National University of Singapore
ME5411: Robot Vision and AI (Group 9)

Comparing the Processing Effectiveness of a Convolutional Neural Network (CNN) vs. Non-CNN Methods for Image Pre-and-Post Processing

---

Student :
Kukadia, Parthiv Vinubhai (A0304932J)
Zhao, Yunshuo (A0303651N)
Saravanan, Sharanya Nambi (A0240626L)

Teacher :
Dr. Chui Chee Kong
Dr. Peter C. Y. Chen

November 15, 2024

Contents

1 Introduction

The discipline of image processing and machine learning has seen great advancements over the past decade. These advancements have allowed people to use computers to perform tasks like object detection, classification, and enhancement of images with much greater accuracy. This project aims to implement and evaluate a number of image processing techniques and classification algorithms to preprocess an image of a microchip label, see Figure 1, and recognize the characters in the image. The project is built around using MATLAB to explore both traditional image processing techniques and modern machine learning approaches.

The project's first few tasks start with fundamental image manipulation, where we display and enhance the contrast of the provided image. The goal of these tasks is to prepare the image to be processed further by improving its clarity and providing an emphasis on the features that are of our interest. We will use filters such as a 5x5 averaging filter and a high-pass filter in a frequency domain to demonstrate their effects on smoothing the image and enhancing the edges. These tasks will help provide further insight into the role of spatial and frequency-domain filtering in standard image processing techniques.

Following these tasks, more advanced image processing steps are implemented, where we extract a sub-image of the characters and convert it into a binary image. These are steps that are required to simplify the image so that it is easier to detect the outlines of the characters in the image. The outlines characters are then segmented to separate and label individual characters in the image. The segmentation is vital in preparing our data to be used in image classification, allowing us to ensure that each character can be individually analyzed.

The main aspect of this project is focused on classification of the characters using two different approaches: A Convolutional Neural Network (CNN) and a non-CNN-based method. CNN's have proven to be greatly efficient for image classification tasks when dealing with complex patterns like handwritten characters. This task requires a CNN to be designed to help classify the characters in the image from the dataset provided to train the algorithm. For comparison, a non-CNN method is selected from traditional techniques covered in this course, which we employed to classify the characters. These methods could be Support Vector Machines (SVM), Multi Layer Perceptron (MLP), Reinforcement Learning (RL), and others, which all offer a different approach to classify images. The team has chosen to use Multi-Layer Perceptron as their non-CNN-based method.

Apart from classification, this project will also involve experimenting with pre-processing techniques and hyperparameter tuning. The pre-processing steps can help affect the performance of the classifiers by resizing, padding, or normalizing the input images. Hyperparameter tuning can help the optimize the CNN's performance by adjusting the learning rate, varying the batch size, and applying a number of layers or filters on the characters. Exploring these aspects will allow us to better understand how sensitive the classifiers are with respect to changes in the input data (the image) and the training parameters (the provided dataset).

This report will present the algorithms and techniques used, along with a detailed result from each stage of the project. We will provide discussion on why specific methods were chosen, the challenges we encountered, and the lessons we learned throughout the process. Our holistic approach will allow the team to demonstrate their ability to implement image processing and classification systems, while also highlighting the importance of evaluating and optimizing various different machine learning models for real-world applications.

2 BMP Image Processing

2.1 Displaying original image on screen; experimenting with contrast enhancement of the image, and commenting on the results

Using the code shown in Figure 5, we are able to obtain the Original image as shown in Figure 1. As the image shows, the characters are hard to see, and therefore we must apply some contrast enhancement such that the pixel intensity is redistributed across the image and the characters are more clearly visible.

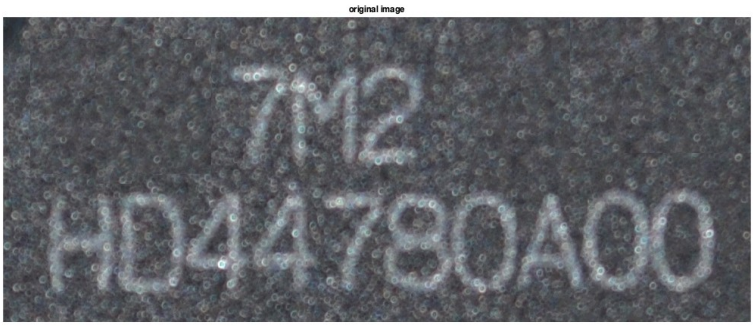

Figure 1: Original Image

The next step is to enhance the contrast of the grayscale image, a processing technique that helps improve the visibility of specific features of the image. We want to optimize features of the image such that we are able to conduct specific tasks for analysis and further processing. There are a number of different techniques that can be used to enhance the image, and we will explore 4 of them:

1. Using the `imadjust` function
   (a) This is the default contrast enhancement that can be done on a grayscale image. It adjusts the pixel brightness (intensity values) to improve the image contrast.
   (b) The pixel values are remapped from a range of input values to a definite output value, where the input and output range by default is the full intensity range of a grayscale image 0 - 255
2. Plotting the Histogram and using the `imadjust` function with a custom input and output range
   (a) This method requires the plotting of the pixel intensity histogram of the grayscale image to identify where the pixel intensities are concentrated. By plotting the histogram, we are able to control the contrast enhancement by spreading the intensity across the entire grayscale range.
   (b) Figure 2 shows the basic gray scale image histogram. This histogram was used to define the input and output range of the imadjust function for method 2.

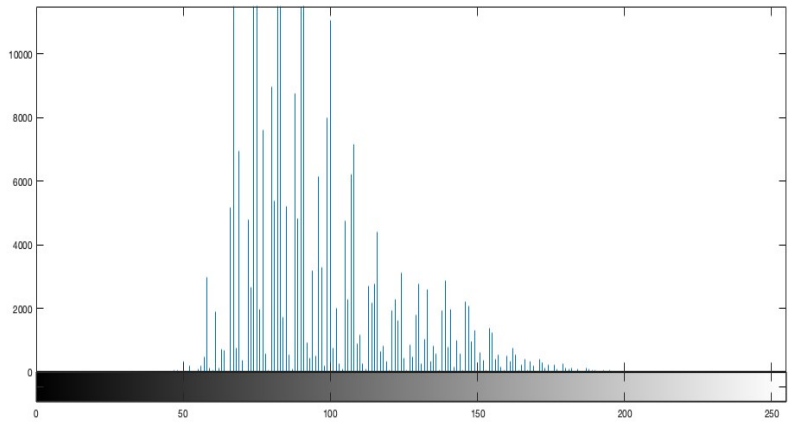

Figure 2: Grayscale Image histogram

(c) From the histogram, we can see that the concentration is between 50 - 130, which is the intensity range out of 255. This shows a small intensity range across a large portion of pixel values not being used, telling us that there is low contrast.

(d) Therefore, using the histogram, we are able to define our lower and upper input bounds are 50/255 = 0.196, 130/255 = 0.51. This becomes our custom input range in the imadjust function [0.19 0.51].

(e) Since we want to maximize the contrast enhancement, we define our output range to be the complete grayscale intensity range, 0/255 - 255/255, which is [0 1]. Therefore, we are able to map the original (input) range to the entire grayscale intensity range, providing us an image that has improved contrast and pixel intensity values that are stretched across the grayscale spectrum.

3. Using Histogram Equalization

(a) This method spreads the pixel intensity values to cover the entire grayscale range, by redistributing the intensities in a more even manner. It allows intensity values to occupy an equal number of pixels in each value so that you are able to obtain a more balanced and uniform histogram.

(b) The `histeq` function allows for more uniform distribution through equalizing the histogram and improving the global contrast of the image.

4. Using Adaptive Histogram Equalization

(a) This method is used by making details of the image more visible in areas where pixel intensity/image brightness is varying. It is implemented by dividing areas of the image into smaller regions (tiles) and applying adaptive contrast enhancement to each tile, which helps improve contrast in individual areas vs. globally changing the contrast. This ensures that both details in dark and light regions are made more visible and features are more prominent.

(b) The `adapthisteq` function allows for the improvement of local contrast by implementing histogram equalization to smaller regions of the image. This is extremely useful when the image has varying pixel intensities across the different areas.

The 4 techniques can be seen in Figure 3. The 1st method used was the imadjust function (The top left image). The 2nd method used was by defining a custom input/output range of the imadjust function using the grayscale image histogram (Figure 2 (The top right image). The 3rd method used was by using a

histogram equalization technique, which spreads out the pixel intensity values over the entire grayscale range (The bottom left image). The last method used was by using an adaptive histogram equalization technique, which improves the local contrast of an image (The bottom right image).

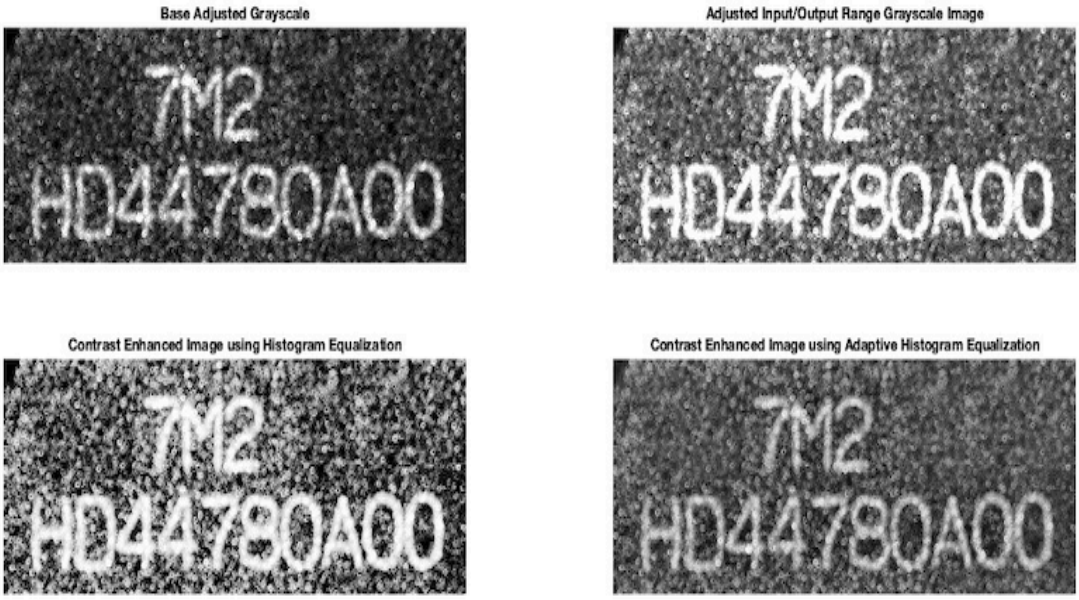

Figure 3: Different Contrast Enhancement Methods

From the 4 different methods used, some seem to have better contrast enhancement than others, but it is still difficult to tell from a naked human eye as to which image shows better contrast enhancement. Therefore, a Histogram is plotted for the 4 different enhanced images to identify which method is most efficient in improving the pixel intensity and providing better image features. These histograms are shown in Figure 4.

