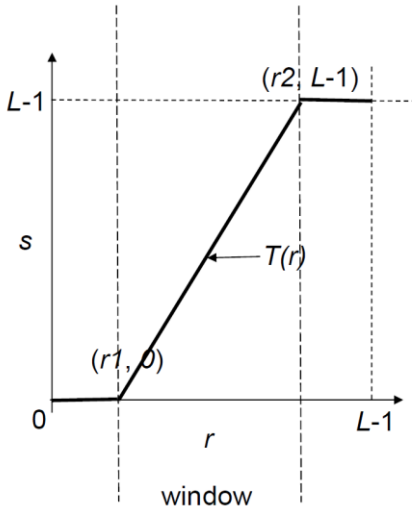


Figure 2: Contrast Stretching

2.2.1. Contrast Stretching Algorithm

Below are the steps for the Contrast stretching algorithm:

1) Image acquisition
    a. the image is read from the specified file path using the *imread* function.
    b. The image is converted from RGB to grayscale using the luminosity method (Grayscale Pixel = 0.2989R + 0.5870G + 0.1140B).
2) Display original image
    a. The original image is displayed using the *imshow* function.
3) Contrast stretching
    a. Contrast stretching is performed at different intensity ranges to observe the effects on the image. A contrast stretching function *contrast_stretch* is created to map the pixel intensity values of the image to a new range, and this range is defined by the parameters low and high. The image pixel values are converted to *double* data type and then shifted and scaled as per this equation

$$stretched = \frac{pixel - 255 \times low}{255 \times (high - low)}$$

        i. This formula maps the pixel values from their original range to the range defined by [255xlow, 255xhigh], where the low and high values are specified in the code.
    b. Following that, the pixel values are clipped between 0 and 1 to ensure that no values fall below 0 or exceed 1
    c. Normalised pixel values in this range are scaled to the 8-bit intensity range [0,255]
    d. Contrast stretching was applied with various low and high thresholds: 10-90%, 20-80%, 30-70%, and 40-60%.
4) The adjusted images are displayed

Algorithm Contrast Stretching

BEGIN ContrastStretching
Input: `imagePath` ← `'Photos/charact2.bmp'` ▷ Path to the input image
Output: Processed images with adjusted contrast
`originalImage` ← **ReadImage**(`imagePath`) ▷ Load the input image
if `originalImage` is a color image (3 channels) **then**
    `grayscaleImage` ← $0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$ ▷ Convert to grayscale using luminosity method
    Convert `grayscaleImage` to 8-bit integer format
else
    `grayscaleImage` ← `originalImage` ▷ Already grayscale
end if
DisplayImage(`grayscaleImage`, `"Original Microchip Image"`) ▷ Show the original image
Define Function: `ContrastStretch(image, low, high)` ▷ Contrast stretching function
    Convert `image` to double precision
    `stretched` ← $\frac{\text{image} - 255 \cdot \text{low}}{255 \cdot (\text{high} - \text{low})}$ ▷ Stretch pixel values
    Clip `stretched` to range [0, 1]
    Rescale `stretched` to range [0, 255] and convert to 8-bit integer
    Return `stretched`
Apply Contrast Stretching:
`tenPercentImage` ← `ContrastStretch(grayscaleImage, 0.10, 0.90)` ▷ Stretch with 10% lower and upper limits
DisplayImage(`tenPercentImage`, `"10% upper & lower limit"`)
`twentyPercentImage` ← `ContrastStretch(grayscaleImage, 0.20, 0.80)` ▷ Stretch with 20% lower and upper limits
DisplayImage(`twentyPercentImage`, `"20% upper & lower limit"`)
`thirtyPercentImage` ← `ContrastStretch(grayscaleImage, 0.30, 0.70)` ▷ Stretch with 30% lower and upper limits
DisplayImage(`thirtyPercentImage`, `"30% upper & lower limit"`)
`fortyPercentImage` ← `ContrastStretch(grayscaleImage, 0.40, 0.60)` ▷ Stretch with 40% lower and upper limits
DisplayImage(`fortyPercentImage`, `"40% upper & lower limit"`)
SetFigureTitle(`"Contrast Stretching"`) ▷ Set the figure title
Save `tenPercentImage` to `'Photos/contrast_stretched_image.bmp'` ▷ Save the processed image
END ContrastStretching

Figure 3: Contrast Stretching Algorithm

2.2.2. Contrast Stretching Results

2.2.2.1. Display Original Image

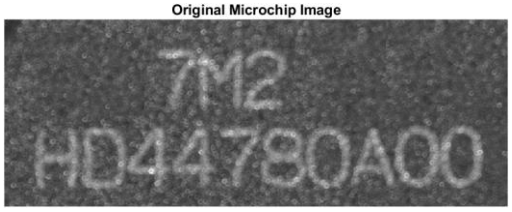


Figure 4: Display Orginal Image

The *imread()* function was used to read the image file from the specified path. The function returns the image data as a matrix, where each pixel is represented by its intensity values (for grayscale images) or color channels (for RGB images).

Following that, the *imshow()* function was utilised to display the image in a separate figure window on MATLAB. The function enables image data that is stored in a matrix format to be visualised within MATLAB.

2.2.2.2. Contrast stretching at different limits

The initial thresholds set at the upper and lower limits of 10%. This means that pixels with intensity values at the low 10% of the intensity spectrum were allocated a value of zero, making them black while pixels with intensity values above 90% of the spectrum were allocated a value of one, making them white. The resulting image is shown in Figure 5 below.

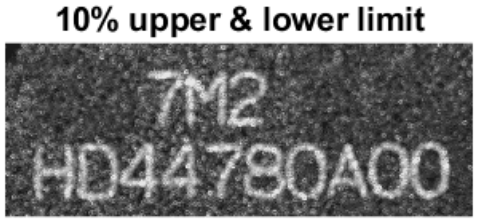


Figure 5: 10% upper and lower limit thresholds

Comparing the new image to the original image, we can see that the dynamic range of pixel intensities has been widened and the image appears more vibrant. In addition, the alphanumeric characters in the image are now clearer and more defined as the features in the image become more distinguishable.

We then increased the thresholds to 20%, 30% and 40% to evaluate the impact of raising the threshold values. The results are shown in Figure 6 below:

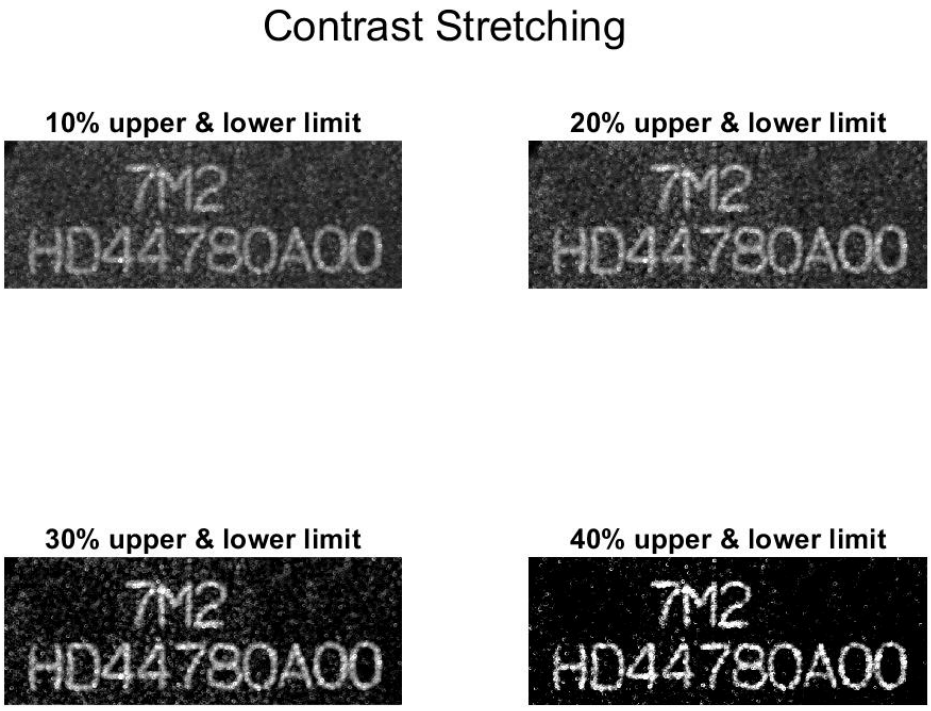


Figure 6: Comparison of 10%, 20%, 30% and 40% upper and lower limit thresholds

With the further increase in threshold values, there is a loss of details in the darker parts of the image and the lighter areas. With a lower upper threshold, the bright areas of the image are now compressed into a narrower intensity range, resulting in loss of detail in the highlights. On the

other hand, with a higher lower threshold, the dark areas are pushed to a higher minimum intensity value, resulting in loss details for the shadows. Darker regions now appear completely black which removes the texture from them. We can see that at a 40% threshold, the image background appears black instead of gray and the characters become whiter. The whitish dots that are speckled in the image have been reduced.

2.2.3. Conclusion

In Task 1, we learned how to import image files into MATLAB, storing them as matrices for display. We also explored the effects of various thresholds on contrast stretching. Our findings showed that while contrast stretching can enhance the visibility of image features and expand the dynamic range, using certain thresholds may actually have the opposite effect, diminishing clarity instead. We select the image with the 10% threshold for contrast stretching to be used in the subsequent task 2.

2.3. Brightness Thresholding

Brightness thresholding is a form of contrast enhancement that segments an image into a binary black and white image. A thresholding function maps all pixel values below a specified threshold to zero and all pixel values above this threshold to 255 for a 8-bit grayscale image. This is described in Figure 7 below where the threshold is set as r1=r2 and pixels with intensity values equal to or more than r2 are set to 255 while pixels with intensity values less than r1 are set to 0.

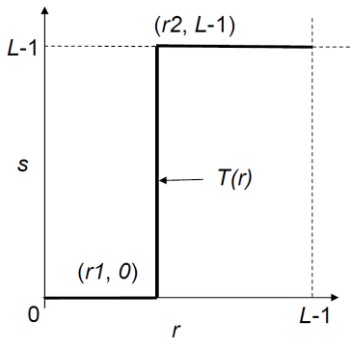


Figure 7: Brightness Thresholding

2.3.1. Brightness Thresholding Algorithm

A zero matrix of the same dimensions as the input image is created to hold the processed pixel values. A nested loop iterates through each pixel of the image, assigning intensity values of 0 or 255 based on a predefined threshold of 128 (50%). Pixels with intensity values below 128 are set to 0, while those equal to or above 128 are set to 255. This effectively segments the image into

binary values based on brightness. This was repeated for different threshold values of 64 (25%) and 191(75%)

Algorithm 2 Brightness Thresholding

```
Let microchip be the input image
Let a ← size(microchip, 1)
Let b ← size(microchip, 2)
Create a zero matrix microchip_thres of size (a, b)
for i = 1 to a do
    for j = 1 to b do
        if microchip(i, j) < 128 then
            microchip_thres(i, j) ← 0
        else
            microchip_thres(i, j) ← 255
        end if
    end for
end for
return microchip_thres
```

Figure 8: Brightness Thresholding Algorithm

2.3.2. Brightness Thresholding Results

Figure 9 below shows the results of brightness thresholding implemented with thresholds of 25%, 50% and 75%. Three binary images with varying results are produced from the algorithm.

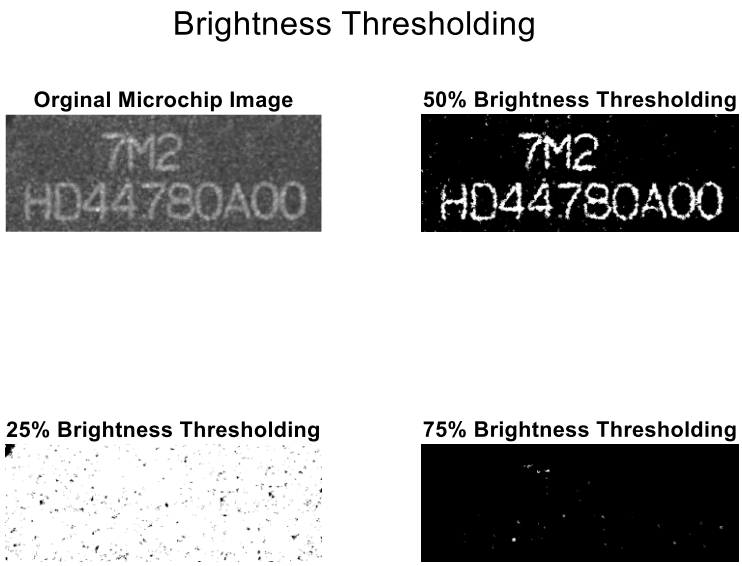


Figure 9: Brightness Thresholding for 25%, 50% and 75% thresholds

At a 50% brightness threshold, the alphanumeric characters on the microchip label are more distinctly isolated, as the thresholding enhances the brightness of the pixels corresponding to the characters. However, with a lower threshold of 25%, most pixels are assigned an intensity value of 255 (white), resulting in a complete loss of detail in the characters, as many low-intensity pixels are now white. This renders the features of the image indistinguishable.

Conversely, at a much higher threshold of 75%, the entire image appears nearly black, with most pixels in the alphanumeric characters assigned an intensity value of 0 (black). This also leads to a significant loss of detail in the image features.

2.3.3. Conclusion

Brightness thresholding can help with segmenting the characters in the microchip label by turning it into a high contrast binary image where the lighter characters are converted into white regions which standout more against the darker background which is converted to black. This may backfire if the brightness threshold is set too low or too high as the entire image may become white or black with the features being indistinguishable.

2.4. Gray Level Slicing

Gray level slicing is an image processing technique that enhances specific gray levels within an image. For gray levels outside the region of interest, they can either be preserved or be diminished to a constant, low level. The two implementations are shown in Figure 10 below.

