Continuous Assignment Report

ME5411 Robot Vision and AI

Group 1

LIN YI (A0285080E)

LI HAOWEI (A0285319W)

SIMA KUANKUAN (A0284990M)

Abstract

This report presents the outcomes and implementation details of the continuous assignment for NUS ME5411. Each task is described in a structured manner, beginning with an introduction to the problem and its context to ensure a clear understanding of the objectives. The developed algorithms and processes are then explained in detail, supported by pseudo-algorithms to facilitate comprehension. Running effects, in the form of screenshots, are provided to illustrate the practical applications of the algorithm at each stage of image processing. The discussion and conclusion sections explore the rationale behind the chosen methodology, the investigations conducted during the project and potential directions for future research.

The team members collaborated effectively to complete this assignment. For the image processing task, LI HAOWEI focused on the average mask and rotation mask, SIMA KUANKUAN handled image binarization, and LINYI worked on contour extraction and image segmentation. For the character recognition task, SIMA KUANKUAN developed a CNN network from scratch and performed additional experiments, while LI HAOWEI implemented a pure MLP-based character classification. The team also worked together to create the report.

It is worth noting that the required tasks were implemented almost entirely from scratch, without relying on any off-the-shelf libraries or toolboxes. For more information about the implementation details, please refer to the source code included in the .zip file or visit the open-source repository at https://github.com/brian00715/CNN-From-Scratch.

Contents

1 Task 1

1.1 Introduction

In this section, we should display the picture on the screen.

1.2 Solution

First of all, we use the *imread()* function to read the image. The down-link code implements an image file called ”project1.jpg” and assigns the pixel matrix of the image to the variable I. And this file name needs to be enclosed in single quotation marks.

When it comes to picture displaying, we use the *imshow()* function to display the image.

```
1 A=imread('chapter1.jpg');
2 figure;
3 imshow(A)
```

Figure 1: Display the Figure on the Screen

1.3 Discussion and Conclusion

1.3.1 Principle of Functions

imread() is a function for reading pictures, which are stored in two dimensions (gray image) or three dimensions (color image). *imread()* can read the pixel matrix information of the picture and carry out subsequent processing.

imshow() uses the default display range for the image data type and optimizes the picture window, coordinate area, and image object properties to display the image.

1.3.2 Conclusion

Above all, image display is an important means to visually present data. It makes the data easier to understand and analyze. In image processing, by displaying the images in the

process of processing, the changes, features and processing results of the images can be visually observed.

At the same time, in the stage of algorithm development and debugging, the effect and correctness of the algorithm can be quickly verified by displaying images. This helps check that the processing steps are working as expected and helps locate and resolve the problem.

2 Task 2

2.1 Introduction

We should implement and apply a 3x3 average mask and a 3x3 rotation mask on the image, resulting in an blurred image with reduced noise. We use masks of different sizes and the mask is used to filter different times. Finally we compare and comment on the results of the respective image smoothing methods.

2.2 Solution

In the stage of average mask processing, we separated the image into 3 channels and calculated the average pixel intensity of a masked area of each pixel in every channel. The size of the mask is set to be 3x3.

Algorithm 1 Averaging Mask Algorithm

```
function AveragingMask(I)
    for each pixel (i, j) in Image do
        Avg ← average I(i − 1 : i + 1, j − 1 : j + 1)
        Filtered Image(i, j) ← Avg
    end for
    return Filtered Image
end function
```

The second inquiry is regarding rotating mask. The major idea of rotating mask is to find a mask area with lowest variance around a pixel and apply the average intensity of the selected mask area to the pixel. To implement this, we first used 2 matrices to store the mean and variance of each mask, which significantly reduced the computational complexity by avoiding a lot of repetitive calculations. Then the mask area with lowest variance was chosen for each pixel and the average intensity was given to the pixel.

2.3 Discussion and Conclusion

2.3.1 Principle of Average Mask

The average mask is a commonly used smoothing mask in image processing, which blurs the image by calculating the average intensity of the pixels in the neighborhood. This smoothing technique helps remove noise and, to a certain extent, blur the image to eliminate irrelevant details.

Figure 2: Figure after Averaging Mask

Algorithm 2 Rotating Mask Algorithm

```
function RotatingMask(I)
    for each pixel (i, j) in Image do
        Avg ← average I(i − 1 : i + 1, j − 1 : j + 1)
        Var ← variance I(i − 1 : i + 1, j − 1 : j + 1)
        Filtered Image(i, j) ← Avg(min(Var))
    end for
    return Filtered Image
end function
```

This mask is usually a square matrix where each element has the same weight and is used to calculate a new value for each pixel value in the image. In general, the values of the elements in this mask are all equal, so it is called an average mask.

For example, for a 3x3 average mask:

$$A = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} \tag{1}$$

Each element in this matrix is 1/9, representing an average processing of each pixel in the image and its surrounding neighborhood pixels.