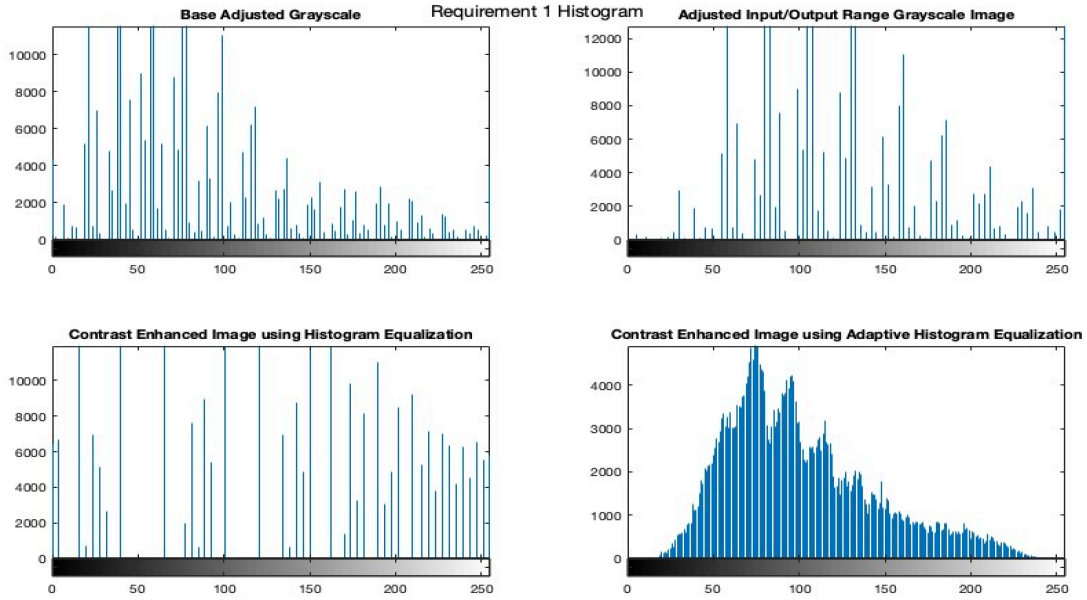

Figure 4: Different Contrast Enhancement Methods Histograms

The Histograms allow us to identify which contrast enhancement technique provides us with the best image to continue using in our image processing.

1. The top left (base adjusted grayscale histogram) shows sharp, narrow peaks that are concentrated over a specific range of pixel intensity values with gaps in between. Having most of the pixel intensities in the lower range between (0-100), the distribution is not uniform and we can conclude that there is a limited/poor contrast using this method. This is understandable since we are not making any changes to the imadjust function, and using the default input/output range of the full grayscale range (0 - 255).

2. The top right (adjusted input/output range histogram) shows peaks that are spread out over a greater intensity range than its predecessor, showing that the pixel intensity distribution has improved. However, there are still narrow peaks using this methodology and we are unable to see smooth transitions between the intensities, showing us that this is not the best contrast enhancement method.

3. The bottom left (contrast enhancement using histogram equalization) shows a greater spread in comparison to the base image, with greater intensity values, that are uniformly distributed, across the grayscale range. However, the pixel spread is not smooth and there are still narrow peaks, concluding that this is not the best method for us to use.

4. The bottom right (contrast enhancement using adaptive histogram equalization) shows a smooth and evenly distributed intensity histogram that shows a peak indicating a large number of pixels being pushed to the mid-level intensity range providing us with a balanced contrast enhancement. **We can see that this enhancement methodology is the best one out of the 4 contrast methods** for us to use with no extreme spikes or unused regions in the intensity scale.

```
%% Req 1 - Display original image and experimenting with contrast
% Load the .BMP file and display the original image
PSY = imread('charact2.bmp');  % Load the Image in :)
figure(1);
imshow(PSY);                   % Display the Original Image
title("Original Image");

% Convert Image to Grayscale from RGB
gray_PSY = rgb2gray(PSY);

% Method 1: Adjust image intensity value with default contrast enhancement
adjusted_gray_PSY = imadjust(gray_PSY);

% Method 2: Provide an adjust input/output intensity range from histogram
%imhist(gray_PSY);            % Display histogram of grayscale image
enhanced_gray_PSY = imadjust(gray_PSY, [0.19 0.51], [0 1]); % Change the range for
    intensity from histogram

% Method 3: Use Histogram Equalization to improve contrast in low-contrast images
equalized_gray_PSY = histeq(gray_PSY);

% Method 4: Use Adaptive Histogram Equalization to improve local contrast
adaptive_equalized_gray_PSY = adapthisteq(gray_PSY);

% Plot the different contrast enhancement methods
figure;
sgtitle("Requirement 1");
subplot(2,2,1), imshow(adjusted_gray_PSY), title('Base Adjusted Grayscale');
subplot(2,2,2), imshow(enhanced_gray_PSY), title('Adjusted Input/Output Range
    Grayscale Image');              % Display adjusted grayscale image
subplot(2,2,3), imshow(equalized_gray_PSY), title('Contrast Enhanced Image using
    Histogram Equalization');
subplot(2,2,4), imshow(adaptive_equalized_gray_PSY), title('Contrast Enhanced Image
    using Adaptive Histogram Equalization');
```

```
31
32 figure;
33 sgtitle("Requirement 1 Histogram");
34 subplot(2,2,1), imhist(adjusted_gray_PSY), title('Base Adjusted Grayscale');
35 subplot(2,2,2), imhist(enhanced_gray_PSY), title('Adjusted Input/Output Range
       Grayscale Image');              % Display adjusted grayscale image
36 subplot(2,2,3), imhist(equalized_gray_PSY), title('Contrast Enhanced Image using
       Histogram Equalization');
37 subplot(2,2,4), imhist(adaptive_equalized_gray_PSY), title('Contrast Enhanced Image
       using Adaptive Histogram Equalization');
38
39 Enhanced_gray_PSY = adjusted_gray_PSY;
```

Figure 5: MATLAB Code for Requirement 1

2.2 Implementation and application of a 5x5 averaging filter to the image; experimenting with filters of different sizes and Comparing the results of the respective image smoothing methods

During image processing, applying a filter enhances/suppresses features within an image, either to improve its quality or to clean the image up further for additional analysis. In our project, we will be applying an averaging filter, which is a type of smoothing filter used to reduce the noise and detail of an image. The averaging filter takes the average pixel intensity value of its surrounding neighbors and replaces the pixel value with that average, which is helpful in reducing the noise present in the image and blurring it out. Figure 6 shows a 5x5 filter being applied to the grayscale image.

Figure 6: 5 x 5 Averaging Filter

We decided to experiment with 4 different filter sizes, a 3x3 filter, a 5x5 filter, a 7x7 filter, and a 9x9 filter as shown in Figure 7. Experimenting with filters of different sizes, the following can be observed: as the filter size increases, the smoothing improves but image details also reduce. Therefore, deciding what size of filter the task requires is based on the requirements of applying the filter - whether you want to eliminate noise by using a larger filter, or if you want to preserve detail using a smaller filter.

In our case the following could be observed from the different filters:

1. 3x3 Filter
   (a) The filter smooths the image slightly, and preserves more detail than the others while reducing the noise slightly. The edges are still visible making it a good choice if you want to

apply mild smoothing, but not effective for images with significant noise

2. 5 x 5 Filter

   (a) This filter averages over a larger area of pixels with more of a blurring effect but it will start causing fine details to be lost. The image will become softer to look at as the smoothing will be greater.

3. 7 x 7 Filter

   (a) This filter smooths the filter even more, by reducing the noise a lot more and also implementing more of a blurring effect. It is useful for high noise images where the edge needs to be preserved.

4. 9 x 9 Filter

   (a) This filter causes the image to look washed out, with the edges and textures being obscured, and the averages of the filter being over a larger area. Best out of the 4 sizes for noise reduction but the worst for detail preservation.

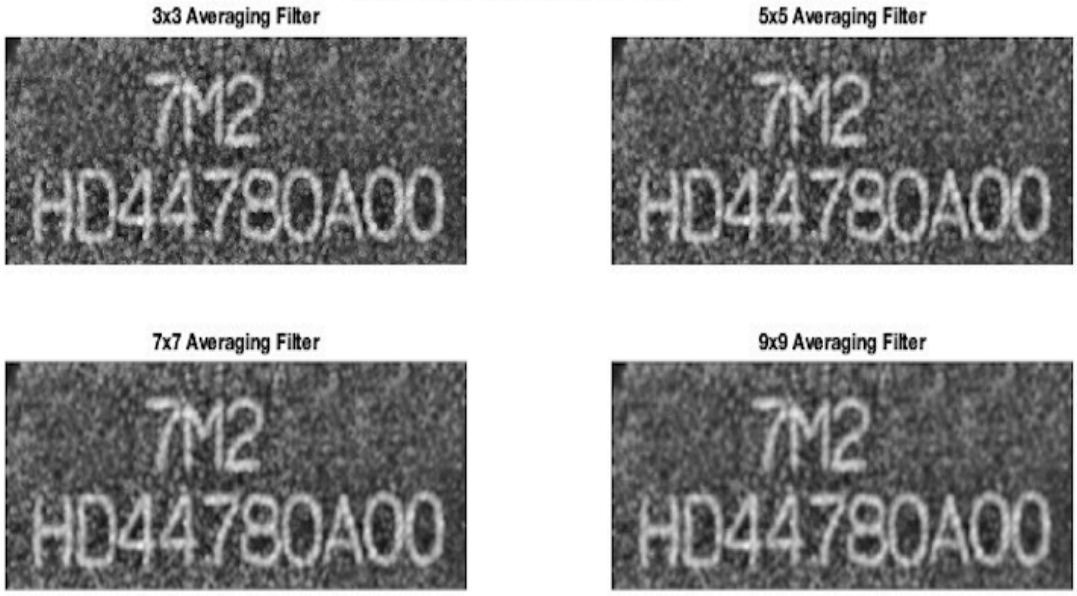

Figure 7: Comparison of different Filter Sizes

```
%% Req 2 - Implement and apply a 5x5 averaging filter
% Implementing a 5x5 Filter
a = 5; %change this number
ffffilter = ones(a,a)./a./a;
filter_PSY = imfilter(Enhanced_gray_PSY, ffffilter); %requirement 2 5*5 averaging
    filter

figure;
imshow(filter_PSY);
title("requirement 2");
filters = [3, 5, 7, 9] ; % change this number to experiment with different filter
    sizes

% Create loop to test different filter sizes
figure;
```

```
for k = 1:length(filters)
    a = filters(k);                 % Define filter size
    ffffilter = ones(a,a)./(a*a);    % Create filter
    filter_PSY = imfilter(Enhanced_gray_PSY, ffffilter); % Filter on Image

    % Plot the filtered images
    subplot(2, 2, k), sgtitle('Averaging Filters with Different Sizes');
    imshow(filter_PSY);
    title(sprintf('%dx%d Averaging Filter', a, a));
end
```