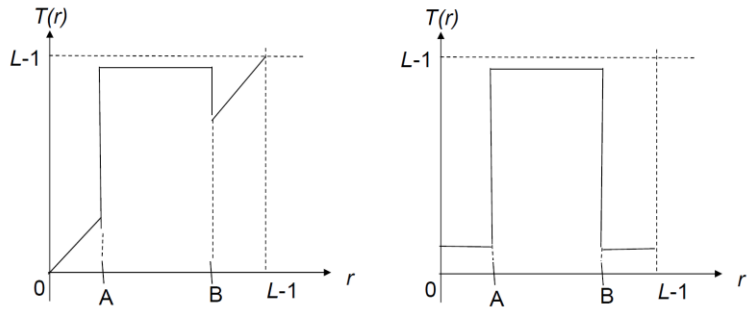


Figure 10: Two implementations of Gray Level Slicing

2.4.1. Gray Level Slicing Algorithm

The Gray Level Slicing algorithm enhances specific intensity ranges within an image to isolate features of interest. It begins by converting the input RGB image to grayscale, creating two copies of the image for processing. The algorithm then defines two threshold values, T1 and T2, which determine the intensity range to be enhanced. It iterates through each pixel in the image, setting pixels within the specified range to maximum intensity (255). For the rest of pixels outside the specified range, the pixels are preserved for one output image and set to black (0) for the second output image. This results in two processed images: one highlighting the features of interest against a dark background, and the other preserving the original background but

enhancing the specified features. The algorithm concludes by displaying the original and processed images for comparison

Algorithm 3 Gray Level Slicing

```
Initialize: Clear command window and workspace
i ← rgb2gray(imread('Photos/charact2.bmp'))
j ← double(i)
k ← double(i)
[row, col] ← size(j)
T1 ← 120
T2 ← 200
for x = 1 to row do
    for y = 1 to col do
        if (j(x, y) > T1) ∧ (j(x, y) < T2) then
            j(x, y) ← 255
            k(x, y) ← 255
        else
            j(x, y) ← i(x, y)
            k(x, y) ← 0
        end if
    end for
end for
Display Images:
figure
subplot(3, 1, 1), imshow(i), title('Original image')
subplot(3, 1, 2), imshow(uint8(j)), title('Graylevel slicing with background')
subplot(3, 1, 3), imshow(uint8(k)), title('Graylevel slicing without background')
sgtitle("Gray Level Slicing")
```

Figure 11: Gray Level Slicing Algorithm

2.4.2. Gray Level Slicing Results

Figure 12 below shows the results when T1 is set to 120 and T2 is set to 200. For the implementation of Gray level slicing with preservation of background pixel Gray intensities, the contrast of the image is enhanced with greater clarity of the alphanumeric microchip label. In the second implementation with setting the background pixels to black, the image becomes a binary black and white image and since the alphanumeric characters generally lie in the region of interest, the clarity and detail of the characters are enhanced.

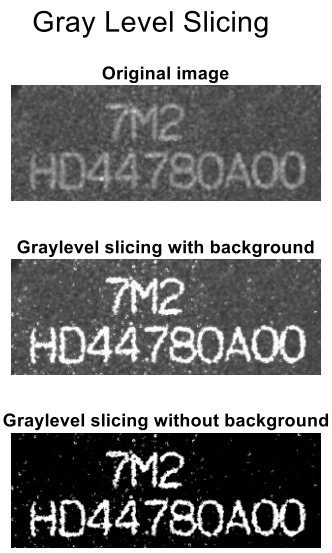


Figure 12: Gray Level Slicing for T1 = 120 and T2 = 200

2.4.3. Conclusion

Gray level slicing can help with improving the contrast of images especially if the features of interest lie within the specified intensity level band. This can be observed in the results here, where it can be observed that the label characters lie within pixel intensities of 120-200 as their contrast is significantly improved with Gray level slicing applied.

2.5. Histogram Equalization

Histogram equalization is a method of contrast adjustment that makes use of the image's histogram. It finds a gray level transformation function that generates an output image with a uniform/nearly uniform histogram. This transformation matrix is applied to the entire image and replaces each intensity of the input image with a new intensity value.

2.5.1. Histogram Equalization Algorithm

A function *histogram_equalization* was created that takes an input grayscale image and outputs an equalized image with improved contrast. The *histcounts* function calculates the histogram of the image by forming a 1D array of all pixel intensities in the image. The cumulative distribution function (CDF) is calculated by taking the cumulative sum of the histogram's intensity values (using the *cumsum* function) and dividing this by the number of elements in the image array (using the *numel* function). The CDF maps the cumulative frequency of intensities. The pixel intensities of the original image are remapped using the normalised CDF, with the CDF value for each pixel being multiplied by 255 to stretch the intensity range to [0,255]. The histograms for both the original image and the equalized image are displayed along with the images.

Algorithm Histogram Equalization

Function: `histogram_equalization(image, nbins)`
Input: `image` (grayscale image), `nbins` (number of bins)
Output: `equalized_image` (grayscale image with equalized histogram)
`hist_counts` ← **histcounts**(`image(:)`, $[0:256]$) ▷ Compute histogram with 256 bins
`cdf` ← **cumsum**(`hist_counts`) /**numel**(`image`) ▷ Compute cumulative distribution function
`equalized_image` ← **uint8**(`cdf(double(image)` $+1) \cdot 255$) ▷ Map intensities to $[0, 255]$
End Function

Main Program:
Display the original image and its histogram:
DisplayImage(`microchip`, "Original Microchip Image")
DisplayHistogram(`microchip`, "Original Microchip Image Histogram")
Perform histogram equalization:
`equalized_image` ← **histogram_equalization**(`microchip`, 256)
Display the equalized image and its histogram:
DisplayImage(`equalized_image`, "Equalized Image")
DisplayHistogram(`equalized_image`, "Equalized Image Histogram")
Add the title "Histogram Equalization" to the figure
End Program

Figure 13: Histogram Equalization Algorithm

2.5.2. Histogram Equalization Results

Histogram equalization helps to enhance contrast and visibility of the alphanumeric label on the microchip. We can see that implementation of histogram equalization yields good results for this image as the background and foreground was originally quite dark as seen in the original image histogram. It helps to highlight the detail when the histogram is spread out across the entire intensity range.

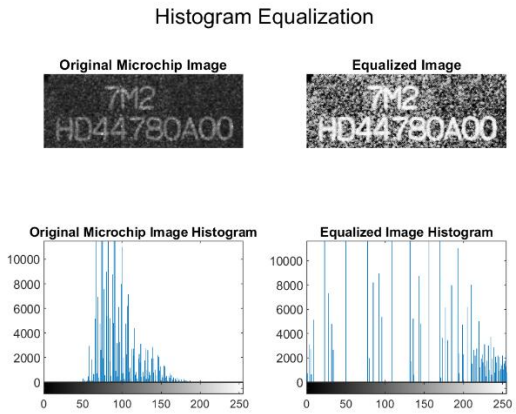


Figure 14: Histogram Equalization Results

2.5.3. Conclusion

Histogram equalization yields good results for this image as it helps to segment out the label characters quite clearly. However, it is prudent to keep in mind that it modifies the image in a global sense, so background noise may be increased and reduces the overall usability of the image. We can somewhat see this in the middle section of the image where the background has some white regions that can potentially interfere with the outline of the white characters.

3. Task 2: Implementation of a 5x5 averaging filter to image and with filters of different sizes

3.1. Introduction

The objective of task 2 was to subsequently implement a 5x5 averaging filter followed by filters of different sizes. We experimented with averaging filters of different mask sizes such as 3x3 and 9x9 to observe the effect of the mask size on the results.

The averaging mask blurs the image by calculating the average intensity of the pixels in the specified neighbourhood. It is used to suppress image noise and to help to “smooth” the image to blur and remove small irrelevant details.

For this filter implementation, we used the input image from the 10% upper and lower limit contrast stretching in task 1.

3.2. 5x5 Averaging Filter Algorithm

The algorithm processes the grayscale image using a 5x5 averaging filter. It first defines a kernel of size 5, where each element of the kernel is set such that the total sum equals 1. This averaging kernel is used to compute the average of pixel values within the window defined by the kernel, effectively blurring the image. After applying the filter, the resulting blurred image is displayed.

Algorithm Averaging Filter

1: **BEGIN** AveragingFilter
2: **Input:** `imagePath` ← `'Photos/contrast_stretched_image.bmp'`
3: **Output:** Filtered image with averaging mask applied
4: Load the image: `microchip` ← **ReadImage**(`imagePath`)
5: Display the original image with the title `"Image after contrast enhancement in task 1"`
6: Define the kernel size: `windowSize1` ← 5
7: Define the averaging kernel:

$$\texttt{kernel1} \leftarrow \frac{\textbf{ones}(\texttt{windowSize1}, \texttt{windowSize1})}{\texttt{windowSize1}^2}$$

8: Apply convolution using `conv2`:

$$\texttt{blurred\_microchip5x5} \leftarrow \textbf{uint8}\left(\textbf{conv2}(\textbf{double}(\texttt{microchip}), \texttt{kernel1}, \texttt{'same'})\right)$$

▷ Ensures the output size matches the input

9: Display the filtered image with the title `"5x5 averaging mask"`
10: **END** AveragingFilter

Figure 15: 5x5 Averaging Filter Algorithm

The 5x5 mask is a square matrix used to compute the average of pixel values in a specified neighbourhood. It consists of 25 equal values with each value set to 0.04 (1/25), with the total sum of all 25 values equalling 1. As the mask is slid over the image, the underlying pixel values are multiplied by the mask values and the product is summed to product a new pixel value that replaces the centre pixel. This is shown in equation 1.

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

3.3. Averaging Filter Results

The averaging filter is applied with a 3x3, 5x5 and 9x9 mask. Figure 16 shows the implementation of averaging filters of different sizes below.

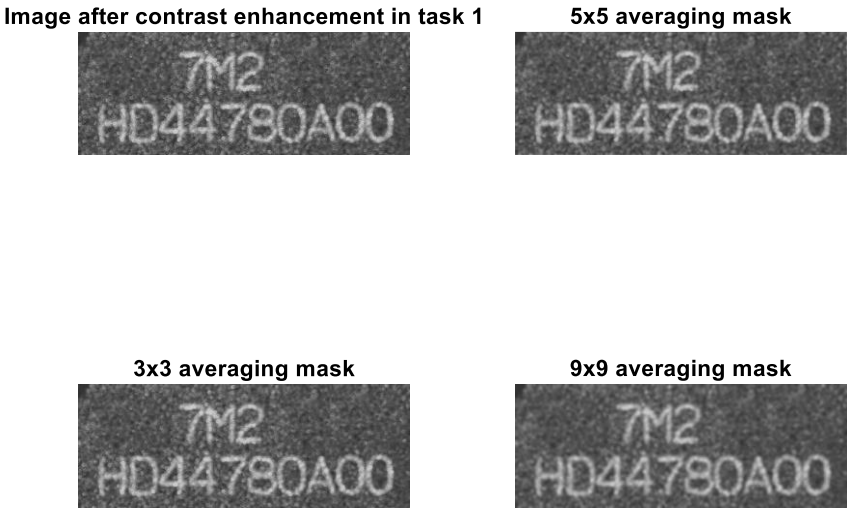


Figure 16: Averaging Filter of 3x3, 5x5 and 9x9 sizes

We can see that as the size of the mask increases, the image gets smoother and less detailed but there is a reduction in the background noise.

3.4. Conclusion

Applying an average filter can help to remove image noise but at the cost of clarity of the image due to increased blurring. Hence, in this case, a 5x5 mask appears to be reasonably effective at reducing noise without sacrificing too much image detail. We use the 5x5 average filtered image for task 3.

4. Task 3: Implementation of High-pass Filter in the Frequency Domain

4.1. Introduction

The objective of task 2 was to implement a high-pass filter on the image after performing tasks 1 and 2. There were three types of high-pass filter techniques applied to the image, namely the Ideal High-Pass Filter, the Butterworth High-Pass Filter and the Gaussian High-Pass Filter.

The image can be transformed from its spatial domain to frequency domain. The frequency domain represents the rate of which the pixel values are changing in its spatial domain. The low frequency components correspond to the smooth regions in the image whereas the high frequency components correspond to the edges. The three high-pass filter techniques are applied to the image in the frequency domain and the images are then converted back to the spatial domain for display on the screen.

4.2. High-Pass Filter Algorithms

The High-Pass Filter algorithm below outlines the process for filtering an image in the frequency domain using high-pass filters. Initially, the algorithm pads the original image dimensions (MxN) to create a larger image matrix to allow for more comprehensive frequency analysis. It then constructs a frequency grid to facilitate the calculation of the Discrete Fourier Transform (DFT), which converts the padded image into the frequency domain. Three types of high-pass filter masks—Ideal, Butterworth, and Gaussian—are generated based on a specified cutoff frequency. The algorithm then applies these masks to the DFT of the image through element-wise multiplication, effectively filtering out low-frequency components. After filtering, the algorithm computes the inverse DFT to transform the image back into the spatial domain, while also undoing any centring adjustments made during the frequency transformation. Finally, the algorithm extracts the relevant sections of the filtered images

Algorithm: High-Pass Filter in the Frequency Domain

Input: Image matrix `microchip`, dimensions M, N
Output: Filtered images gI, gB, gG
$P \leftarrow 2 \times M$
$Q \leftarrow 2 \times N$
`microchip_p` = zeros(P, Q)
`microchip_p(1:M, 1:N)` = `microchip`
Create frequency grid:
$[x, y] \leftarrow$ meshgrid$(0 : Q-1, 0 : P-1)$
`microchip_p` = `microchip_p` $\times (-1)^{(x+y)}$
F`microchip_p` $\leftarrow$ fft2(`microchip_p`)
$H_I \leftarrow$ createFilterIdeal(P, Q)
$H_B \leftarrow$ createFilterButterworth(P, Q)
$H_G \leftarrow$ createFilterGaussian(P, Q)
$G_I \leftarrow H_I \times F$`microchip_p`
$G_B \leftarrow H_B \times F$`microchip_p`
$G_G \leftarrow H_G \times F$`microchip_p`
$g_{pI} \leftarrow$ real(ifft2(G_I))
$g_{pB} \leftarrow$ real(ifft2(G_B))
$g_{pG} \leftarrow$ real(ifft2(G_G))
$g_{pI} \leftarrow g_{pI} \times ((-1)^{(x+y)})$
$g_{pB} \leftarrow g_{pB} \times ((-1)^{(x+y)})$
$g_{pG} \leftarrow g_{pG} \times ((-1)^{(x+y)})$
$g_I \leftarrow g_{pI}(1 : M, 1 : N)$
$g_B \leftarrow g_{pB}(1 : M, 1 : N)$
$g_G \leftarrow g_{pG}(1 : M, 1 : N)$

Figure 17: High-Pass Filter Algorithm

The following sections will explain the algorithms for each of the three high-pass filters.