2.3.2 Principle of Rotation mask

Rotating mask takes the variance of each mask area into consideration and choose the area with lowest variance, which will theoretically end up with a smoother image. Meanwhile, the perception field of each pixel is 5x5, greater the given 3x3 mask, which allows the algorithm to take a larger neighborhood area around each pixel into consideration.

Figure 3: Figure after Rotating Mask

However, due to the fact that 3x3 mask area is very small in compared with the whole image, these difference is not very obvious.

3 Task 3

3.1 Introduction

In this section, we should create an image that is a sub-image of the original image, containing the center line -HD44780A00. This means that we should cut the original image into smaller parts.

3.2 Solution

Image cropping process mainly use the function *imcrop(I,rect)*, and we should set the corresponding size of rectangle to fit the characters.

The actual size of the output image does not always exactly match the width and height specified by the rect. In our project, we use rect equal to [50 900 200 150]. This is the result of continuous debugging by our eyes, which may be optimized in the following project.

```
1 [x1,x2,y1,y2] = deal(50,950,200,350);
2 rect = [x1,y1,abs(x1-x2),abs(y1-y2)];
3 I1 = imcrop(I,rect);
```

The cropped image is shown in the following figure. It can be seen that there are only a line of letters and numbers in HD44780A00 in the image.

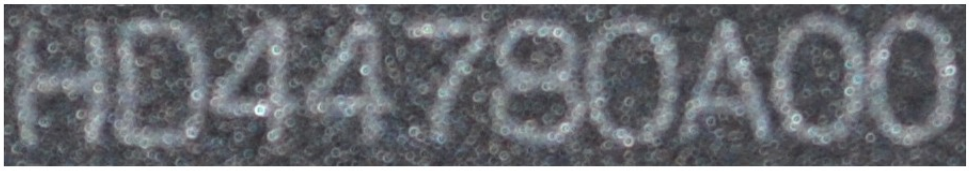


Figure 4: Figure after Cropping

3.3 Discussion and Conclusion

3.3.1 Principle of Functions

imcrop(I,rect) crop the image I according to the position and dimension specified in the crop rectangle rect. The cropped image includes all the pixels in the input image that are completely or partially surrounded by the rectangle.

3.3.2 Conclusion

In general, the role of cropping an image to a line of characters is to segment and process the text in the image for character recognition, text analysis, and subsequent data processing operations. This helps improve the accuracy and efficiency of character recognition systems and allows for deeper analysis of text.

However, the result is continuous debugged by our eyes, we can optimize the cropping method in the following project to work with characters in different size and length.

4 Task 4

4.1 Introduction

The given image, *charact2.bmp*, is a grayscale image stored in the computer as a pixel array. Each pixel is denoted by an integer ranging from 0 to 255. Let I be the digital image; hence, each pixel value can be accessed through $i = I(x, y)$, where x and y denote the row and column indices of the pixel.

If a threshold value T is provided, the process of binarizing a grayscale image is straightforward and can be expressed using the pseudo-algorithm 3.

4.2 Solution

Although image binarization is a straightforward process, the primary challenge lies in determining the optimal threshold value. Manual tuning is one approach, but it proves time-consuming and lacks the ability to automatically adapt to new images. Following thorough investigation, we recommend employing Otsu [1] method for our binarization technique, as detailed in pseudo-algorithm 4. The Otsu binarization algorithm, devised by Japanese researcher Nobuyuki Otsu, utilizes the image histogram. The core concept involves dividing the

Algorithm 3 Binarization Algorithm

function BinarizeImage(I, T)
  for each pixel (i, j) in Image **do**
    if Intensity$I(i, j) >$ Threshold T **then**
      Set I(i, j) to 1 ▷ White
    else
      Set I(i, j) to 0 ▷ Black
    end if
  end for
  return Binarized Image
end function

histogram into two classes with maximum inter-class variance. These two intensity classes in the histogram represent the foreground and background, respectively.

Algorithm 4 Otsu Thresholding Algorithm

function OtsuThresholding(Image I)
  Compute histogram H for I
  Normalize histogram to get probabilities $P(i)$ for each intensity level i
  Initialize variables: $\sigma_b^{\text{max}} \leftarrow 0$, $T \leftarrow 0$
  for $t \leftarrow 1$ to 255 **do** ▷ Iterate through possible thresholds
    Compute probabilities $P_1(t)$ and $P_2(t)$ for classes separated by threshold T
    Compute class means $\mu_1(t)$ and $\mu_2(t)$
    Compute between-class variance $\sigma_b(t) = P_1(t) \cdot P_2(t) \cdot (\mu_1(t) - \mu_2(t))^2$
    if $\sigma_b(t) > \sigma_b^{\text{max}}$ **then**
      $\sigma_b^{\text{max}} \leftarrow \sigma_b(t)$
      $T \leftarrow t$
    end if
  end for
  return I_{sem}
end function