Figure 8: MATLAB Code for Requirement 2

2.3 Implementing and applying a high-pass filter on the image in the frequency domain; Comparing and commenting on the results and the resultant image in the spatial domain

Implementing and applying a high-pass filter on the image in the frequency domain consists of 3 steps.

1. Conducting Fourier Transform: to convert the image that is provided in the spatial domain to the frequency domain. The frequency domain will allow us to manipulate and change the image's frequency components.

2. Create a high-pass filter: this filter will allow high-frequency components to pass through the image while reducing the low-frequency components. This filter is made, such that frequencies within a specific radius from the center are made 0, and higher frequencies remain.

3. Applying the filter: the filter is then applied to the Fourier transform shifted image.

From Figure 9, we can observe that the image has enhanced edges and the details are finer when the "High-Pass Filter, Cut-off = 20". The high-pass filter allows the removal of lower-frequency components, which are usually the background intensities, and smooth variations. When we see the high-pass image against the original image, we are able to see some features that are more prominent because we have eliminated the lower frequencies. The image seems more defined now with a sharpness to it. To change the smoothness/details of the image, we are able to change the cut-off frequency of the high-pass filter, where a lower cutoff will keep more high-frequency details, and a higher cutoff will result in a smoother image. We can see that a high-pass filter can be used to enhance the edges of the image and provide finer details. It can be used for image sharpening and the level of detail can be controlled by changing the frequency cut-off value, as we can see being compared in Figure 9. We compared the high-pass filter cut-off frequency on our image changing from 5, 10, 20, 25. We can observe different effects on the image based on the value of our cut-off frequency:

1. Cut-off Frequency = 5: The filter can remove a few low-frequency data, and the image can retain a lot of the original detail with some minor enhancements. The subtle details are still visible but not with a lot of edge enhancements.

2. Cut-off Frequency = 10: The filter states to remove more low-frequency components with the 10 Hz filter, and we are able to see the edges and finer detail become more prominent and there is a sharper transition in the intensity. The image starts to look clearer and the image sharpness is a lot better.

3. Cut-off Frequency = 20: A large number of low-frequency components are removed, and we are able to see a lot more of the texture and enhanced edges, the image is more contrasted with the overall structure barely visible, and most of the image being lost due to the aggressive filtering.

4. Cut-off Frequency = 25: We are now focused almost only on the high-frequency components, with the image being extremely sharp but without much of the contextual information remaining. The image looks slightly artificial with minimal smooth transitions being visible now. The background elements look almost pixelated and too harsh.

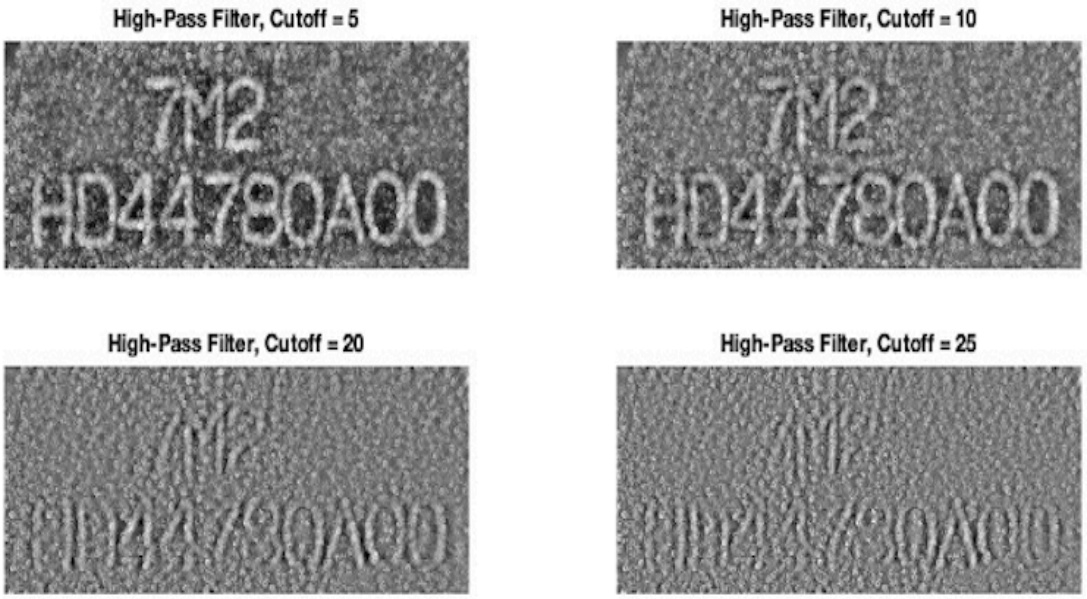

Figure 9: High-pass filter on the image in the frequency domain

There we are able to see that the optimal cut-off frequency depends on the goal of image processing and the filter has drastic effects on the visibility of detail and image quality. With an increase in frequency, the definition of the edges and details improve, with context and smoothness being lost. With lower frequency, we are able to retain more context but lose our enhanced edges. Therefore experimenting with different values is best in helping to determine the best balance between detail and natural appearance of the image.

```
%% Req 3 - Implementing and Applying a high-pass filter
fly = fft2(double(Enhanced_gray_PSY)); % Applying Fourier Fat Transform (2D)
fftt = fftshift(fly);

[l, w] = size(Enhanced_gray_PSY); % Image Dimenstions
[X, Y] = meshgrid(1:w, 1:l);
c_X = ceil(w/2); % Center of X
c_Y = ceil(l/2); % Center of Y

% Create loop to test different cut-off frequencies
highps_range = [5, 10, 20, 25]; % Different Cut-off frequencies
number_filters = length(highps_range);
highpass_PSY = cell(1,number_filters);

figure;
for k = 1:number_filters
    highps = highps_range(k); % Cut-off Frequency
    highpass_filter = sqrt((X - c_X).^2 + (Y - c_Y).^2) > highps;   % Create Filter
    highpass_fftt = fftt .* highpass_filter; % Implementing high-pass
    highpass_PSY{k} = real(ifft2(ifftshift(highpass_fftt))); % Conducting Inverse
        Fourier Trans. to get to spatial domain

    % Plot the comparisons
```

```
    subplot(2, 2, k), sgtitle('Req 3: Comparison of different cut-off frequency high
        -pass filtered images');
    imshow(highpass_PSY{k}, []);
    title(['High-Pass Filter, Cutoff = ', num2str(highps)]);
end
```

Figure 10: MATLAB Code for Requirement 3

2.4 Creating a sub-image that includes the middle line - HD44780A00

For this section, we have used a function called 'imcrop to crop a portion of the image. This created the sub image required as shown in Figure 11 - requirement 4(down)

Here is the breakdown of our code and the reasonings behind the steps taken:

1. x = 1: This starts the crop from the left side of the image.
2. y = l/2: This starts the crop from the middle of the image and therefore crops the bottom half of the image which is our requirement.
3. w : This captures the full width of the line.
4. Height = (l/2) -1: This crops half the original image's height minus one pixel, ensuring that it covers the entire bottom half of the image.

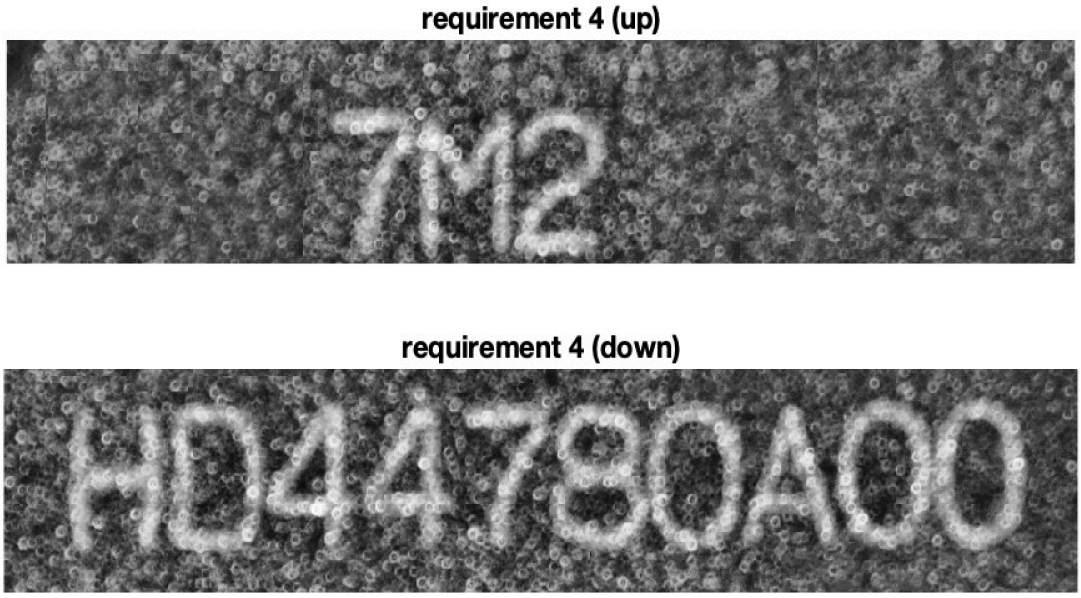

Figure 11: Creating a Sub-image that includes the middle line - HD44780A00

```
%% Req 4 - Create a sub-image of HD44780A00
sub_PSY1 = imcrop(Enhanced_gray_PSY, [1, 1, w, (l/2)-1]);
sub_PSY2 = imcrop(Enhanced_gray_PSY, [1, (l/2), w, (l/2)-1]);
figure;
imshow(PSY);  %scale
subplot(2,1,1);
imshow(sub_PSY1);
title("requirement 4 (up)");
subplot(2,1,2)
```

```
imshow(sub_PSY2);
title("requirement 4 (down)");
```