4.2.1. Ideal High-Pass Filter

The Ideal High-Pass Filter function first defines the filter dimensions P and Q and generates a frequency grid using the *meshgrid* function. The grid calculates the distance D from the origin within the frequency domain by using the formula $\mathrm{D} = \sqrt{\mathrm{u}^2 + \mathrm{v}^2}$. A cutoff frequency D_0 is specified to sort the low and high frequencies. In this case the D_0 value specified is 15. The filter mask H is then created by evaluating the condition of double (D >D0); if the distance is more than D_0, the binary mask then assigns a value of 1. On the other hand, if the distance is lower than the cutoff value, a value of 0 is assigned instead. As per the above algorithm, the created mask H is applied to the Fourier-transformed image through element-wise multiplication to produce the image with ideal high-pass filter.

Function: createFilterIdeal
Input: Integers P, Q
Output: Matrix H

1: **Create an Ideal High-pass filter**
2: $[u, v] \leftarrow \text{meshgrid}(-\frac{Q}{2} : \frac{Q}{2} - 1, -\frac{P}{2} : \frac{P}{2} - 1)$
3: $D_0 \leftarrow 15$ {Cut-off frequency}
4: $D \leftarrow \sqrt{u^2 + v^2}$
5: $H \leftarrow \text{double}(D > D_0)$ {1 for high frequencies, 0 for low frequencies}

Figure 18: High-Pass Filter Function

4.2.2. High-Pass Butterworth Filter

The initial steps are the same as the ideal high-pass filter where the dimensions P and Q are defined and a frequency grid is generated using the *meshgrid* function. The grid calculates the distance D from the origin within the frequency domain by using the formula $\mathrm{D} = \sqrt{\mathrm{u}^2 + \mathrm{v}^2}$. A cutoff frequency D_0 is specified to sort the low and high frequencies. In this case the D_0 value specified is 15. The order of the filter, n, is set to 4 and this determines the steepness of the filter transition. The Butterworth filter matrix H is calculated using the following equation:

$$H = 1 - \frac{1}{1 + \left(\frac{D}{D_0}\right)^{2n}} \tag{2}$$

As per the above algorithm, the resulting filter mask H is applied to the Fourier-transformed image through element-wise multiplication to produce the image with high-pass Butterworth filter.

Function: createFilterButterworth
Input: Integers P, Q
Output: Matrix H

1: **Create a High-pass Butterworth filter**
2: $[u, v] \leftarrow \text{meshgrid}(-\frac{Q}{2} : \frac{Q}{2} - 1, -\frac{P}{2} : \frac{P}{2} - 1)$
3: $D_0 \leftarrow 15$ {Cut-off frequency}
4: $D \leftarrow \sqrt{u^2 + v^2}$
5: $n \leftarrow 4$ {Order of filter}
6: $H \leftarrow 1 - \frac{1}{1+\left(\frac{D}{D_0}\right)^{2n}}$
7: $H(D == 0) \leftarrow 0$ {Handle zero distance case to avoid division by zero}

Figure 19: High-Pass Butterworth Filter Function

4.2.3. High-Pass Gaussian Filter

The initial steps are the same as the ideal high-pass filter where the dimensions P and Q are defined and a frequency grid is generated using the *meshgrid* function. The grid calculates the distance D from the origin within the frequency domain by using the formula $\mathrm{D} = \sqrt{\mathrm{u}^2 + \mathrm{v}^2}$. A cutoff frequency D_0 is specified to sort the low and high frequencies. In this case the D_0 value specified is 15. The Gaussian high-pass filter matrix H is then computed using the Gaussian function formula:

$$H = 1 - e^{-\frac{D^2}{2D_0^2}} \tag{3}$$

As per the above algorithm, the resulting filter mask H is applied to the Fourier-transformed image through element-wise multiplication to produce the image with high-pass Gaussian filter.

Function: createFilterGaussian
Input: Integers P, Q
Output: Matrix H

1: **Create a Gaussian High-pass filter**
2: $[u, v] \leftarrow \text{meshgrid}(-\frac{Q}{2} : \frac{Q}{2} - 1, -\frac{P}{2} : \frac{P}{2} - 1)$
3: $D_0 \leftarrow 15$ {Cut-off frequency}
4: $D \leftarrow \sqrt{u^2 + v^2}$
5: $H \leftarrow 1 - \exp\left(-\frac{D^2}{2 \cdot (D_0^2)}\right)$ {Gaussian High-pass filter}

Figure 20: High-Pass Gaussian Filter Function

4.3. High-Pass Filter Results

The below image shows the results for the 3 types of high-pass filter with cut-off frequencies set to 5, 10 and 15 respectively.

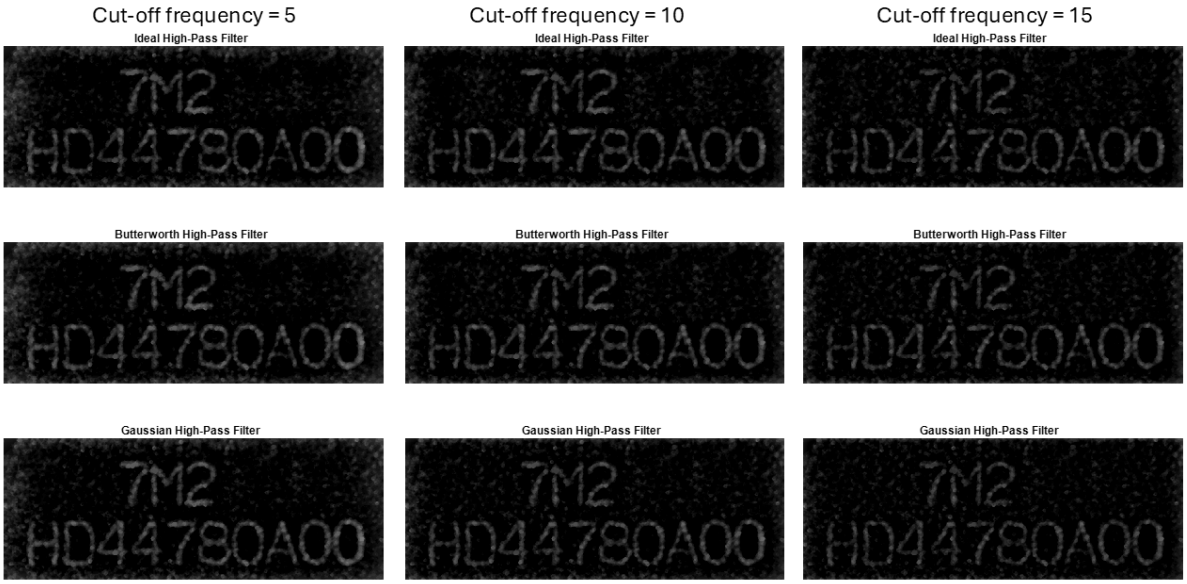


Figure 21: High-Pass Filter Results

4.4. Conclusion

The results indicate that a cut-off frequency of 10 strikes an effective balance between preserving details and reducing noise. However, the letter "A" shows some loss of detail in its horizontal line at both cut-off frequencies of 10 and 15. Similarly, the number "8" exhibits a loss of detail in its central region at a cut-off frequency of 15. Overall, the performance of the three filters at the cut-off frequency of 10 appears largely interchangeable. Therefore, for Task 4, we will proceed with the image processed using the Butterworth High-Pass Filter.

5. Task 4: Create a sub-image that includes the middle line – HD44780A00

5.1. Introduction

The objective of task 4 was to create a sub-image from the input image from task 3 that includes the middle line – HD44780A00. The input image would be cut into smaller components to isolate the portion that contains the "HD44780A00" line.

5.2. Image Cropping Algorithm

As each letter is the same size with uniform spacing between the letters, we simply need to split the image into two parts and take the lower half of the image before carrying out an equal division of each of the characters. The detailed steps are as follows:

1. By using the *imtool* function in the 'Image Processing Toolbox', we can get the margin of position of letter 'H' and '0', like following shown:

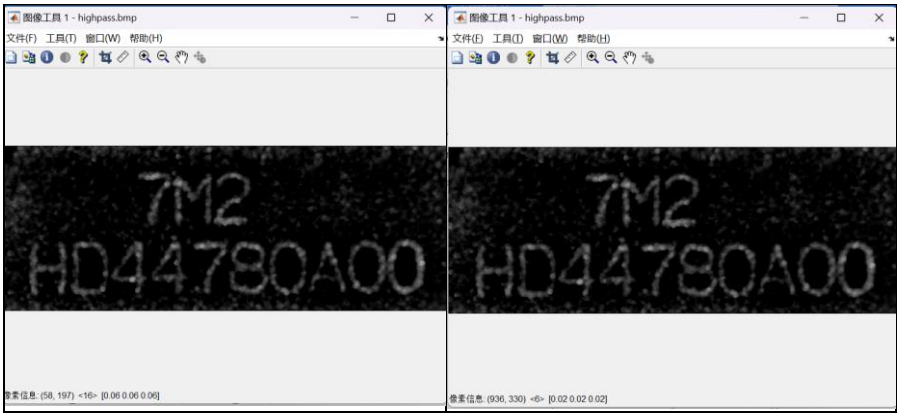


Figure 22 Look for the coordinates of the edge position

The coordinates shown in the bottom left corner of the left image represent the coordinates of the top left corner of the letter H (i.e. (56,206)), while the right image displays the coordinates of the bottom right corner of the number 0 (i.e. (936,330)). To facilitate the preprocessing of the following task 9, we selected the rectangle = [60,206,945-60,127].

2. Then we can set the '*rect*' to extract the image, and divide the extracted image into 10 equal parts, drawing lines in between for display. The pseudocode and result are as follows:

Algorithm 1 Function: Create sub-image

Input: High-Pass Filter Result
Output: A sub-image that includes the middle line – HD44780A00
rect = [60, 206, 945-60, 127];
Extract the image using **`imcrop(img, rect)`**
Display the extracted image
Get the height and width of the extracted image using **`size()`**
Compute **`segmentWidth = width / 10`**
for $i = 1$ to 9 **do**
    Calculate $x = i \times segmentWidth$
    Draw a vertical line at x from 1 to $height$
end for
Release the displayed image

Figure 23 Pseudocode for creating the sub-image

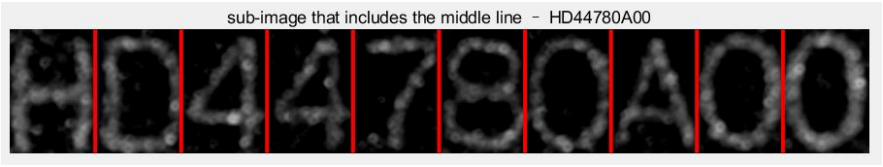


Figure 24 Sub-image

6. Task 5: Convert the sub-image into a binary image

6.1. Introduction

The objective of task 5 is convert the sub-image obtained from task 4 into a binary black and white image.

6.2. Image Binarization Algorithm

Image Binarization is the process of setting the grayscale values of pixels in an image to either 0 or 255, producing a clear black-and-white effect across the entire image. In a binary image, each pixel has only two possible values: either pure black or pure white.

Binary images allow for a more straightforward analysis of letter shapes and contours, facilitating recognition tasks. There are various methods for binarization, with the most common being thresholding. This approach sets the grayscale values of pixels greater than a specified threshold to the maximum grayscale value (255) and those less than this value to the minimum (0), achieving binarization. Threshold selection can be based on methods such as the Mean Thresholding Method, 2-Mode Method, Otsu Method, and Iterative Thresholding Method, among others.

Here, Mean Thresholding Method and 2-Mode Method didn't work well on most tasks, so we compare the effectiveness of Otsu Method (this method is the default method in MATLAB

'*graythresh*' function) and Iterative Thresholding Method. The detailed description of the code flow and experimental results will be as follows:

The pseudocode for the Iterative Thresholding Method is as follows:

Algorithm 1 Iterative Thresholding Method

Input: Image I
Output: Binary Image J
$T_0 \leftarrow 0.01$ ▷ Convergence threshold
$T_1 \leftarrow \frac{\min(I)+\max(I)}{2}$ ▷ Initial threshold
repeat
  $r_1 \leftarrow \{\text{pixels in } I \mid I > T_1\}$ ▷ Pixels brightness greater than T_1
  $r_2 \leftarrow \{\text{pixels in } I \mid I \leq T_1\}$ ▷ Pixels brightness less than or equal to T_1
  $T_2 \leftarrow \frac{\text{mean}(I[r_1])+\text{mean}(I[r_2])}{2}$ ▷ Update threshold
until $|T_2 - T_1| < T_0$ ▷ Check for convergence
$J \leftarrow \text{imbinarize}(I, T_2)$ ▷ Binary image

Figure 25 Iterative Thresholding Method

The pseudocode for Ostu method is as follows:

Algorithm 2 Otsu's Thresholding Method

Input: Grayscale Image I
Output: Binary Image I_{bw}
Compute the histogram H of the image I
Compute the total number of pixels $N = \sum H$
Initialize: $maxVar \leftarrow 0$, $T \leftarrow 0$
for $t = 0$ to 255 **do**
  Calculate $p_1(t) = \frac{\sum_{i=0}^{t} H[i]}{N}$ ▷ Probability of class 1
  Calculate $p_2(t) = \frac{\sum_{i=t+1}^{255} H[i]}{N}$ ▷ Probability of class 2
  if $p_1(t) > 0$ and $p_2(t) > 0$ **then**
    Calculate the means:
    $\mu_1(t) = \frac{\sum_{i=0}^{t} i \cdot H[i]}{\sum_{i=0}^{t} H[i]}$ ▷ Mean of class 1
    $\mu_2(t) = \frac{\sum_{i=t+1}^{255} i \cdot H[i]}{\sum_{i=t+1}^{255} H[i]}$ ▷ Mean of class 2
    Calculate the between-class variance:
    $var_b(t) = p_1(t) \cdot p_2(t) \cdot (\mu_1(t) - \mu_2(t))^2$
    if $var_b(t) > maxVar$ **then**
      $maxVar \leftarrow var_b(t)$
      $T \leftarrow t$ ▷ Update threshold
    end if
  end if
end for
$I_{bw} \leftarrow \text{imbinarize}(I, T)$ ▷ Thresholding

Figure 26 Otsu's Thresholding Method

The image below shows a comparison of the results from the two methods, where the first and third columns are the binary images obtained using the Otsu method, while the second and fourth columns are the binary images produced by the Iterative Thresholding Method. Currently, they appear quite similar. However, since many areas of the characters are not well connected, we decided to eliminate the noise first through usage of the `*bwareaopen(J, 1000)*` function, before applying multiple expansion corrosion operations to fill in these gaps.