We implemented the Otsu threshoding algorithm in the following form of API.

```
>> myOtsuThres.m
function best_threshold = myOtsuThres(img_gray)
...
end

>> task1_6.m
...
opt_thres = myOtsuThres(img);
img_bin = imbinarize(img, opt_thres);
...
```

Further more, we implemented erosion and dilation before binarize the image. Erosion and dilation are two fundamental operations in morphological image processing, a branch of

mathematical morphology that deals with the shape and structure of objects. Erosion is a morphological operation that ”erodes” or shrinks the boundaries of objects in an image. The basic idea is to use a structuring element (a small, predefined pattern or kernel) and slide it through the image, setting the pixel intensity to 1 or 0 according to the nearby area. The result of erosion is that the boundaries of objects in the image are eroded away, and small or thin structures are removed. Dilation is the opposite of erosion; it expands the boundaries of objects in an image. Similar to erosion, it involves sliding a structuring element through the image. The result of dilation is that objects become thicker and small gaps between objects may be filled.

For the provided image, Otsu's calculated optimal threshold value is 0.35 (the image is normalized from $[0, 255]$ to $[0, 1]$), and the binarization result is depicted in Figure 5.

Figure 5: Comparison of the original image (top) and the Otsu-binarized image (bottom).

4.3 Discussion and Conclusion

Task 4 aims to simplify images into binary representations for applications like object recognition. The challenge was selecting an optimal threshold, traditionally done manually. We addressed this using Otsu's method, a dynamic algorithm maximizing between-class variance. The implementation achieved effective binarization with a threshold of 0.35 (normalized). Visually, Otsu's method demonstrated clear separation between foreground and background, showcasing its effectiveness in enhancing image interpretability.

In conclusion, Otsu's Thresholding Algorithm emerged as a robust solution for image binarization, effectively addressing manual tuning challenges and enhancing interpretability.

5 Task 5

5.1 Introduction

After obtaining the binary image, the next stage is to contour extraction from binary image, which helps more clearly reflect the important events and changes of the image property.

In this section, we mainly use Sobel Operator and Canny Operator to contour extraction from binary image. Then we compare the advantages and disadvantages between these to operators.

5.2 Solution

5.2.1 Sobel Operator

This code implements the Sobel edge detection algorithm, and thresholding the results, setting pixels greater than 0.7 as 1, and pixels less than or equal to 0.7 as 0.

x, y, z = size(I) takes the dimensions of the image I and assigns them to the variables x, y, and z, respectively. Where x and y represent the height and width of the image.

I_sobelx and *I_sobely* are two matrices of the same size as I_sobel, they are created to store the results of the Sobel operator in the x and y directions.

The following code uses nested loops to traverse the pixels in the image, applying the Sobel operator to each pixel for edge detection:

I_sobelx(i, j) is the edge intensity of the image in the x direction is calculated in line 8.

I_sobely(i, j) is the edge intensity of the image in the y direction is calculated in line 10.

And in line 13, the edge intensity of x direction and y direction is superimposed to obtain sobel, and the complete Sobel edge detection results are stored in sobel.

```
1  [x,y,z]=size(I);
2  I_sobel=zeros(x,y);
3  I_sobelx=I_sobel;
4  I_sobely=I_sobel;
5  for i=2:x-1
6      for j=2:y-1
7          I_sobelx(i,j)=abs(I(i-1,j+1)-I(i-1,j-1)+
8          2*I(i,j+1)-2*I(i,j-1)+I(i+1,j+1)-I(i+1,j-1));
9          I_sobely(i,j)=abs(I(i-1,j-1)-I(i+1,j-1)+
10         2*I(i-1,j)-2*I(i+1,j)+I(i-1,j+1)-I(i+1,j+1));
11     end
12 end
13 I_sobel=I_sobelx+I_sobely;
14 for i=1:x
15     for j=1:y
16         if I_sobel(i,j)>0.7
17             I_sobel(i,j)=1;
18         else
19             I_sobel(i,j)=0;
20         end
21     end
22 end
```

The image of edge extraction by sobel operator is shown as the following figure. It can be seen that some small area regions still exist after binarization, so these regions are also extracted in the process of edge extraction. But the overall effect is still clear.

5.2.2 Canny Operator

The Canny Contour Extraction consists of several steps: smoothing, computed gradient, non-maximum suppression, and double-threshold edge tracking.

Figure 6: Contour Extraction using Sobel Operator

The first step is Gaussian Smoothing. The image is smoothed by using a Gaussian filter to reduce the influence of noise.

Then we use Gradient Calculation to calculate the gradient on the smoothed image. In this section, we use the Sobel operator to calculate the gradient amplitude and direction of each pixel in the image. In the line 6 of our code, we convert gradient direction to 0-180 degree range.