Figure 12: MATLAB Code for Requirement 4

2.5 Converting the sub-image into a binary image

For this part, we have first passed the image through a gaussian filter. This is done to smoothen the image and reduce noise. After that, we used high frequency component extraction to emphasize edges and fine details. This is done by calculating the high frequency components of the image by removing the blurred version of the image due to the filter with the original image. Subsequently, the original image is sharpened further using a sharpness factor of 2. The last step before binarization is to enhance the contrast of the image. This is done by using the function 'histeq' which performs histogram equalization and redistributes the pixels intensities of the image. We finally perform the binarization using 'imbinarize' with a threshold of 0.839. This means that pixels above this threshold become white (1) and pixels lesser than the threshold become black (0) as shown in Figure 13.

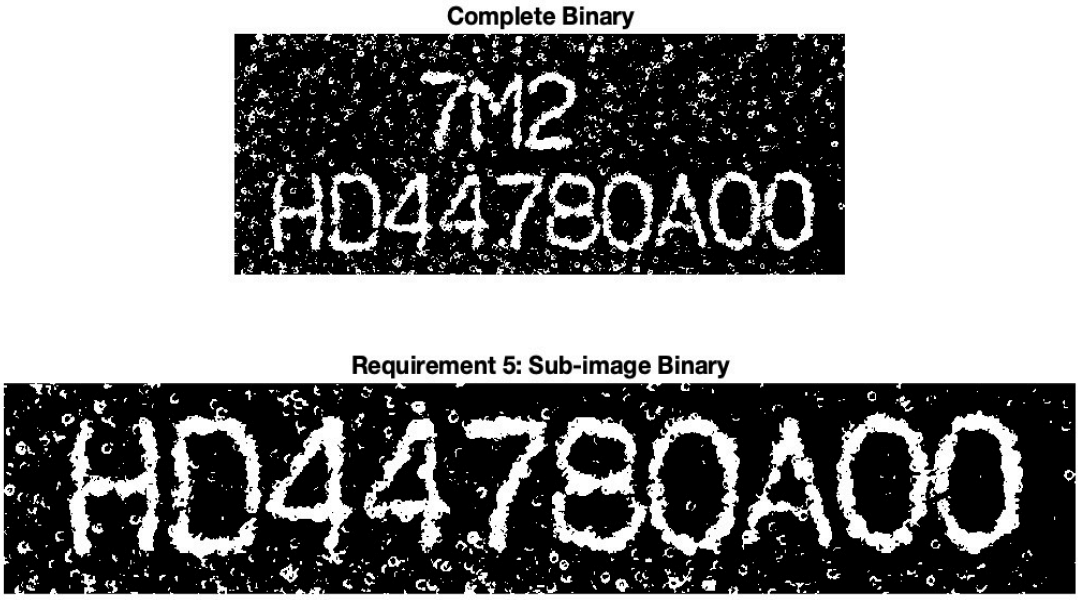

Figure 13: Converting the Sub-Image into a Binary Image

```
%% Req 5 - Convert sub-image to binary
guass_PSY = imgaussfilt(Enhanced_gray_PSY, 20);
high_freq_PSY = Enhanced_gray_PSY - guass_PSY;

sharpness_factor = 2;
sharpened_PSY = Enhanced_gray_PSY + sharpness_factor * high_freq_PSY;

contrast_PSY = histeq(sharpened_PSY);

bar = 0.839;
bi_PSY = imbinarize(contrast_PSY, bar);
imshow(imcrop(bi_PSY, [1, (1/2), w, (1/2)-1]));
figure;
imshow(bi_PSY);
title("Requirement 5: Sub-image Binary");
```

Figure 14: MATLAB Code for Requirement 5

2.6 Determining the outline of the characters in the image

For images of poor quality, accurately recognizing the characters within them can be very challenging. Although in previous steps we have applied filters and other operations to obtain a binarized image, these methods cannot completely eliminate the noise present in the image. As observed, even in step five, the resulting image still contains a significant amount of salt noise. Therefore, before performing edge detection, we need to further reduce this noise.

An effective method for removing salt noise is the opening operation, a common morphological operation that processes images by first applying erosion followed by dilation. The opening operation begins with erosion, which shrinks the foreground region, removing noise and isolated points smaller than the structuring element, thus eliminating small objects in the image. This is followed by dilation, which restores the size of the foreground region, preserving the main structure of the image. This process effectively removes small noise or isolated points while maintaining the overall shape and outline of larger foreground objects.

It is important to note that the choice of the structuring element in the opening operation must be suitable for the characteristics of the image. If the structuring element is too large, it may remove useful details; conversely, if it is too small, it may fail to eliminate all the noise.

In practice, we found that the two zeros in the bottom-right corner often merge, making them appear as a horizontal ”8.” This can significantly impact subsequent operations. After numerous attempts, we finally selected an appropriate structuring element for the opening operation.

After using the opening operation to remove salt noise, we can proceed to extract the edges of the characters using the Canny edge detection algorithm. Canny edge detection is a powerful algorithm that follows a definined methodology:

1. It begins by applying Gaussian filtering to reduce noise.
2. You then calculate the gradient magnitude and direction of the image.
3. This is followed by performing non-maximum suppression to precisely locate the edges.
4. You are to then perform double thresholding to classify gradient values into strong edges, weak edges, and non-edges.
5. Finally, edge tracking by hysteresis connects weak edges to strong edges, resulting in the final binary edge image

Using this algorithm effectively removes noise while precisely extracting the edge features of the image, making the contours of the characters clearer as shown in Figure 15.

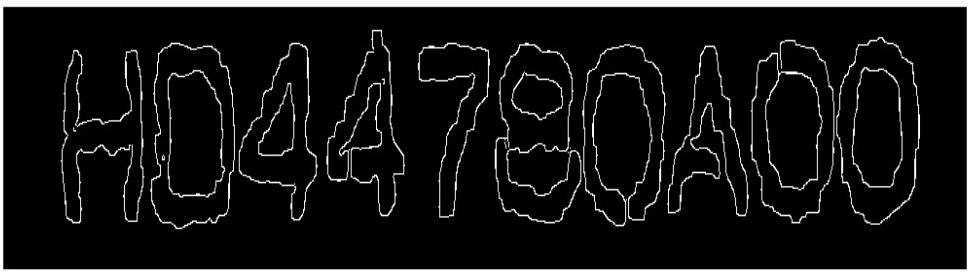

Figure 15: Outline of the Characters in the Image

```
%% Req 6 - Determine Outline of Characters
open_PSY = bwareaopen(bi_PSY, 185);

square1 = ones(5,5);
erode_PSY = imerode(open_PSY, square1);

square2 = ones(7,5);
erode_PSY = imdilate(erode_PSY,square2);

ud = ones(7,1);
erode_PSY = imerode(erode_PSY, ud);

ud = ones(13,1);
erode_PSY = imdilate(erode_PSY, ud);

edge_PSY = edge(erode_PSY, "canny");
figure;
imshow(edge_PSY);
title("requirement 6")
```

Figure 16: MATLAB Code for Requirement 6

2.7 Segmenting the image to separate and label the different characters as clearly as possible

After obtaining clear edges, separating different characters becomes much easier. First, we need to fill in the edge image, ensuring that the white areas (the 1s in the binary image) fully represent the character regions, while the black areas (the 0s in the binary image) represent the background. Once the filling is complete, we can crop the image to extract individual images of each character.

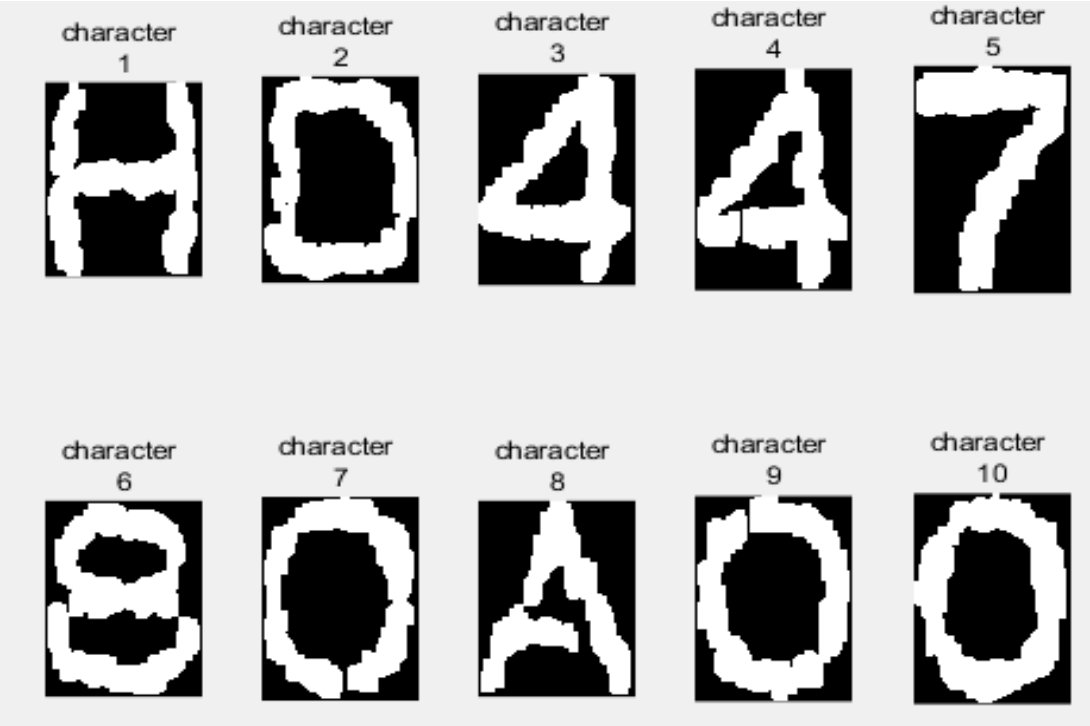

Figure 17: Segmenting the images separately

```
%% Req 7 - Segment the image to separate characters
open_PSY = bwareaopen(edge_PSY, 185);
fill_PSY = imfill(open_PSY, 'holes');
%imshow(fill_PSY);

figure;
[labe, nummm] = bwlabel(fill_PSY);
numimages = cell(1, nummm);
numprops = regionprops(labe, 'BoundingBox');

for k = 1:nummm
    thisBoundingBox = numprops(k).BoundingBox;
    numimages{k} = imcrop(erode_PSY, thisBoundingBox);
    subplot(ceil(nummm/5), 5, k)
    imshow(numimages{k});
    title(["character ", num2str(k)])
end
```

Figure 18: MATLAB Code for Requirement 7

3 Pre-Processing for CNN and MLP

For requirement 9, the team decided to reshape the characters into a 128x128 sized image, by padding the edges and ensuring that each character was the same size. We did 128x128 was done because the test characters provided in the training dataset were 128x128, allowing us to ensure that we were able to limit the inconsistencies in image classification and provide the most accurate results.