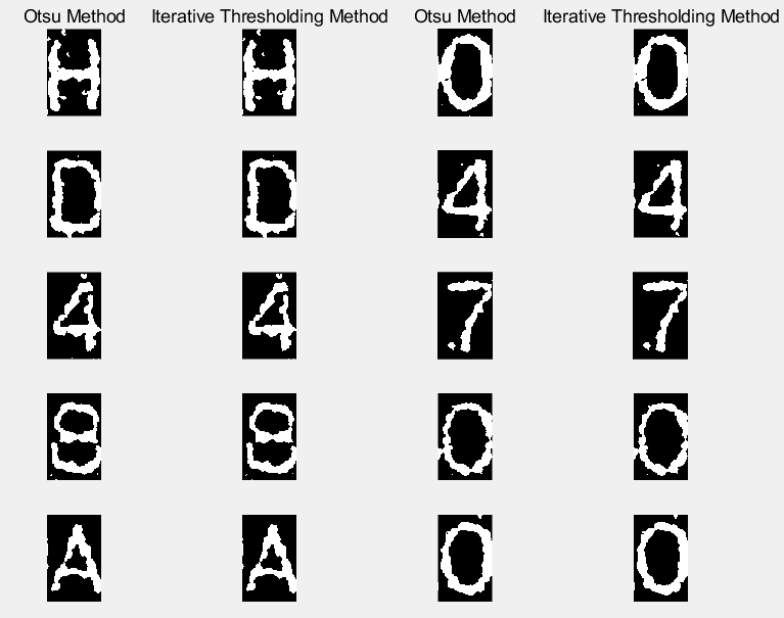


Figure 27 Comparison before closed operation

The results obtained after applying multiple expansion and corrosion operations are shown below. Overall, the effects are similar. Codes for closed operation are as follows:

```
I_bw = bwareaopen(I_bw,1000);
se1 = strel('diamond', 2);
process = imdilate(I_bw, se1);
process = imdilate(process, se1);
process = imerode(process,se1);
process = imdilate(process,se1);
process = imdilate(process,se1);
I_bw = imerode(process,se1);
```

```
J = bwareaopen(J,1000);
se1 = strel('diamond', 2);
process = imdilate(J, se1);
process = imdilate(process, se1);
process = imerode(process,se1);
process = imdilate(process, se1);
process = imdilate(process, se1);
J = imerode(process,se1);
```

Figure 28 Code for closed operation

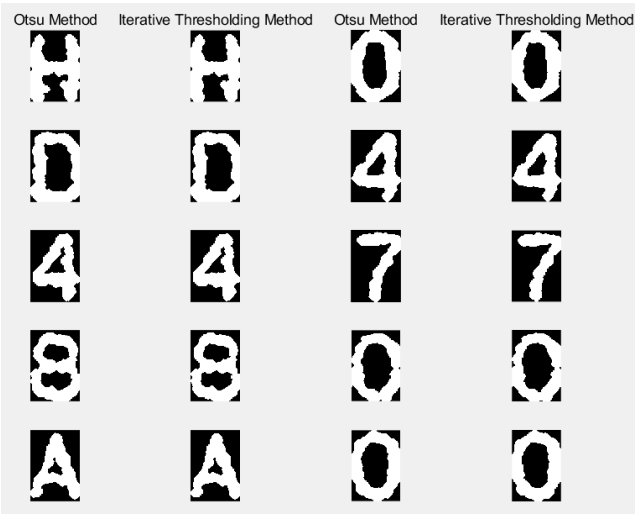


Figure 29 Comparison after closed operation

Finally, we choose Otsu method, and the result is shown in the figure below.

Figure 30 result in Task 5

7. Task 6 Determine the outline(s) of characters in the image

7.1 Introduction

In this section of the report, our objective is to achieve edge detection on a given image by testing various algorithms. This section will cover the details of the algorithms we used, including Sobel and Laplacian, as well as the implementation process and the results of each method.

7.2 Edge detection algorithms

In the given image, due to the excessive amount of noise, it is necessary to first perform noise reduction. To achieve this, the image goes through contrast stretching, erosion, and gaussian blurring as shown in the pseudo code:

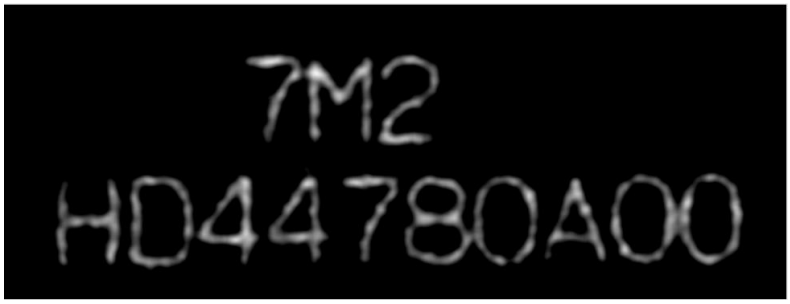


Figure 31 Pre-processed image for edge detection

Function: Pre-processing of edge detection image

Input: Grayscale image $I(x, y)$
Output: Processed image $I_{\text{final}}(x, y)$
Contrast Adjustment:
$I_{\text{adjusted}}(x, y) = \frac{I(x,y)-0.3}{0.6-0.3} \times (1 - 0) + 0$
Erosion:
$I_{\text{eroded}}(x, y) = \min_{(i,j)\in\text{disk}(3)}\{I_{\text{adjusted}}(x + i, y + j)\}$
Gaussian Blurring:
Define Gaussian kernel $G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right), \quad \sigma = 15$
Apply convolution: $I_{\text{blurred}}(x, y) = \sum_{i=-5}^{5} \sum_{j=-5}^{5} I_{\text{eroded}}(x + i, y + j) \cdot G(i, j)$
Final Contrast Adjustment:
$I_{\text{final}}(x, y) = \frac{I_{\text{blurred}}(x,y)-0.25}{1.0-0.25} \times (1 - 0) + 0$

Figure 32 Preprocessing of Edge Detection Images

7.2.1 Laplacian edge detection

The Laplacian edge detection method is a second-order derivative approach used to highlight regions of rapid intensity change in an image, which typically correspond to edges. This method calculates the second derivative of the pixel intensity values, identifying points where there is a rapid transition in brightness. Unlike first-order derivative methods like Sobel, which detect edges based on gradient direction, the Laplacian operator is isotropic, meaning it responds equally to edges in all directions.

After preprocessing the images from the previous section, the Laplacian edge detection is achieved using a convolution with a Laplacian kernel that approximates the second derivative. In this case, we used the below kernel:

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

Then to detect edges, we convolve the image with the kernel. The result image can be written as:

$$\boldsymbol{I_{edge}} = \boldsymbol{I_{processed}} * \boldsymbol{K} \tag{4}$$

After convolution, a binary filter is used to obtain the result image as shown below:

Figure 33 Edge detection results using Laplacian Kernel

7.2.2 Sobel edge detection

The Sobel edge detection method is a first-order derivative technique used to identify edges by calculating the gradient of image intensity. It uses two 3x3 convolution kernels—one for detecting edges in the horizontal direction and one for the vertical direction:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & 2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The Sobel operator convolves these kernels with the image to calculate gradients G_x and G_y, representing intensity changes along the x and y axes, respectively. The magnitude of the gradient at each pixel, which highlights edges, is calculated as:

$$G = \sqrt{G_x^2 + G_y^2} \tag{5}$$

The gradient then goes through a binary filter to get the final edge detection results:

Figure 34 Edge detection using Sobel edge detection

8. Task 7: Segment the image to separate and label the different characters as clearly as possible

From task 6, we learnt that the Laplacian edge detection method produces a clearer image with non-broken detected edges. Hence, we selected the image obtained from the Laplacian method to help to segment the original Figure 1 image.

In summary, we utilized connected component detection to identify the position of each character and performed bounding box extraction. First, small noise areas were removed, similar to Task 5, using the `*bwareaopen(I, 800)*` function. Then, the bounding boxes were obtained. However, some of the resulting bounding boxes might encompass two characters, so any bounding box exceeding a certain width needs to be split in half. The pseudocode code is as follows:

Algorithm 1 Character Segmentation and Extraction

Input: Binary image I, Original image O, width threshold $W_{\text{threshold}} = 100$
Output: Cropped character regions saved as images
$I \leftarrow \text{bwareaopen}(I, 800)$ ▷ Remove small noise areas
$CC \leftarrow \text{bwconncomp}(I, 8)$ ▷ Connected component detection
$stats \leftarrow \text{regionprops}(CC, 'BoundingBox')$ ▷ Compute bounding boxes
$total\_subplots \leftarrow 0$
for each region k in $stats$ **do**
    $bbox \leftarrow \text{stats}(k).BoundingBox$
    $width \leftarrow bbox(3)$
    if $width > W_{\text{threshold}}$ **then**
        $total\_subplots \leftarrow total\_subplots + 2$ ▷ Split wide regions into two
    else
        $total\_subplots \leftarrow total\_subplots + 1$ ▷ Count single region
    end if
end for
$rows \leftarrow \lceil \text{total\_subplots}/4 \rceil$
Create output folder if not exists
$subplot\_idx \leftarrow 1$
for each region k in $stats$ **do**
    $bbox \leftarrow \text{stats}(k).BoundingBox$
    $width \leftarrow bbox(3)$
    $height \leftarrow bbox(4)$
    if $width > W_{\text{threshold}}$ **then**
        $mid\_x \leftarrow bbox(1) + width/2$
        $bbox1 \leftarrow [bbox(1), bbox(2), width/2, height]$
        $bbox2 \leftarrow [mid\_x, bbox(2), width/2, height]$
        Crop O using $bbox1$ and save as $region\_k\_part1$
        Crop O using $bbox2$ and save as $region\_k\_part2$
        Display $region\_k\_part1$ and $region\_k\_part2$ in subplot
    else
        Crop O using $bbox$ and save as $region\_k$
        Display $region\_k$ in subplot
    end if
end for

Figure 35 Pseudocode for figure 1 segment

The specific picture obtained is shown in the following figure:

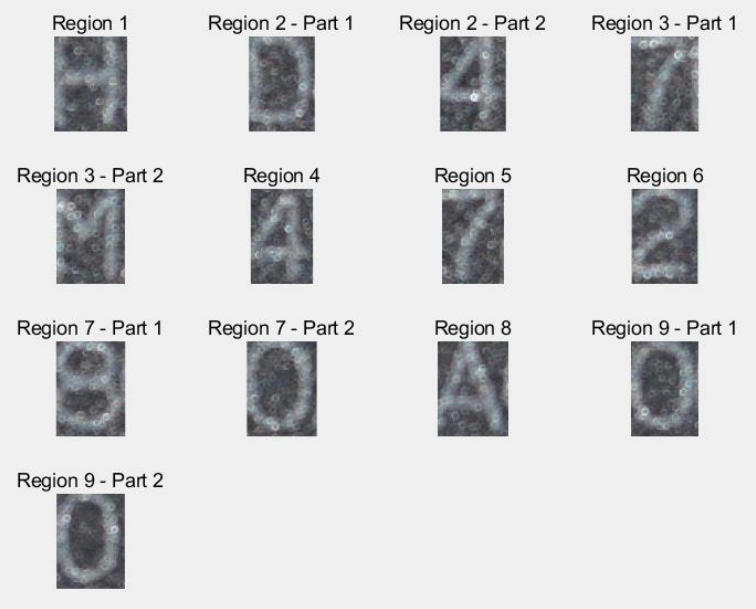


Figure 36 Different original sub-pictures

9. Task 8: Sub-task 1 - Design a CNN to classify each character in Image 1

9.1. Introduction

For task 8 sub task 1, a Convolutional Neural Network (CNN) was designed with the use of MATLAB Deep Learning toolbox to classify each of the letters in image 1. The dataset p_dataset_26 contains the seven characters 'H', 'D', 'A', '8', '7', '4', '0'. From this, it can be assumed that the task is to classify the characters from the 'HD44780A00' line from the image and not the '7M2' line as there was no data provided for it.

9.2. Overview of the Convolutional Neural Network

Section 9.2 will describe the basic overview of the CNN and how it is structured.

At the basic form, a CNN is composed of an input layer, an output layer, and many hidden layers sandwiched between.

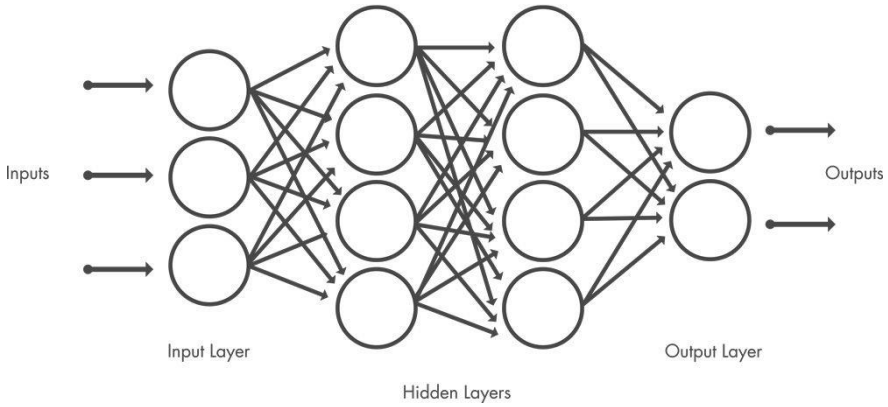


Figure 37: Neural Network Architecture [1]

The input layer captures the data from the image input and passes it onto the hidden layers with the necessary formatting. In this case, we have processed the image to grayscale from RGB and resize the image to match the specified dimensions of 128x128 pixels. Pixel values are also normalized to be centred around 0, to standardize input data for more efficient learning.

The hidden layers consist of the convolution, ReLU, pooling, batch normalization, fully connected and dropout layers. We followed the structure of AlexNet [2], a popular CNN architecture that was introduced back in 2012 by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It was the winner of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by achieving state-of-the-art performance in image classification.

The hidden layer structure in our CNN consists of 19 layers, combined with 5 convolutional blocks and 2 fully connected layers. The details for each layer are shown below:

1. 1st Convolutional Block
    a. Convolutional layer 1

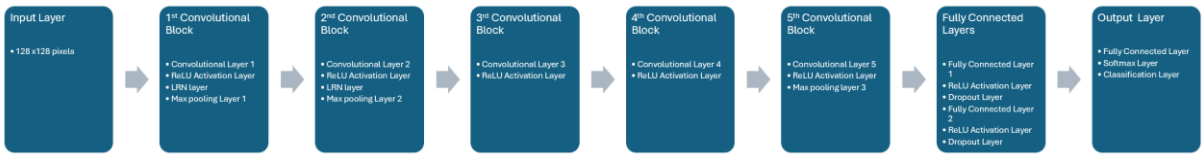
        i. Explanation: Convolution layers help to process the input images through multiple filters with each of them being designed to activate and detect distinct features inside the images. This is done using the ***convolution2dLayer*** function with applies convolution operations to the input data by sliding kernels over the input images.
        ii. Set to 96 filters of size 11x11, stride of 4 (step size for traversing the input vertically and horizontally) and 'same' padding – adds padding to ensure that the output feature map has the same spatial dimensions as the input feature map.
    b. ReLU Activation Layer
        i. The rectified linear unit (ReLU) layer speeds up and improves the training though a conversion of all negative values to zero while retaining positive values. This selective transfer of features is known as activation as only the "activated features" are passed on to the next layer. Each convolution. The ***reluLayer*** function applies the Rectified Linear Unit activation function to accomplish this.
    c. Local Response Normalization (LRN) Layer
        i. This normalizes the activations of neurons across neighbouring feature maps
        ii. The ***crossChannelNormalizationLayer*** function accomplishes this task
    d. Max-Pooling Layer 1
        i. The pooling layer reduces the dimensionality of the data through non-linear downsampling, which minimises the number of parameters that are learnt by the network.
        ii. The ***maxPooling2dLayer*** function follows the ReLu layers and downsamples to a window size of 3x3 and a 'Stride' of 2.
2. 2nd Convolution Block
    a. Convolutional Layer 2
        i. 256 filters, filter size of 5x5 and 'same' padding
    b. ReLU Activation Layer
    c. Local Response Normalization (LRN) Layer
    d. Max-Pooling Layer 2
        i. Window size of 3x3 and a 'Stride' of 2
3. 3rd Convolutional Block
    a. Convolutional Layer 3
        i. 384 filters, filter size of 3x3 and 'same' padding
    b. ReLU Activation Layer
4. 4th Convolutional Block
    a. Convolutional Layer 4
        i. 384 filters, filter size of 3x3 and 'same' padding
    b. ReLU Activation Layer
5. 5th Convolutional Block
    a. Convolutional Layer 5
        i. 256 filters, filter size of 3x3 and 'same' padding
    b. ReLU Activation Layer

c. Max-Pooling Layer 3
    i. Window size 3x3 and a 'Stride' of 2

6. Fully Connected Layers
    a. Fully Connected Layer 1
        i. Fully connected layers conduct the final stages of processing after feature extraction by connecting each neuron in the previous layer to each neuron in the current layer. There are two fully connected layers added at the end of the network.
        ii. The ***fullyConnectedLayer*** function specifies a layer with the number of neurons, in this case 4096 neurons are specified
    b. ReLU Activation Layer
    c. Dropout Layer
        i. Dropout layers help to reduce overfitting by setting a certain amount of neurons to zero during each training iteration. It prevents the model from becoming too dependent on any single neuron.
        ii. The ***dropoutLayer*** function is used and a dropout rate of 0.5 is set, meaning 50% of the neurons in this layer are dropped out for each iteration of training.
    d. Fully Connected Layer 2 – 4096 neurons specified
    e. ReLU Activation Layer
    f. Dropout Layer – dropout rate of 0.5 specified

The output layer will then produce the final classification prediction for each input image. It consists of a fully connected layer, a softmax layer and a classification layer. The fully connected layer is applied with the ***fullyConnectedLayer(numel(unique(imds.Labels)))*** function and this counts the number of unique labels within the dataset, which is 7 in this case. The ***softmaxLayer*** function normalizes the output of the fully connected layer to generate the output values as ***softmaxLayer*** probability values. Lastly, the ***classificationLayer*** function helps to compute the cross-entropy loss, measuring the difference between the predicted probabilities and true labels. The network minimizes this loss to improve accuracy. When an input image is passed to the network, the classification layer would select the class with the highest probability as the predicted label for the input.

The below figure summarises the 'AlexNet' style of architecture that was used.



9.2.1. CNN PseudoCode

The below image shows the pseudocode used for training the CNN with MATLAB Deep Learning Toolbox. It shows how the initial hyperparameters are set up and initialized, with a logging directory setup. Following that, the dataset for the training and test datasets are loaded and preprocessed to size them to fixed dimensions with normalization applied. The training data is

also augmented with rotations, translations and scaling randomly applied. This data augmentations provide the following benefits to training the CNN:

- Mitigate overfitting by allowing the network to effectively train on new variations of the data at each epoch to minimise the chance of the network memorizing specific training examples
- Generalization to unseen data (in this case it would be the characters from charact2.bmp) where the random transformations helps the CNN to better generalize such uncertainties.
- Effective use of limited data where the data augmentation artificially increases the number of training samples beyond what was provided in p_dataset_26.

The network architecture is defined as per the previous section and the training options are specified using stochastic gradient descent (SGDM) with an initial learning rate of 0.01 and 30 epochs. The network is trained and then evaluated using the test dataset before being saved in the stated file directory.

AlexNet-like CNN for Character Classification

Initialize: Set image dimension, learning rate, epochs, batch size, and momentum.
Create log directory to save training results.
Load Dataset:
Define paths for train and test datasets based on the current directory.
Load train and test images using `imageDatastore`.
Apply preprocessing to resize images to fixed dimensions and normalize pixel values.
Data Augmentation:
Apply random rotations, translations, and scaling to training images.
Define Network Architecture:
Input layer with image dimensions and normalization.
Five convolutional blocks with ReLU, optional LRN layers, and max-pooling:
- Conv1: 11×11 filters, 96 channels, stride 4.
- Conv2: 5×5 filters, 256 channels.
- Conv3-5: 3×3 filters with 384 or 256 channels.
Fully connected layers:
- Two hidden layers with 4096 units and dropout.
Output layer with number of classes and softmax activation.
Set Training Options:
Use stochastic gradient descent with momentum (SGDM), initial learning rate 0.01, and 30 epochs.
Shuffle data and enable GPU for execution.
Train the Network:
Train the model using the augmented training dataset.
Evaluate the Network:
Classify test data and compute accuracy.
Display a confusion matrix.
Save Results:
Save the trained network and predictions.
Save hyperparameters and dataset options in JSON format.

Figure 38: Pseudocode for AlexNet CNN

9.2.2. Results from Trained CNN

We ran the MATLAB code using the above pseudocode structure and obtained the following results:

```
>> AlexNet_task_8_subtask_1_V3_withToolBox
Initializing input data normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:04 |       14.84% |       1.9467 |          0.0100 |
|       2 |          50 |       00:00:13 |       20.31% |       1.9381 |          0.0100 |
|       3 |         100 |       00:00:21 |       51.56% |       1.4243 |          0.0100 |
|       4 |         150 |       00:00:30 |       82.03% |       0.4825 |          0.0100 |
|       5 |         200 |       00:00:38 |       94.53% |       0.1483 |          0.0100 |
|       7 |         250 |       00:00:46 |       98.44% |       0.0975 |          0.0100 |
|       8 |         300 |       00:00:55 |       97.66% |       0.1113 |          0.0100 |
|       9 |         350 |       00:01:03 |       97.66% |       0.0889 |          0.0100 |
|      10 |         400 |       00:01:11 |       97.66% |       0.0863 |          0.0100 |
|      11 |         450 |       00:01:20 |      100.00% |       0.0058 |          0.0100 |
|      13 |         500 |       00:01:28 |       99.22% |       0.0285 |          0.0100 |
|      14 |         550 |       00:01:36 |       98.44% |       0.0588 |          0.0100 |
|      15 |         600 |       00:01:45 |      100.00% |       0.0066 |          0.0100 |
|      16 |         650 |       00:01:53 |       96.88% |       0.0795 |          0.0100 |
|      18 |         700 |       00:02:01 |       99.22% |       0.0180 |          0.0100 |
|      19 |         750 |       00:02:10 |       99.22% |       0.0181 |          0.0100 |
|      20 |         800 |       00:02:18 |       98.44% |       0.0205 |          0.0100 |
|      21 |         850 |       00:02:26 |       99.22% |       0.0206 |          0.0100 |
|      22 |         900 |       00:02:34 |      100.00% |       0.0100 |          0.0100 |
|      24 |         950 |       00:02:43 |      100.00% |       0.0210 |          0.0100 |
|      25 |        1000 |       00:02:51 |       99.22% |       0.0143 |          0.0100 |
|      26 |        1050 |       00:02:59 |       99.22% |       0.0322 |          0.0100 |
|      27 |        1100 |       00:03:08 |      100.00% |       0.0093 |          0.0100 |
|      29 |        1150 |       00:03:16 |       99.22% |       0.0128 |          0.0100 |
|      30 |        1200 |       00:03:24 |       99.22% |       0.0304 |          0.0100 |
|      30 |        1230 |       00:03:29 |       99.22% |       0.0536 |          0.0100 |
|=========================================================================================|
Training finished: Max epochs completed.
Test accuracy: 97.08%
```

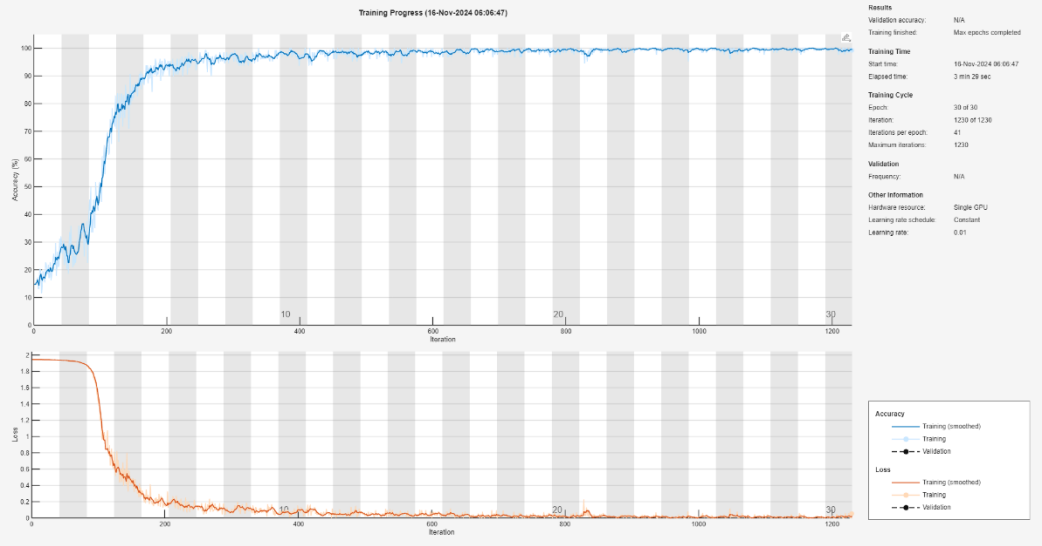
Figure 39: AlexNet CNN results

The results from the training showed a test accuracy of 97.08% for the validation dataset. This is a good result as it means almost all of the validation dataset images were predicted correctly. The hyperparameters were tuned as shown in the below table:

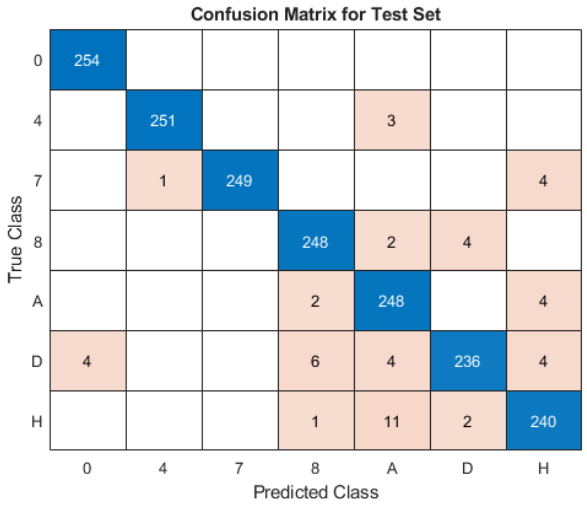
Hyperparameter	**Value**
Input Size	96
Conv1 Filter Size	4
Conv1 Number of Filters	5x5
Conv1 Stride	256
Conv2 Filter Size	3x3
Conv3 Number of Filters	384
FC1 Number of Neurons	4096
Dropout Rate	0.5
Optimizer	SGDM
Learning Rate	0.01
L2 Regularization	0.0005
Number of Epochs	30
Mini-Batch Size	128
Data Augmentation	Rotation [−25°,25°], Translation [±1%]

Figure 40: CNN Hyperparameters

The plot below shows the progress in accuracy and the decline in the Loss over the 30 epochs of training.



The below image shows the confusion matrix for this training. A confusion matrix is a table that evaluates the performance of the classification model by providing a clear visual representation of how the model performed for each class. The blue boxes that right diagonally downward s from left to right in the image shows the number of samples that were classified correctly for the 7 characters. On the other hand, the rest of the boxes show the number of wrong predictions for each of the classes.



From the confusion matrix, the total number of images classified correctly for the 7 classes was 1726 while there were only 62 misclassifications across the entire validation set range.

9.2.3. Validation of the CNN on the segmented images

We tested the CNN on unprocessed images of the segmented characters. The performance observed was generally poor and the CNN struggled to classify the image. One possible reason was that there was a mismatch between the characteristics of the training dataset and the test images. The training dataset consists of black text on a white background, whereas the test images feature white text on a grey background, introducing a substantial visual inconsistency.