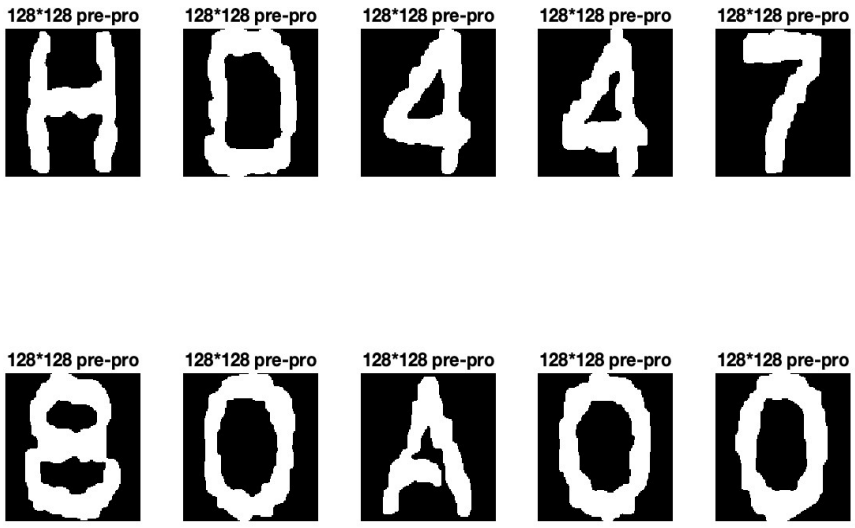

Figure 19: Pre-Processing the characters for CNN and MLP - 128x128

```
%% Req 9 - pre-processing 128*128
figure;
```

```
test_img = cell(1, nummm);

for ijk = 1:nummm
    [height, width] = size(numimages{ijk});
    if height > 128
        start_row = floor((height - 128) / 2) + 1;
        numimages{ijk} = numimages{ijk}(start_row:start_row+127, :);
    end

    if width > 128
        start_col = floor((width - 128) / 2) + 1;
        numimages{ijk} = numimages{ijk}(:, start_col:start_col+127);
    end

    [height, width] = size(numimages{ijk});

    pad_height = max(0, 128 - height);
    pad_width = max(0, 128 - width);
    test_img{ijk} = padarray(numimages{ijk}, [floor(pad_height / 2), floor(pad_width
        / 2)], 0, 'both');

    if mod(pad_height, 2) ~= 0
        test_img{ijk} = padarray(test_img{ijk}, [1, 0], 0, 'post');
    end

    if mod(pad_width, 2) ~= 0
        test_img{ijk} = padarray(test_img{ijk}, [0, 1], 0, 'post');
    end

    abc = ones(3,3);
    test_img{ijk} = imdilate(test_img{ijk}, abc);

    h = fspecial('gaussian', [5, 5], 2);
    test_img{ijk} = imfilter(double(test_img{ijk}), h);
    test_img{ijk} = imbinarize(test_img{ijk}, 0.5);

    subplot(ceil(nummm/5), 5, ijk)
    imshow(test_img{ijk});
    title("128*128 pre-pro");
end

chaaa = {'H', 'D', '4', '4', '7', '8', '0', 'A', '0', '0'};
chaaab = string(chaaa);
```

Figure 20: MATLAB Code for Requirement 9

4 Design of CNN

The team classified characters in the microchip label images= using Convolutional Neural Networks (CNN) due to their exceptional performance processing grid-like data. CNNs utilize convolutional layers that automatically learn spatial hierarchies of features, which make them extremely effective in recognizing edges, patterns, and textures which are used in identifying characters that have varying shapes and orientations.

In contrast to traditional methods like Support Vector Machines (SVM), which use manual feature extraction methods, CNNs autonomously extract features and learn from the raw image data when training the network. This end-to-end learning capability allows CNNs to handle complex image data and high dimensionality, making them adaptable and scalable to various image classification tasks. They are also more efficient in classying tasks.

CNNs are robust to variations in the input, including noise and distortions which are seen in our microchip image. They apply a local receptive field and pooling layer to focus on the most relevant features of the image. This design is used to minimize the impact of irrelevant details and enhances the model's ability to generalize efficiently across different images. Additionally CNNs facilitate a streamlined learning process by incorporating modern techniques like data augmentation and dropout, used to improve the performance of the model and avoid over-fitting. The result of this is high accuracy in character classification tasks.

While simpler algorithms may work for smaller, less complex datasets, CNNs are known to excel in environments that have intricate image data, as they use their layered structure to capture detailed patterns and build relationships. The decision to implement CNN for our character classification task is due to their advanced feature learning, superior performance in handling complex image data, and strength to noise.

Overall, the choice of CNN was driven by the models ability to effectively process high-dimensional image data, learn hierarchical features, and achieve high performance in character classification tasks in the context of our microchip label image.

4.1 Implementation of CNN

1. Loading Image Data for CNN

   (a) The code starts off the setting up the image datasources for training and testing datasets. Using the imageDatastore function, images are loaded from specified folders ($'p\_dataset\_26'$) for training and $'p\_dataset\_26'$ for testing). All subfolders are also included and automatically assigned labels based on folder names (with 'LabelSource' set to 'foldernames').

   (b) CNNs have a typical image size which is the most optimal. Therefore, each image is resized to 128x128 pixels. The ReadFcn function defines a custom reading process for each image. This process includes reading the image, resizing it to the target dimensions (128x128), converting it to binary format using imbinarize, and inverting the colors with imcomplement (making the background 0 and the foreground 1). This process ensures the dataset for CNN training is consistent.

   (c) The code specifies $num_classes = 7$, indicating there are seven unique classes in the dataset. Additionally, imageSize = [128 128 1] is set, which specifies that on top of resizing the image it is also important that it has a single-color channel (grayscale). This image size is necessary for CNN layers to understand the input shape when constructing the network.

2. CNN Network Architecture

   (a) The network starts with an input layer that takes in the 128x128 grayscale images. Normalization is set to zero center, which speeds up the training and allows to model to converge more efficiently.

   (b) The network has six blocks of layers. Firstly, a convolutional layer (convolution2dLayer) that applies 3x3 filters (kernels) with increasing filter counts from 8 to 256. The 'Padding', 'same' option ensures input size is preserved. Increasing the filter counts helps to capture more complex patterns as well.

   (c) Subsequently, batch normalization layer (batchNormalizationLayer) reduces dependencies on initial weights and allows higher learning rates. This leads to faster and more efficient training.

   (d) A ReLU activation layer (reluLayer) introduces non-linearity, allowing the network to learn complex patterns in the data.

   (e) A max pooling layer (maxPooling2dLayer) downsamples the feature maps, which reduces computation and helps the network focus on the most prominent features

   (f) After the convolutional layers, the feature maps are flattened and passed through a fully connected layer with 512 units which helps to extract the most prominent features. A final fully connected layer provides class scores, which are converted to probabilities by the softmax layer. The classification layer then assigns class labels based on these probabilities, calculating loss during training.

3. Training the CNN

   (a) trainNetwork uses the preprocessed dataset to train the network to recognize patterns associated with each class which optimizes weights and biases across the network to minimize classification error.

   (b) The trained network is stored in net. The net can be used to make predictions on new images or evaluate its performance on a test dataset.

4. Testing the CNN

   (a) The model is finally tested on the dataset and the accuracy is calculated based on the difference between the desired and predicted values. Testing this new dataset provides gives us an idea of model performance, indicating how well CNN can classify unseen data.

   (b) The model after testing is again saved so that it can be used on a completely new set of data that the model has not seen before. This will determine if the trained model is accurate enough.

5. Usage of trained model

   (a) Firstly, the pre-trained CNN model saved in a file named cnn.mat is loaded. This pre-trained model will be used to classify the test images.

   (b) Each test image in $test\_img$ (assumed to be a cell array). Each image is inserted into the $input\_img\_CNN$ matrix as a single channel (grayscale) image in the format required for CNN input.

   (c) The predict function uses the CNN (net) to classify the images stored in $input\_img\_CNN$. The output, $char\_pred\_CNN$, is a matrix where each row contains the predicted probabilities for each class for a single image. The model selects the class with the highest probability as the prediction maximizes confidence and thus is more reliable.

(d) A new figure is created with a specific size and position to display the results.