Second, the test images contain a high level of noise, with numerous white spots inside the test images that make it more difficult for the CNN to classify the letters correctly. The prediction of results of the CNN are shown below:

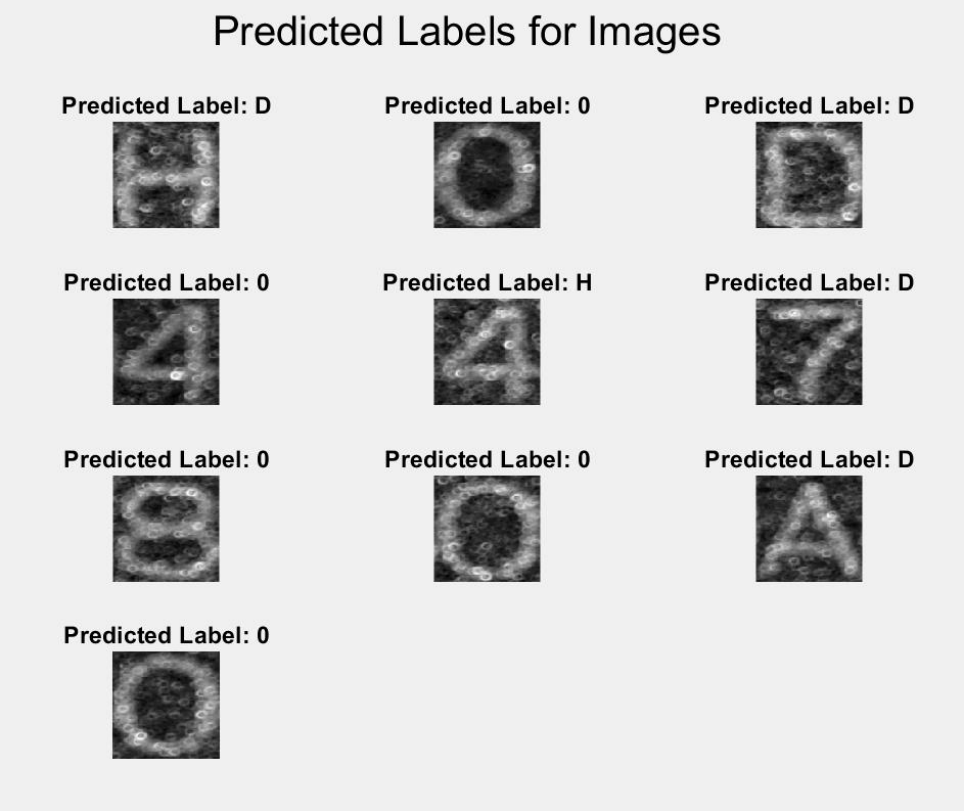


Figure 41: CNN predictions of the segmented images (unprocessed)

As observed, the CNN was only able to predict "0" and "D" correctly, and predicted all other characters wrongly. Section 11 of this report explores our work in processing the segmented images to improve the performance of our CNN classifier.

10. Task 8: Sub-task 2 - Design a MLP to classify each character in Image 1

10.1 Introduction

In this subsection, we classify character images using an MLP neural network. Two different approaches will be employed: the first utilizes MATLAB's Learning Toolbox, and the second involves implementing the MLP from scratch. Both methods will be used for training, validation, and image detection to evaluate their performance.

For this part, both neural network implementations employ a two-layer MLP with hidden layers containing 256 and 128 neurons, respectively. The hidden layers use ReLU as the activation function, while the classification output is achieved through a sigmoid function. These parameters were chosen because charactor recognition tasks typically do not require complex neural networks. On the contrary, overly deep or complex networks may lead to issues such as

overfitting, vanishing gradients, or exploding gradients, ultimately impairing training performance. Additionally, identical parameters were selected for both implementations to facilitate a direct comparison of their performance and provide a clearer assessment of the differences between the two approaches.

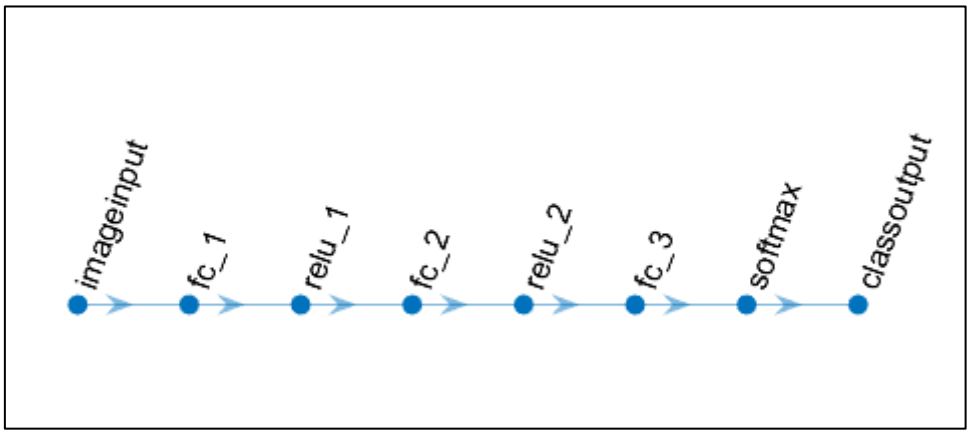


Figure 42 Network structure of the MLP

10.2 Implementation using MATLAB learning toolbox

10.2.1 Implementation details

Training an MLP using MATLAB's Learning Toolbox is straightforward. First, we define a neural network with two hidden layers, as illustrated in the diagram below:

```
layers = [
    imageInputLayer([imgSize 1])                % Input layer for grayscale images
    fullyConnectedLayer(256)                      % Hidden layer with 100 neurons
    reluLayer                                     % ReLU activation function
    fullyConnectedLayer(128)                       % Second hidden layer with 50 neurons
    reluLayer
    fullyConnectedLayer(numel(unique(imds.Labels))) % Output layer with neurons equal to the number of classes
    softmaxLayer                                  % Softmax for output probabilities
    classificationLayer                           % Classification layer
];
```

Figure 43 Definition of network structure

Subsequently, we define the training parameters as below. This training method utilizes gradient descent with momentum to ensure stability and improve efficiency during training. Additionally, since the original data is sorted by labels when loaded, it is likely that each batch contains only a single label, which can cause issues during training. To address this, we shuffle the label order at the start of each epoch, enhancing both the training efficiency and accuracy.

```
options = trainingOptions('gd', ...           % Stochastic gradient descent with momentum
    'InitialLearnRate', 0.0001, ...
    'MaxEpochs', 5, ...                        % Number of epochs
    'MiniBatchSize', 64, ...
    'Shuffle', 'every-epoch', ...
    'ValidationData', imdsTest, ...
    'ValidationFrequency', 30, ...
    'Verbose', false, ...
    'Plots', 'training-progress', ...
    'ExecutionEnvironment', 'gpu');            % Display training progress
```

Figure 44 Training options

Then we can train the policy with “trainNetwork” function from the Learning Toolbox.

10.2.2 Trained results

We conducted training on the dataset using the training set, and the results obtained are as follows:

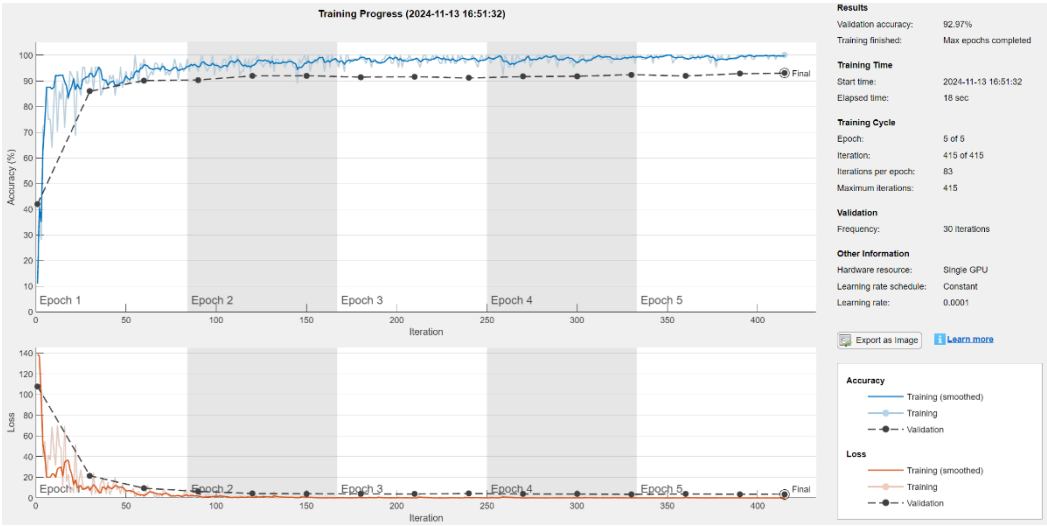


Figure 45 Training results of MLP with toolbox

From the above training graph, we observed that with a learning rate of 0.0001, the network successfully converged, achieving an accuracy of approximately 93% on the test set. However, even with this relatively simple MLP architecture, some overfitting was evident. This is reflected in the training results, where the accuracy on the training set exceeded 97%. Since the accuracy is not decreasing after 5 epochs, the training is stopped after that. The testing results are shown in the figure below:

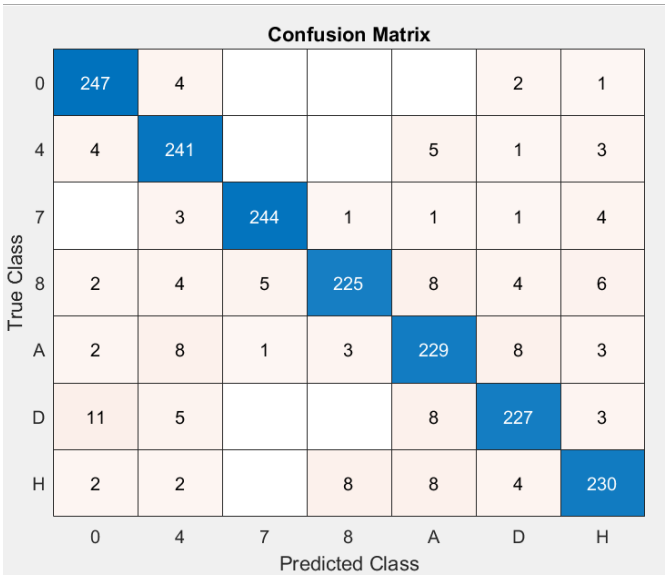


Figure 46 Confusion matrix of the MLP using Learning Toolbox

10.2.3 Validation of the results on the segmented images

We tested the MLP on unprocessed images and found that it struggled to classify the image. Several potential factors may contribute to this poor performance. First, there is a mismatch between the characteristics of the training and test datasets. The training dataset consists of black text on a white background, whereas the test images feature white text on a grey background, introducing a substantial visual inconsistency. Second, the test images contain a high level of noise, with numerous white glitters that the MLP struggles to filter out, further complicating the recognition task. Third, the MLP exhibits a strong sensitivity to positional variations within the input images, making it difficult for the model to generalize and correctly classify characters that appear in varying positions. The predictions are shown below:

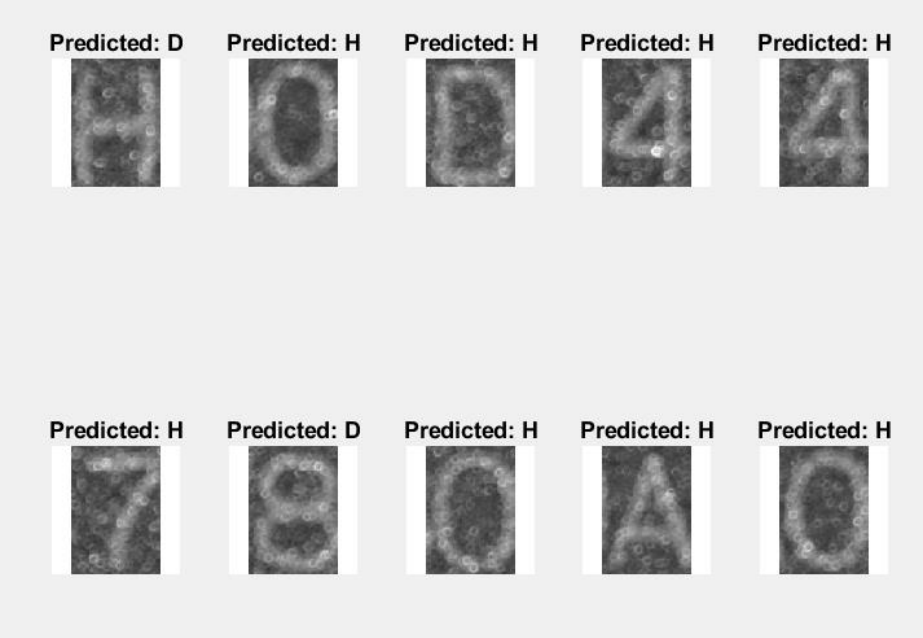


Figure 47 MLP predictions of the segmented images with toolbox

10.3 Implementation of MLP from scratch

10.3.1 Implementation details – Forward propagation

To implement a MLP to classify the images, we start from flattening the images into a 1D array, randomizing the orders, then forming them into batches. After which, we start by doing forward propagation for the hidden and output layers.