(e) If the predicted label matches the actual label, the title indicates a correct classification with green text. If it does not match, the title indicates a wrong classification with red text. This gives us insights into model accuracy and confidence and how to fine tune the parameters used for training

4.2 CNN Model Flowchart

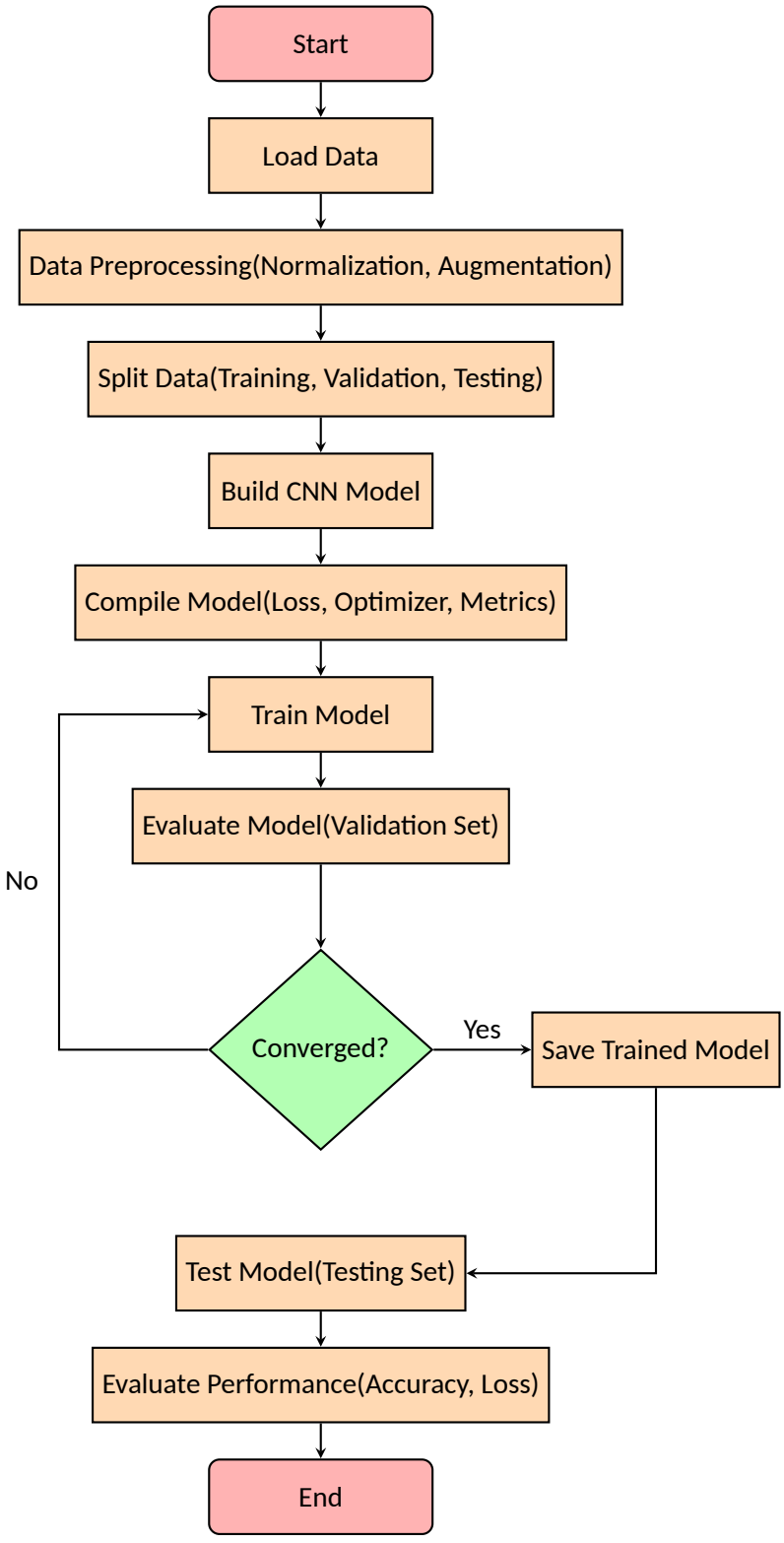

Figure 21: Flowchart for Developing CNN Network

Below we can see the image classification using CNN, we can see the letters being classified with 100% accuracy for our microchip image.

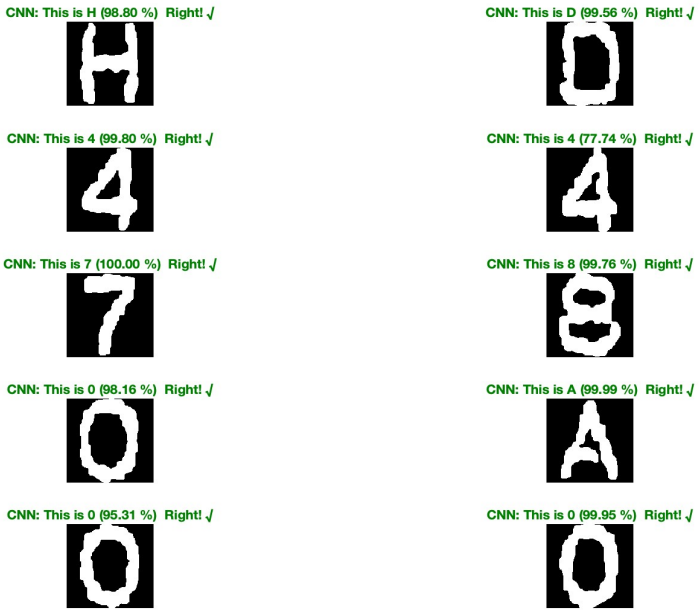

Figure 22: CNN based Classification of Images

To run the code, first run the code provided in Figure 23 to obtain a .NET framework integration that allows MATLAB to create objects and call for a method. Once this cnn.net file is created, run the code in Figure 24 to obtain the final results.

4.3 CNN Code

```
clear; clc; close all;

training = imageDatastore('p_dataset_26\dataset\train', ...
    'IncludeSubfolders', true, ...
    'LabelSource', 'foldernames', ...
    'ReadFcn', @(x) imbinarize(imresize(imcomplement(imread(x)), [128, 128])));

testing = imageDatastore('p_dataset_26\dataset\test', ...
    'IncludeSubfolders', true, ...
    'LabelSource', 'foldernames', ...
    'ReadFcn', @(x) imbinarize(imresize(imcomplement(imread(x)), [128, 128])));

num_classes = 7;
imageSize = [128 128 1];

layers = [
    imageInputLayer([128 128 1], 'Normalization', 'zerocenter', 'Name', 'imageinput'
        )

    convolution2dLayer(3, 8, 'Padding', 'same', 'Stride', [1 1], 'Name', 'conv_1')
    batchNormalizationLayer('Name', 'batchnorm_1')
    reluLayer('Name', 'relu_1')
    maxPooling2dLayer(2, 'Stride', [2 2], 'Padding', [0 0 0 0], 'Name', 'maxpool_1')

    convolution2dLayer(3, 16, 'Padding', 'same', 'Stride', [1 1], 'Name', 'conv_2')
```

```
    batchNormalizationLayer('Name', 'batchnorm_2')
    reluLayer('Name', 'relu_2')
    maxPooling2dLayer(2, 'Stride', [2 2], 'Padding', [0 0 0 0], 'Name', 'maxpool_2')

    convolution2dLayer(3, 32, 'Padding', 'same', 'Stride', [1 1], 'Name', 'conv_3')
    batchNormalizationLayer('Name', 'batchnorm_3')
    reluLayer('Name', 'relu_3')
    maxPooling2dLayer(2, 'Stride', [2 2], 'Padding', [0 0 0 0], 'Name', 'maxpool_3')

    convolution2dLayer(3, 64, 'Padding', 'same', 'Stride', [1 1], 'Name', 'conv_4')
    batchNormalizationLayer('Name', 'batchnorm_4')
    reluLayer('Name', 'relu_4')
    maxPooling2dLayer(2, 'Stride', [2 2], 'Padding', [0 0 0 0], 'Name', 'maxpool_4')

    convolution2dLayer(3, 128, 'Padding', 'same', 'Stride', [1 1], 'Name', 'conv_5')
    batchNormalizationLayer('Name', 'batchnorm_5')
    reluLayer('Name', 'relu_5')
    maxPooling2dLayer(2, 'Stride', [2 2], 'Padding', [0 0 0 0], 'Name', 'maxpool_5')

    convolution2dLayer(3, 256, 'Padding', 'same', 'Stride', [1 1], 'Name', 'conv_6')
    batchNormalizationLayer('Name', 'batchnorm_6')
    reluLayer('Name', 'relu_6')
    maxPooling2dLayer(2, 'Stride', [2 2], 'Padding', [0 0 0 0], 'Name', 'maxpool_6')

    fullyConnectedLayer(512, 'Name', 'fc_1')
    reluLayer('Name', 'relu_7')

    fullyConnectedLayer(num_classes, 'Name', 'fc_2')
    softmaxLayer('Name', 'softmax')
    classificationLayer('Name', 'classoutput')
];

options = trainingOptions('adam', ...
    'InitialLearnRate', 0.001, ...
    'MaxEpochs', 9, ...
    'Shuffle','every-epoch', ...
    'MiniBatchSize', 64, ...
    'Shuffle', 'every-epoch', ...
    'ValidationData', testing, ...
    'ValidationFrequency', 20, ...
    'Verbose', false, ...
    'Plots', 'training-progress', ...
    'ExecutionEnvironment', 'gpu');


net = trainNetwork(training, layers, options);

YPred = classify(net, testing);
YTest = testing.Labels;

accuracy = sum(YPred == YTest)/numel(YTest);
disp(['Accuracy: ', num2str(accuracy * 100), '%']);

save('cnn_test.mat', 'net');
```

Figure 23: CNN Method Source Code

```
%% CNN Classification Method
load('cnn.mat', 'net');
input_img_CNN = zeros(128, 128, 1, nummm);

for i = 1:nummm
```

```
    input_img_CNN(:, :, 1, i) = test_img{i};
end

char_pred_CNN = predict(net, input_img_CNN);
[~, predict_CNN] = max(char_pred_CNN, [], 2);

figure(11);
set(gcf, 'Position', [950, 100, 800, 600]);
for i = 1:nummm
    predict_num_CNN = predict_CNN(i);
    predict_letter_CNN = label_map{predict_num_CNN};
    acc_CNN = max(char_pred_CNN(i, :));
    accs_CNN = num2str(acc_CNN * 100, '%.2f');

    subplot(ceil(nummm/2), 2, i);
    imshow(test_img{i});
    if predict_letter_CNN == chaaab(i)
        title(['CNN: This is ', predict_letter_CNN, ' (', accs_CNN, ' %)' , '  Right
            ! √'], 'Color', '[0 0.5 0]');
    else
        title(['CNN: This is ', predict_letter_CNN, ' (', accs_CNN, ' %)' , '  Wrong
            ! ×'], 'Color', 'red');
    end
end
```

Figure 24: CNN Method Source Code

5 Design of MLP (Non-CNN Method)

The team chose to use Multilayer Perceptron (MLP) for classifying characters in the microchip label image because of its superior ability to handle the complexity and high-dimensional nature of data for images. The MLP has an architecture of being multi-layer, which allows us to use it to model non-linear relationships effectively, which is important in detecting the variation of shapes, sizes, and orientation of different characters.