Hidden Layer (ReLU)

For each hidden layer performs a linear transformation followed by an activation function. For a hidden layer i, the output can be expressed as:

$$\boldsymbol{Z}_{i+1} = \boldsymbol{W}_i \cdot \boldsymbol{A}_{\{i\}} + \boldsymbol{b}_i \tag{6}$$

$$\boldsymbol{A}_{i+1} = \mathrm{ReLU}(\boldsymbol{Z}_{i+1}) \tag{7}$$

Here, $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ are the weights and biases for layer i, $\boldsymbol{A}_{\{i\}}$ is the input to the layer (or the output from the previous layer), and ReLU is the activation function:

$$\mathrm{ReLU}(x) = \max(0, x)$$

Output Layer (Softmax)

Since this task is a classification task, output layer applies a linear transformation followed by a sigmoid activation function to produce the final predictions:

$$\hat{\boldsymbol{Y}} = \mathrm{softmax}(\boldsymbol{Z_{out}}) = \mathrm{softmax}(\boldsymbol{W_i} \cdot \boldsymbol{A_i} + \boldsymbol{b_i})] \quad (8)$$

The softmax function maps the output to a probability prediction of the labels, and is calculated as:

$$\mathrm{softmax}(\boldsymbol{Z_{\{out\}}}) = \frac{e^{\boldsymbol{Z_{\{out\}}}}}{\sum_k^N e^{\boldsymbol{Z_k}}}$$

The implementation code of this in MATLAB is:

```
% Gradient Descent Forward Propagation
A{1} = imagesMatrix(:, batchStart:batchEnd);
for i = 1:n
    Z{i+1} = W{i}' * A{i} + b{i};
    if i < n
        % ReLU activation
        A{i+1} = max(Z{i+1}, 0);
    else
        A{i+1} = softmax(Z{i+1});
    end
end
```

Figure 48 Forward propagation

10.3.2 Implementation details – Back propagation

After implementing forward propagation, the next step is to perform backpropagation, which calculates gradients for updating the model's weights and biases. Backpropagation involves propagating the error from the output layer back through the network, layer by layer, using the chain rule of calculus.

Output Layer (Softmax)

The error at the output layer is computed based on the difference between the predicted probabilities and the actual labels. For softmax, the gradient of the loss L with respect to the $\boldsymbol{Z_{out}}$ is:

$$\frac{\partial L}{\partial \boldsymbol{Z_{out}}} = \hat{\boldsymbol{Y}} - \boldsymbol{Y} \quad (9)$$

Where $\hat{\boldsymbol{Y}}$ is the predicted probability and $\boldsymbol{Y}$ is the true label (one-hot encoded). The gradients for the weights and biases are calculated as follows:

$$\frac{\partial L}{\partial \boldsymbol{W_{out}}} = \left(\frac{\partial L}{\partial \boldsymbol{Z_{out}}}\right) * A_i^T \quad (10)$$

$$\frac{\partial L}{\partial \boldsymbol{b_{out}}} = \frac{\partial L}{\partial \boldsymbol{Z_{out}}} \tag{11}$$

Hidden Layers (ReLU)

For the hidden layers, the gradient of the loss with respect to the pre-activation output $\boldsymbol{Z_{i+1}}$ is propagated backward using the gradient from the subsequent layer and the ReLU activation function. The gradient for the i-th hidden layer is denoted as:

$$\frac{\partial L}{\partial \boldsymbol{Z_i}} = \left(\frac{\partial L}{\partial \boldsymbol{Z_{i+1}}} * \boldsymbol{W_i^T}\right) \circ \mathrm{ReLU}'(\boldsymbol{Z_{i+1}}) \tag{12}$$

Where $\boldsymbol{W_{i+1}^T}$ are the weights of the subsequent layer, $\circ$ denotes element-wise multiplication, and $\mathrm{ReLU}'(\boldsymbol{Z_i})$ is:

$$\mathrm{ReLU}'(\boldsymbol{Z}_i) = \begin{cases} 1, & \boldsymbol{Z_i} \geq 0 \\ 0, & \boldsymbol{Z_i} < 0 \end{cases}$$

The gradients for the weights and biases of layer i are:

$$\frac{\partial L}{\partial \boldsymbol{W_i}} = \frac{\partial L}{\partial \boldsymbol{Z_{i+1}}} * \boldsymbol{A_i^T} \tag{13}$$

$$\frac{\partial L}{\partial \boldsymbol{b_i}} = \frac{\partial L}{\partial \boldsymbol{Z_{i+1}}} \tag{14}$$

After the $\frac{\partial L}{\partial \boldsymbol{W_i}}$ and $\frac{\partial L}{\partial \boldsymbol{b_i}}$ are calculated, the new weight and bias can be calculated as:

$$\boldsymbol{W_{new}} = \boldsymbol{W_{old}} - \alpha \frac{\partial L}{\partial \boldsymbol{W_i}} \tag{15}$$

$$\boldsymbol{b_{new}} = \boldsymbol{b_{old}} - \alpha \frac{\partial L}{\partial \boldsymbol{b_i}} \tag{16}$$

Where α is the learning rate. The MATLAB code to implement this are shown below:

```
% Back propagation
dA = cell(1, n + 1);
dW = cell(1, n);
db = cell(1, n);
dA{n+1} = A{n+1} - trueValue;
for i = n:-1:1
    if i == n
        dZ{i+1} = dA{i+1};
    else
        dZ{i+1} = dA{i+1};
        dZ{i+1}(Z{i+1} <= 0) = 0;
    end
    dA{i} = W{i} * dZ{i+1};

    dW{i} = A{i} * dZ{i+1}';
    db{i} = sum(dZ{i+1}, 2);
    W{i} = W{i} - lr * dW{i};
    b{i} = b{i} - lr * db{i};
end
```

Figure 49 Backward propagation

10.3.3 Trained results

By setting the learning rate to 0.00001 and applying a learning rate decay that halves the rate at each epoch, we successfully achieved convergence during training. Experimental results showed that the classification accuracy on the test set reached 90% with the same parameters in previous subsection, slightly lower than the performance achieved using the toolbox.

We attribute this difference to two main factors. First, the toolbox utilizes momentum optimization, whereas we implemented only basic gradient descent, which led to slower convergence and required a lower learning rate to stabilize training. Second, the toolbox shuffles the data at the beginning of each epoch, ensuring better randomness, while in our implementation, we shuffled the data only once at the start, potentially introducing some bias during training.

After further experiments, we found that the batch size contributes significantly to minimizing the loss. With an adequate batch size, the performance of the NN can increase significantly. If the batch size is too large, the NN might even fail to converge. Below are the testing results:

Number of epochs	Mini-batch size	Learning Rate	Test Accuracy (%)
5	32	0.00001	91.3386 %
5	64	0.00001	90.1012%
5	128	0.00001	22.8909%
5	256	0.00001	14.0045%

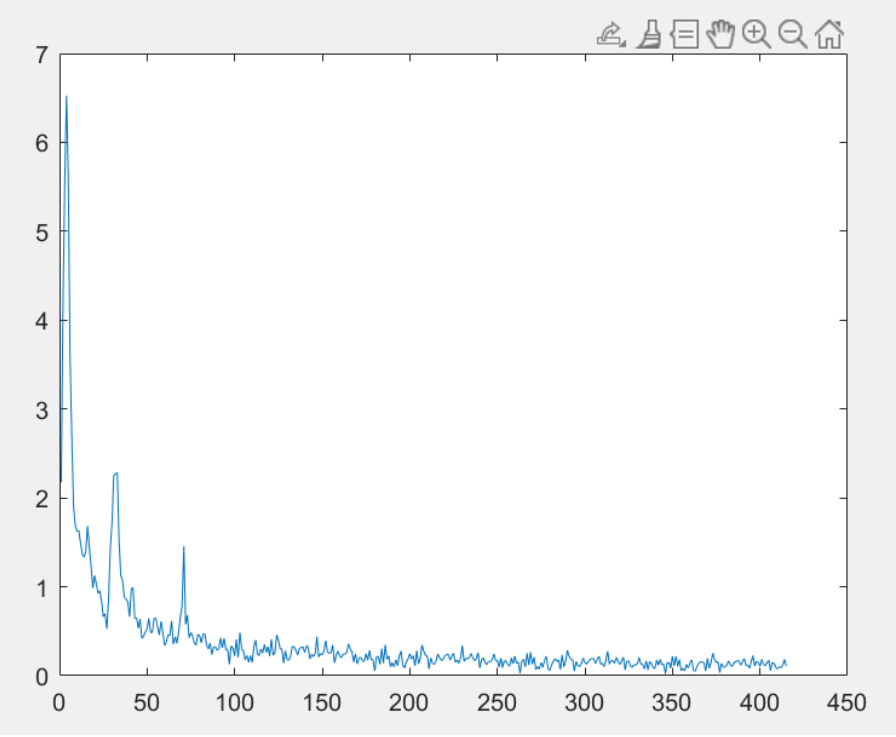


Figure 50 Training loss of the MLP from scratch (batch size = 64)

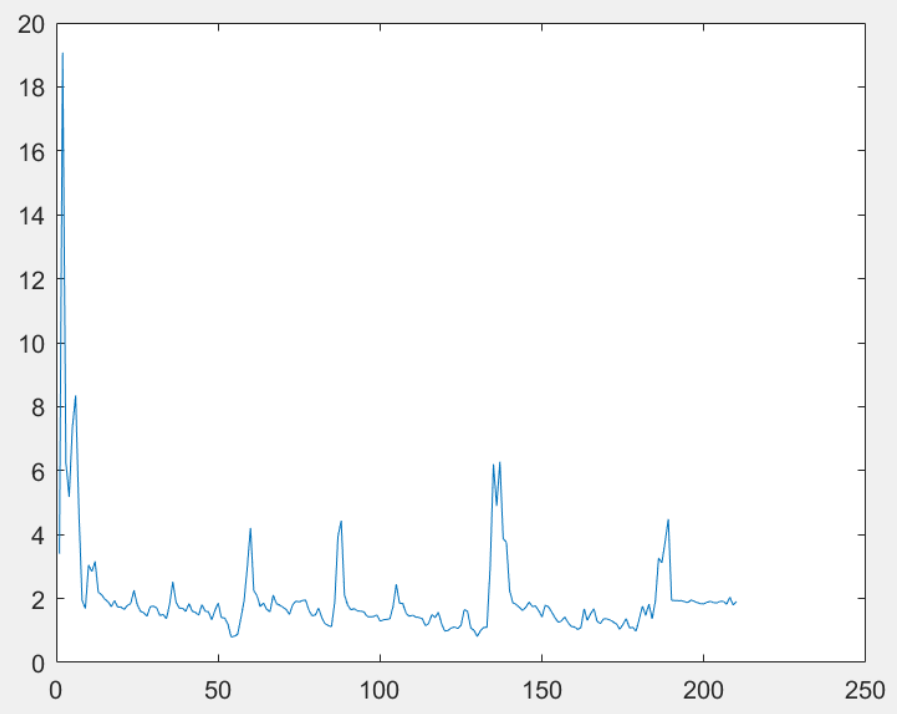


Figure 51 Training loss of the MLP from scratch (batch size = 128)

10.2.4 Validation of the results on the segmented images

The manually implemented MLP also performed poorly when predicting unprocessed images, facing similar issues as the implementation based on the toolbox. These issues include mismatched data characteristics and sensitivity to noise and positional variations, as previously discussed. The prediction results are illustrated in the figure below.

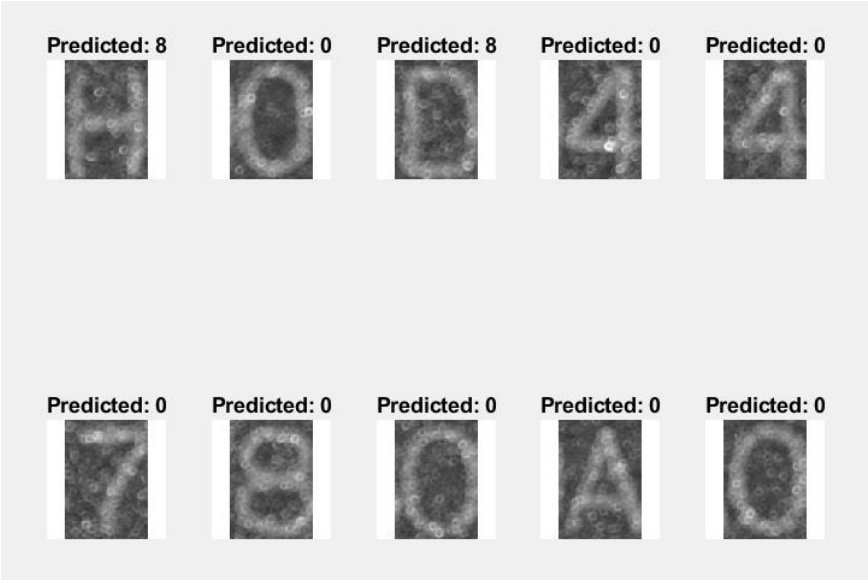


Figure 52 Classification results from manually implemented MLP

11. Task 9: Experimentation with pre-processing of data in Task 8

11.1. Pre-processing

Since the pixel size of the training images is uniformly 128×128×1, and all images have black characters on a white background, it is necessary to ensure a consistent format. Additionally, as the characters in the training images occupy a relatively small proportion, an additional 10% white margin is added around the images. Finally, the images are resized to 128×128. The pseudocode is as follows:

Algorithm 1: Image Padding and Resizing

Input: Folder path `subfolder` containing images matching pattern `bw_sub_*.bmp`

Output: Processed and saved images with padding and resizing applied

```
subfolder = 'Photos/';
imageFiles = list of all files in subfolder matching pattern 'bw_sub_*.bmp';
for k = 1 to length(imageFiles) do
    Ipath = full path of imageFiles[k];
    I = read image from Ipath;
    currentWidth = number of columns in I;
    currentHeight = number of rows in I;
    targetWidth, targetHeight = 128;
    paddingWidth = targetWidth − currentWidth;
    paddingHeight = targetHeight − currentHeight;
    leftPadding = ⌊ paddingWidth /2⌋;
    rightPadding = ⌈ paddingWidth /2⌉;
    upPadding = ⌊ paddingHeight /2⌋;
    downPadding = ⌈ paddingHeight /2⌉;
    I_inverted = complement of I;
    I_padded = pad I_inverted with:
        • leftPadding columns on the left,
        • rightPadding columns on the right,
        • upPadding rows on the top,
        • downPadding rows on the bottom,
        • an additional 13 rows and columns on all sides.
    Padding value is set to 255;
    I_padded = resize I_padded to 128 × 128 using bicubic interpolation;
    Extract name and ext from imageFiles[k];
    newFileName = construct new filename as 'resize_padding_inverted_{name}{ext}';
    Save I_padded to newFileName;
```

Figure 53 Pseudocode of pre-processing

After pre-processing, the characters are shown as follows:

Figure 54 Result of pre-processing

11.2. Experimentation with training of the CNN

We looked at the following hyperparameters that are typically deemed to be critical to the performance of the CNN.