Unlike SVM, a traditional method, which requires the algorithm to manually extract the features of the image, MLPs learn the features from the raw image pixels. This allows the use of MLP to be more scalable and flexible for various tasks that involve varying pixel-level data. Furthermore, MLPs are built such that they are robust to distortions and noise in images, which exist in microchip labels due to how they are manufactured or taken images of.

MLPs are also used to offer end-to-end learning, where the algorithm is able to directly learn and classify the different characters without requiring intermediate pre-processing steps. This process is streamlined, such that the ability to fine-tune the model is done using modern optimization techniques, making MLP an efficient and powerful choice for our task. We are aware that SVM and similar methods can work for simpler and smaller datasets, but MLPs are better developed to handle the complexities present in image data like the label.

Overall, the choice of MLP was made because of its ability to model non-linearity's, efficiently handle high-dimensional image data with noise, automatically learn different features in the image, and provide character classification with greater performance.

5.1 Implementation of MLP

MLP is a type of feedwork neural network with completely connected layers that help map input data, which are image pixels in this project, to output classes, which are the character labels for us. MLPs work well for smaller, and simpler datasets, but their performance worsens on complex image data in comparison to Convolutional Neural Networks because they lack spatial awareness. However, if we pre-process the images well enough, MLPs can still be effective for certain image classification tasks.

The implementation of MLP is done by implementing the code in Figure 27 followed by Figure 30. Insight into the algorithm can be seen below:

1. Data Preparation to load the images
    (a) The code starts off by loading images from a folder that contains the character datasets for training and testing using the imageDataStore function. The labels for each image are extracted from folder names (LabelSource, 'foldernames').
    (b) The training and testing of the datasets are all binary images of the size 128x128. This corresponds to 16384 pixels. For MLP, we require a 1D input, and therefore we flatten it out to a vector for each image in the data set
    (c) Each image undergoes being converted into a binary format (using imbinarize), followed by inverting the image such that the backgrounds are now 0 and the foregrounds are 1 (using 1 - img), followed by reshaping the image into a 1D vector and storing it in input_training and input_testing matrices.
    (d) All these labels are then converted to numerical values, and using the ind2vec function, we convert them into a one-hot encoded vector that is used to match the format required for

MLP to use for image classification.

2. Architecting the MLP network

   (a) The network architecture is built on 2 hidden laters which are defined in the code as hidden_layer = [128,64], which means that the first hidden layer consists of 128 neurons, while the second one consists of 64 neurons.

   (b) The final layer consists of as many neurons as there are classes (the number of different characters - H, D, A, 4, 7, 8, 0), which we define as num_classes = 7.

3. Begin training the MLP model

   (a) We provide some training parameters, where the network is trained for upto 1000 epochs. We stop the network from training early if the validation performance is not improving for 2000 iterations. The training goal is set to 0 to ensure that the network aims for maximum accuracy. Finally, the minimum gradient is set to 1e-20, such that the network will stop training if the gradient gets smaller than this.

   (b) We don't need to split the training data into validation or testing sets within the network since we have been provided the data separately for training and testing, which is captured using the net.dividePama function.

   (c) The training we do is done using GPU shown from the 'useGPU', 'yes' option to allow for faster computation.

   (d) Finally, once the training is complete, we save the model as a .mat file to use for future cases (mlp.mat).

4. Test the MLP model

   (a) Since we have trained the network, we would like to test the model out now. The network is tested on the dataset using input_testing and the output is processed through vec2ind which converts the one-hot encoded output back to the corresponding class labels.

   (b) The model accuracy is then calculated by comparing our predicted labels with the actual labels that we got from running the model, and this accuracy is displayed as a percentage.

5. Load and use the trained model

   (a) Finally, we load the MLP model that we trained for classification of the characters of our microchip image.

   (b) The network is loaded into our matlab file with the microchip image.

   (c) A new set of character images is processed similar to how we processed the training data, by flattening and storing it in input_characters.

   (d) The network is then used to predict the labels of these characters and we then display them as their corresponding letters.

   (e) The label_map is then used to map the numerical predictions back to their corresponding character labels for the lower half of our microchip label image.

   (f) The image is them displayed using the imshow function, and each character is plotted as a subplot with the predicted character as a label for the subplot images (e.g., This is H, This is D, etc.).

5.2 MLP Model Flowchart

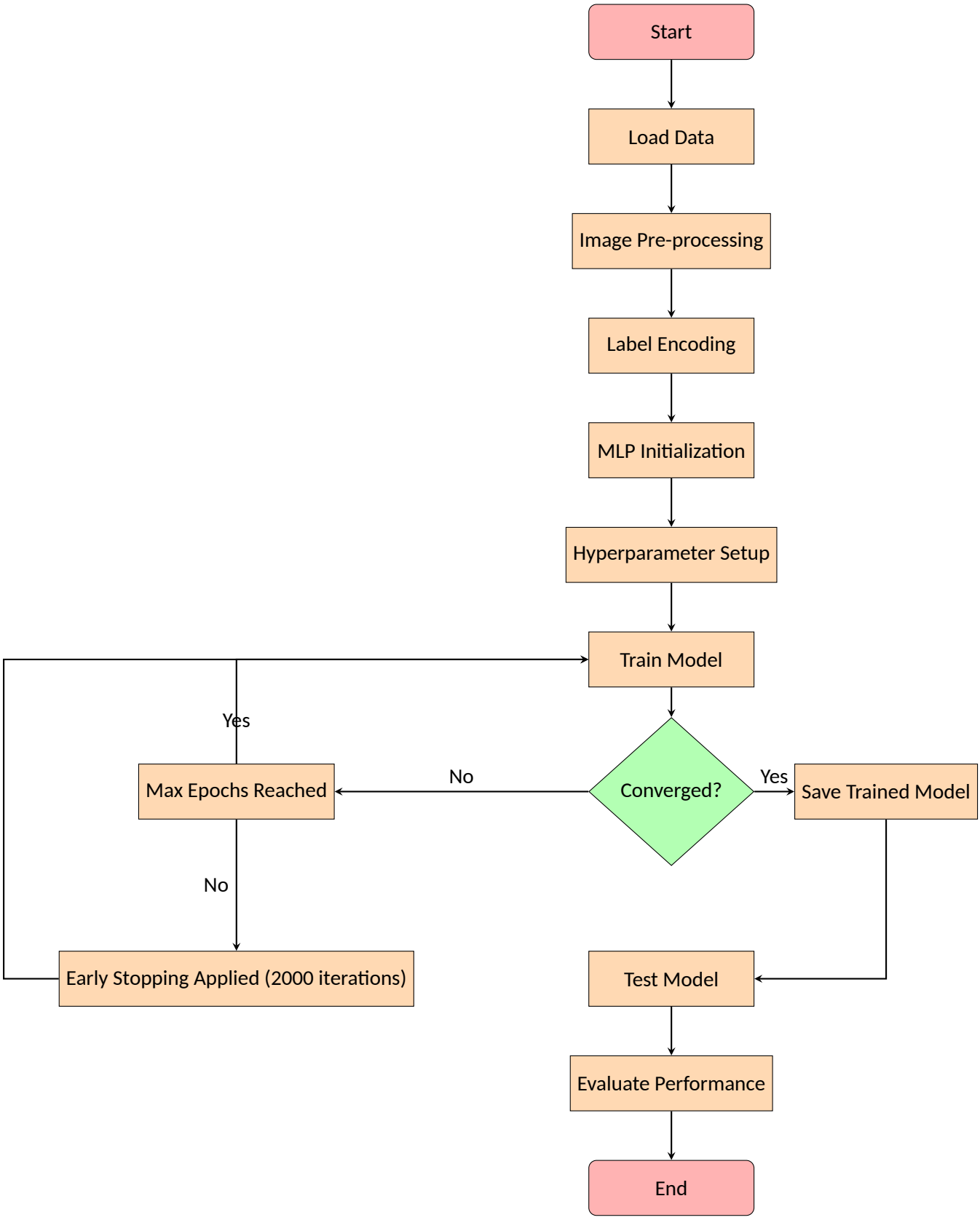

Figure 25: Flowchart for Developing MLP Network

Below we can see the image classification using MLP, we can see the letters being classified with near 100% accuracy for our microchip image.

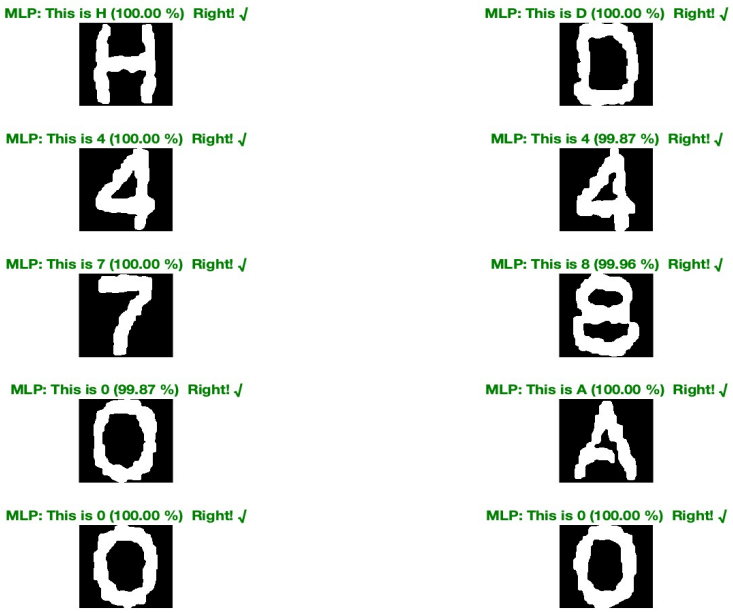

Figure 26: MLP based Classification of Images

To run the code, first run the code provided in Figure 27 to obtain a .NET framework integration that allows MATLAB to create objects and call for a method. Once this mlp.net file is created, run the code in Figure30 to obtain the final results.

5.3 MLP Code

```
clear; clc; close all;

% 128 * 128 = 16384
training = imageDatastore('p_dataset_26\dataset\train', ...
    'IncludeSubfolders', true, ...
    'LabelSource', 'foldernames');

testing = imageDatastore('p_dataset_26\dataset\test', ...
    'IncludeSubfolders', true, ...
    'LabelSource', 'foldernames');

sizee = 16384;
num_training = numel(training.Files);
num_testing = numel(testing.Files);

input_training = zeros(sizee, num_training);
label_training = zeros(1, num_training);

input_testing = zeros(sizee, num_testing);
label_testing = zeros(1, num_testing);

%%%%% 99999 pre-processing
```

```
for i = 1:num_training
    img = readimage(training, i);
    img = imbinarize(img);
    img = 1 - img;
    input_training(:, i) = img(:);
    label_training(i) = single(training.Labels(i));
end

for i = 1:num_testing
    img = readimage(testing, i);
    img = imbinarize(img);
    img = 1 - img;
    input_testing(:, i) = img(:);
    label_testing(i) = single(testing.Labels(i));
end

%%%%%%%%%
num_classes = 7; %0,4,7,8,A,D,H

label_training_onehot = full(ind2vec(label_training, num_classes));
label_testing_onehot = full(ind2vec(label_testing, num_classes));

hidden_layer = [128, 64];
net = patternnet(hidden_layer);

net.layers{3}.transferFcn = 'softmax';
net.trainParam.lr = 1e-3;

net.inputs{1}.processFcns = {};
net.outputs{2}.processFcns = {};

net.trainParam.epochs = 600;
net.trainParam.max_fail = 2000;
net.trainParam.goal = 0;
net.trainParam.time = 90;
net.trainParam.min_grad = 1e-100;

net.divideFcn = 'divideind';
net.divideParam.trainInd = 1:num_training;
net.divideParam.valInd = [];
net.divideParam.testInd = [];

[net, tr] = train(net, input_training, label_training_onehot, 'useGPU', 'yes');
%[net, tr] = train(net, input_training, label_training_onehot);

test_ok = net(input_testing);

predicted_labels = vec2ind(test_ok);

correct_rate = sum(predicted_labels == label_testing) / num_testing;
disp(['Accuracy: ', num2str(correct_rate * 100), '%']);

save('mlp_test.mat', 'net');
```

Figure 27: MLP Method Source Code

```
%% MLP Classification Method
load('mlp.mat', 'net');
label_map = {'0', '4', '7', '8', 'A', 'D', 'H', 'M'} ;
sizee = 128 * 128;
input_img_MLP = zeros(sizee, nummm);

```

```
for i = 1:nummm
    input_img_MLP(:, i) = test_img{i}(:);
end

char_pred_MLP = net(input_img_MLP);
predict_MLP = vec2ind(char_pred_MLP);

figure(10);
set(gcf, 'Position', [100, 100, 800, 600]);
for i = 1:nummm
    predict_num_MLP = predict_MLP(i);
    predict_letter_MLP = label_map{predict_num_MLP};
    acc_MLP = max(char_pred_MLP(:, i));
    accs_MLP = num2str(acc_MLP * 100, '%.2f');

    subplot(ceil(nummm/2), 2, i);
    imshow(test_img{i});
    if predict_letter_MLP == chaaab(i)
        title(['MLP: This is ', predict_letter_MLP, ' (', accs_MLP, ' %)' , '  Right
            ! √'], 'Color', '[0 0.5 0]');
    else
        title(['MLP: This is ', predict_letter_MLP, ' (', accs_MLP, ' %)' , '  Wrong
            ! ×' ...
            ''], 'Color', 'red');
    end

end
```

Figure 28: MLP Method Source Code

6 Results and Discussion

Aspect	MLP-Based Approach	CNN-Based Approach
Architecture	Fully connected layers, requires flattening the image.	Uses convolutional layers that extract spatial features.
Effectiveness	Works well on simple image datasets but struggles with complex patterns and spatial dependencies.	Excellent at capturing spatial hierarchies and patterns, making it more effective for image classification.
Efficiency	Requires more layers to capture complex data, but it requires fewer computations per layer.	Require fewer layers because of its efficient feature extraction, but it is computationally more expensive per layer.
Pre-Processing	Requires manual flattening and resizing. Sensitive to image size and data representation.	Less dependent on pre-processing, as it automatically extracts features.
Hyperparameter Sensitivity	Sensitive to tuning learning rate, number of neurons, batch size, and regularization.	Sensitive to hyperparameters but more robust to minor changes.
Performance on Images	Lack of spatial awareness, compared to CNN, causing it to underperform in image classification	Designed to capture spatial features direction, which results in CNNs outperforming MLPs.

Table 1: Comparison of MLP-Based and CNN-Based Method for Image Classification

The images above show MLP to be more certain than CNN in terms of being certain about what character it is identifying, but this is not accurate. We tested out this hypothesis using a rotated image, and we are able to see that CNN is not as certain on the number, but its classification is accurate vs. MLP that is a lot more certain on the classification but it is wrong as shown in Figure 29.

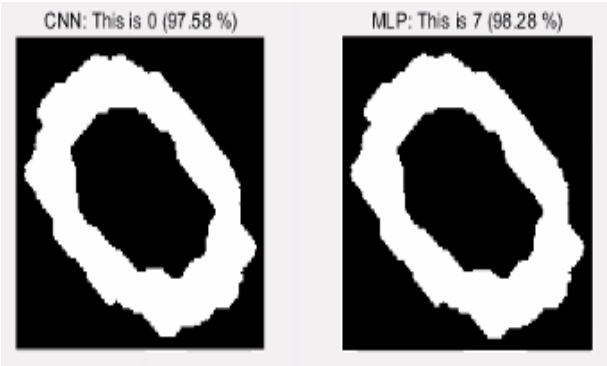

Figure 29: Testing CNN and MLP accuracy and certainty for rotated character

Based on the comparison table, we can see that CNN is a better network for predicting classifications of images. This is primarily because CNNs preserve spatial hierarchies. Making them more suitable to capture details in an image such as the edges and shapes. However, we have also noted that the predictions made by MLP network have a more confidence percentage than CNN. This is due to the robustness of CNN which limits its ability to have more confidence in the results even if the accuracy were to be higher. As shown in Figure 29, we can see that even though CNN has predicted the character correctly, it still

shows a lower confidence level overall. Therefore, with more fine tuning of hyperparameters and more training of the neural network, CNN. Will gradually give a higher confidence level for its predictions

```
%% Test CNN and MLP by Rotating the Character
rot = test_img{9};
angle = 45;

rott = padarray(rot, [64, 64], 0, 'both');
rottt = imrotate(rott, angle, 'bilinear', 'crop');
rotttt = imcrop(rottt, [64 64 127 127]);

predict_num_CNN = classify(net, rotttt);

pred_scores = predict(net, rotttt);
acc_CNN = max(pred_scores);
accs_CNN = num2str(acc_CNN * 100, '%.2f');

figure
subplot(1, 2, 1);
imshow(rotttt);
title(['CNN: This is ', char(predict_num_CNN), ' (', accs_CNN, ' %)']);

input_data = rotttt(:);
load('mlp.mat', 'net');

char_pred_MLP = net(input_data);
predict_num_MLP = vec2ind(char_pred_MLP);
predict_letter_MLP = label_map{predict_num_MLP};

acc_MLP = max(char_pred_MLP);
accs_MLP = num2str(acc_MLP * 100, '%.2f');

subplot(1, 2, 2);
imshow(rotttt);
title(['MLP: This is ', predict_letter_MLP, ' (', accs_MLP, ' %)']);
```

Figure 30: Testing CNN and MLP accuracy using rotated image

7 Conclusion

In this project the team applied two different types of advanced machine learning algorithms for image processing and character recognition - Convolutional Neural Networks (CNN) and Multilayer Perceptron (MLP). Each algorithm has its own distinct strengths and weaknesses and our comparison highlights the difference between their effectiveness for image-based classification tasks.

The MLP was seen to perform well in learning non-linear relationships when it was provided raw pixel data, being able to handle the high-dimensional nature of the image inputs. This method was relatively simple to implements and provided a decent accuracy - specifically after we tuned the hidden layers and the activation functions. When comparing the MLP to CNN, we saw that the MLP's performance was limited due to its lack of inherent spatial awareness. Since MLP works by working on each pixel independently, it is seen to struggle with capturing the local spatial structure of the image, which is very important for character recognition tasks where the neighbouing pixels are often seen to hold related information.

Our results showed that CNN outperformed MLP in both accuracy and efficiency - a result of leveraging convolutional layers. These layers allow CNN to automatically learn the spatial hierarchies of features (i.e., shapes, edges, textures), which provide more meaningful patterns of the images. This feature extraction supported CNN to be the more superior classification performance, especially for complex characters or noisy images. The use of pooling layers helped reduce the dimensionality of the data without causing the image to lose important features, which makes the model a lot more computationally efficient and a lot less prone to over fitting the image data compared to MLP. A similar implementation can happen with MLP, but it requires larger hidden layers to achieve that, making it computationally more taxing and inefficient.

In terms of complexity, it was evident that CNN required more training time and therefore more resources - a result of its deeper architecture and convolutional operations. However, because it is more efficient in conducting feature extraction and has the ability to generalize, it is better suited for our specific image classification task. With larger complexity and the dataset provided, it is the more efficient choice for our project.

In conclusion, the MLP is a viable option when conducting simple image classification task with an easier implementation technique, but CNN is the more robust model for character recognition within the microchip label images. CNN's ability to capture spatial relationships and operate in complex and noisy data makes it the better option. The results emphasize the importance of choosing a model that can exploit the underlying structure of the data, with CNN being the clear choice for image-based tasks where spatial patterns are important for image classification [1].

References

[1] C. C. Kong and P. C. Y. Chen, "Me5411 - robot vision and ai," 2024, course lectures: 1-10, covering topics in Robot Vision and Artificial Intelligence. [Online]. Available: https://canvas.nus.edu.sg/courses/63262