11.2.1. Mini-Batch

Instead of processing the entire dataset at once or updating the weights for each individual image found in the dataset, training with mini-batches takes in groups of images at a time. Each batch size represents the number of training samples processed in each iteration and is considered as an important hyperparameter for deep learning. It helps the model to efficiently process the training data in chunks to fit into memory, provide stable and dynamic gradient updates to help the model to converge to a solution and lastly, to incorporate randomness via data shuffling and data augmentation to improve generalization.

From the below table, we tested the variation in mini-batch size against 10, 20 and 30 epochs and also with different fixed learning rate of 0.01, 0.05 and 0.1. The mini-batch sizes ranged from 128 to 1024.

Number of epochs	Mini-batch size	Learning Rate	Test Accuracy (%)
30	128	0.01	97.08%
30	256	0.01	96.46%
30	512	0.01	95.61%
20	128	0.01	96.85%
20	256	0.01	96.74%
20	512	0.01	93.03%
10	128	0.01	94.09%
10	256	0.01	91.56%
10	384	0.01	85.15%
10	512	0.01	68.62%
10	640	0.01	66.93%
10	1024	0.01	60.91%
10	128	0.05	96.74%
10	256	0.05	96.46%
10	512	0.05	27.78%
10	128	0.1	89.88%

From our observations, between 128 and 256 batch size the test accuracy was generally still good and above 95% for all combinations of epoch numbers and learning. These findings are consistent with the intuition that smaller batches help to introduce increased randomness to the gradient updates and helps with model generalization. However, as the batch size increase, the test accuracy starts to drop significantly, looking at the 10 epoch number test cases. For the

lower learning rate of 0.01, the test accuracy drops from >90% to 60.91% when the batch size increases from 256 to 1024. For the cases with the higher learning rate of 0.05 the test accuracy drops greatly to 27.78% when the batch size is 512. These observations could be explained by the reduced randomness in the gradient updates which may lead to overfitting problems or slower convergence to the optimal solution. Hence, this may be why these poor results are observed for the 10 epoch cases at 512 batch size and above while the 20 and 30 epoch cases still have >90% test accuracy.

11.2.2. Learning Rate

The learning rate is a hyperparameter that helps to control the size of the step that the model takes during the optimization process when updating the weights. Selecting the learning rate helps to strike a balance between the convergence speed and stability, to ensure that the model learns effectively without overshooting or to prevent it from getting stuck in a suboptimal state. The weights of the model are updated in each iteration with the below equation:

$$\omega_{t+1} = \omega_t - \eta . \nabla L(\omega_t) \tag{17}$$

Where ω_t refers to the current weights at iteration t, η is the learning rate, and $\nabla L(\omega_t)$ is the gradient of the loss function with respect to the weights. The gradient points in the direction of the steepest ascent of the loss function, this equation ensures that the weights for the next iteration *t+1* will move in the direction of steepest descent which minimizes the loss. If the learning rate is too small, it can cause the model to be stuck in a local minimum and the model may struggle to escape this region leading to slower convergence. On the other hand, too large of a learning rate may cause the model to overshoot the global minima and not be able to minimize the loss function which also leads to slower convergence or possibly even divergence. Hence, a balance has to be struck between these two extremes to find the ideal learning rate.

We tested three learning rate values: 0.01, 0.05, and 0.1 with epochs set to 10 and batch size set to 128. We plotted overlaid the 3 plots of model accuracy versus training iteration and found that the model with the learning rate of 0.05 converged the fastest of the three. This could mean that the fixed learning rate of 0.05 was the most optimal.

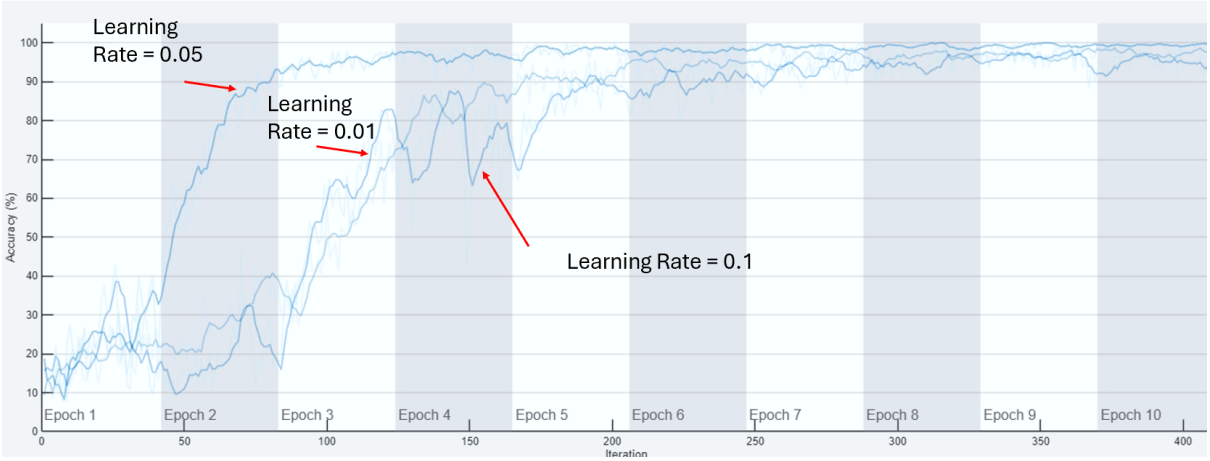


Figure 55: Test accuracy versus Iteration (Learning rate of 0.01, 0.05 and 0.1)

11.2.3. Epoch Number

The epoch number refers to the number of complete passes the model makes throughout the training dataset during the entire training process. This hyperparameter influences the performance of the CNN on both the training process and its effectiveness on unseen data.

If there are too few epochs, the CNN will not be able to train long enough to learn complex patterns in the data so performance would be poor on both the training and test data. The hidden layers in the CNN do not have enough exposure to the data to effectively extract the necessary hierarchical features for learning. On the other hand, if there are too many epochs, the CNN may not be able to generalize well as it tends to overfit to the training dataset. The model may end up learning the noise and specific details unique to the training dataset and end up performing poorly on unseen data even though it performs well on the training data.

Epoch number	Accuracy (%)
1	46.40
2	32.11
4	88.13
6	93.59
8	94.94
10	94.09
20	96.85
30	97.08

Based on the above table, it can be seen that the accuracy generally falls off as the epoch number falls below 4. Above an epoch number of 4, the model accuracy is >90% showing that there is reduced returns from further increasing the epoch number. There is only an incremental gain from increasing the epoch number up to 20 and even 30 epochs. Beyond a certain point, this may lead to an increased risk of overfitting to training data.

11.2.4. Experiment result

The experimental results are shown in the figure below, which shows that the CNN performance after pre-processing has been greatly improved. For given figure, the accuracy reaches as high as 100%.

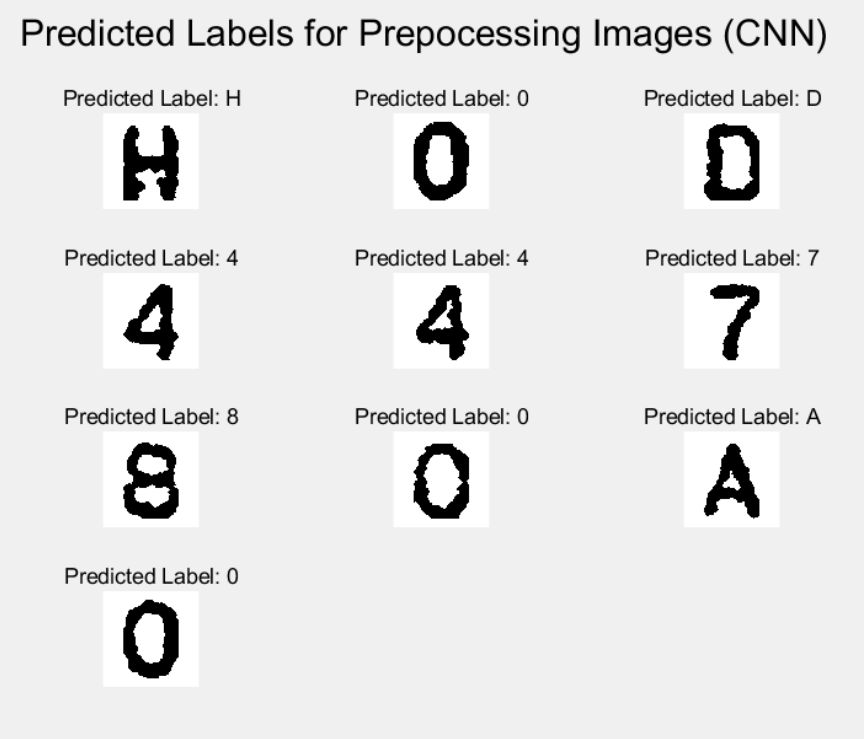


Figure 56 Classification result by CNN predictions (with preprocessing)

11. 2 Experiment with training of MLP

In comparison, using an MLP for image classification, even after preprocessing, yielded suboptimal results. The performance of both the toolbox-based implementation and the manually implemented MLP is illustrated in the figure below:

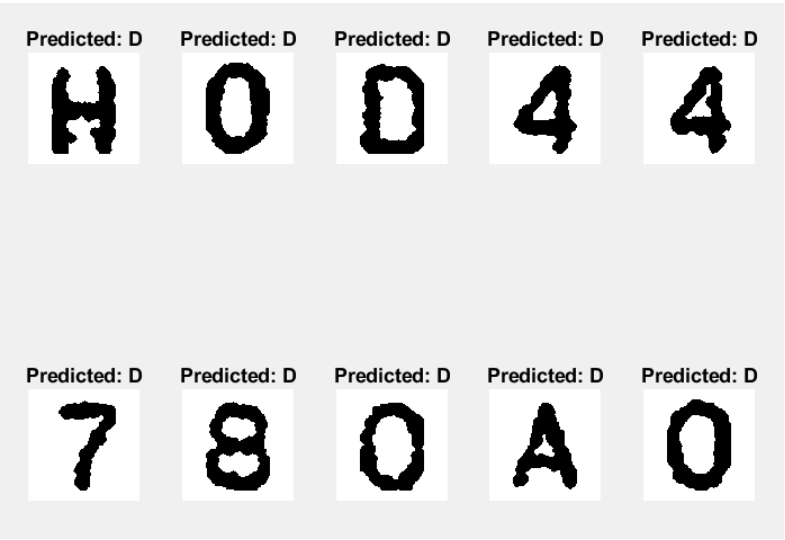


Figure 57 Classification result by MLP predictions (with preprocessing and toolbox)

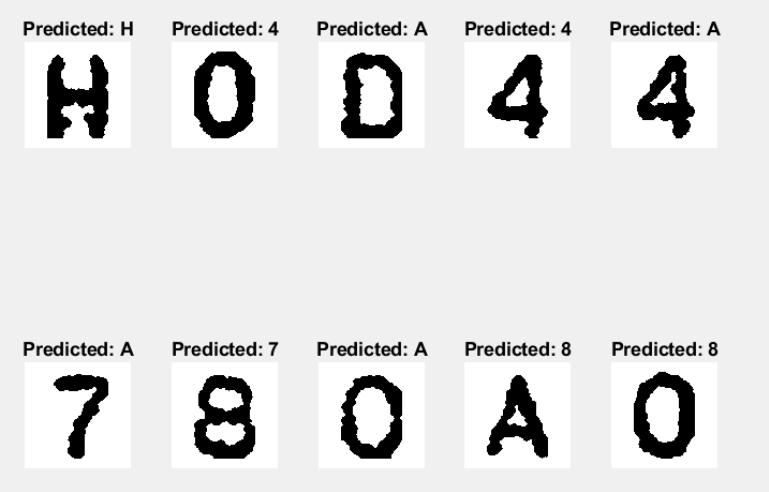


Figure 58 Classification result by MLP predictions (with preprocessing and our own implementation)

We believe the suboptimal performance of the MLP on processed images can be attributed to the following reasons: (1) Despite our efforts to make the pre-processed images resemble the training dataset, significant differences remain, making it challenging for the neural network to correctly classify them. (2) Even with the addition of padding, the results did not improve significantly. This is likely due to the fixed placement of data during training, which limited the model's ability to generalize. Additionally, since the MLP lacks convolutional kernels, it struggles to effectively extract spatial features, further hindering its ability to recognize patterns in the processed images.

Interestingly, we observed that our manually implemented neural network outperformed the toolbox-based implementation in this specific test. We speculate that this could be because the basic gradient descent method, compared to the stochastic gradient descent with momentum (SGDM) used in the toolbox, is less prone to overfitting. While gradient descent may perform worse on a highly similar test set, its reduced tendency to overfit allows the model to retain some level of generalizability, enabling it to produce functional predictions in this scenario.

12. Summary

In that study, we have explored a series of image processing and machine learning techniques to analyze a BMP image. Starting with preprocessing steps like contrast enhancement, brightness thresholding, and histogram equalization, we successfully improved the clarity of key image features. Each method demonstrated its strengths, though we observed trade-offs, such as the potential loss of detail when thresholds were poorly chosen.

Building on these foundations, we applied spatial and frequency domain filters, followed by edge detection algorithms like Laplacian and Sobel. The Laplacian method emerged as the optimal choice for segmenting characters due to its ability to generate non-broken edges, which was critical for accurate character isolation.

For character classification, we implemented both a Convolutional Neural Network (CNN) and a Multi-Layer Perceptron (MLP). The CNN, leveraging MATLAB's Deep Learning Toolbox, showed remarkable accuracy during validation but faced challenges with unprocessed segmented images due to mismatched data characteristics. The MLP provided a simpler architecture but lacked the spatial feature extraction capabilities of the CNN, resulting in suboptimal performance.

Through extensive experimentation with hyperparameters, including mini-batch size, learning rate, and epoch numbers, we demonstrated how these factors critically impact model performance. Preprocessing emerged as a key determinant in improving the CNN's ability to generalize and classify segmented images accurately, with results reaching near-perfect accuracy post-preprocessing.

This project has provided us with a deep understanding of the synergy between traditional image processing techniques and modern machine learning approaches. By carefully combining these tools, we have established a program to do character recognition successfully, and this program is expected to go on to solve more real-world problems.

13. References

[1] MathWorks, "What Is a Convolutional Neural Network?," Accessed: Nov. 17, 2024. [Online]. Available: https://www.mathworks.com/discovery/convolutional-neural-network.html

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, Jun. 2017. doi: 10.1145/3065386.