The following stage is Non-maximum Suppression. Only local maxima in the direction of the gradient are retained to refine the edges. In our Matlab code, we set the high threshold to 20% of the maximum gradient amplitude and low threshold is 10%.

Final step is Double Thresholding and Edge Tracking by Hysteresis. It is based on two thresholds, determine which edges are true edges and which are noise. The connection between the strong edge and the weak edge to form a complete edge.

```
1  smoothedImage = imgaussfilt(I, 1);
2  [Gx, Gy] = imgradientxy(smoothedImage, 'sobel');
3  edgeMagnitude = hypot(Gx, Gy);
4  edgeDirection = atan2d(Gy, Gx);
5  edgeDirection(edgeDirection < 0) =
6  edgeDirection(edgeDirection < 0) + 180;
7  highThreshold = 0.2 * max(edgeMagnitude(:));
8  lowThreshold = 0.1 * max(edgeMagnitude(:));
9  edgeImage = edgeMagnitude > highThreshold;
10 edgeImage = edgeImage & (edgeMagnitude > lowThreshold);
```

Figure 7: Contour Extraction using Canny Operator

It can be found from the figure that the contour extraction effect of Canny operator is better than that of Sobel operator, and its edge is clearer and the detection accuracy is higher.

5.3 Discussion and Conclusion

After computing these different kind of operators in table 1, we finally choose Sobel operator and Canny operator to process our binary figure, since they are effect and suitable for gray gradient.

5.3.1 Principles of Sobel Operators

In sobel operator, they respectively represent the edge detection operators for X axis and Y axis. From the operator structure, it can be clearly found that this filter is to calculate the difference of the gray value of the current pixel on the right side and the left side of the 8 connected pixels.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \tag{2}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3}$$

The Sobel operator is commonly used in image processing to perform edge detection by convolving the image with two 3x3 kernels (matrices). One kernel is for detecting edges in the horizontal direction (G_x) and the other is for the vertical direction (G_y).

Finally, by calculating the gradient image obtained by the horizontal and vertical Sobel operators, the magnitude and direction of the gradient can be calculated by the following formula:

$$GradientMagnitude = \sqrt{G_x^2 + G_y^2} \tag{4}$$

$$GradientDiraction = \left(\frac{G_y}{G_x}\right) \tag{5}$$

5.3.2 Principle of Canny Operator

The Canny edge detection algorithm consists of several steps: smoothing, computed gradient, non-maximum suppression, and double-threshold edge tracking, all of which are listed in the code presentation above.

5.3.3 Conclusion

Character contour extraction helps to extract character shape and contour features. Character outline is one of the key features in character recognition, which can help distinguish different characters and provide the information needed for recognition.

Table 1: Comparison of Different Contour Extraction Operators

Operator	Comparison of advantages and disadvantages
Roberts	The image with steep and low noise has a good processing effect, but the edge extracted by Roberts operator results in thicker edges, so the edge positioning is not very accurate.
Sobel	The image processing effect is better for gray gradient and more noise, and the Sobel operator is more accurate for edge location.
Kirsch	It is good for gray gradation and noisy image processing.
Prewitt	It is good for gray gradation and noisy image processing.
Laplacian	It is accurate to locate the step edge points in the image, is very sensitive to noise, and loses some edge direction information, resulting in some discontinuous detection edges.
LoG	LoG operator often appears double-edge pixel boundaries, and this detection method is more sensitive to noise, so the LoG operator is rarely used to detect edges, but to determine whether edge pixels are located in the bright area or the dark area of the image.
Canny	This method is not easily disturbed by noise and can detect real weak edges.

At the same time, it can emphasize the character edge and help to reduce the influence of noise on character recognition and improve the recognition accuracy.

In our experiment, because in the previous noise reduction and smoothing processing, there are still a number of small areas that are not removed, and the edge of the character is relatively rough, which leads to the generation of noise in the contour extraction.

6 Task 6

6.1 Introduction

The following stage of Image processing is image segmentation, It is the process of dividing an image into multiple regions or parts. In character recognition or text detection, image segmentation can help realize that by splitting an image, different characters can be separated so that each character becomes an independent image unit. Such segmentation can help with character recognition, because independent characters are easier to recognize and classify.

6.2 Solution

Since the letter width in a given image is nearly uniform and the spacing is consistently distributed, we employ a technique to address the aforementioned issue of characters sticking together. Initially, we determine the minimum width of the segmented characters, a crucial factor associated with successful segmentation. Subsequently, for each separated sub-image, we compare its width to the minimum width. If the sub-image width is 1.8 times greater than the minimum, we identify it as a sticking segmentation. We then horizontally divide it into two equal images, constituting the final segmentation result. The final result is depicted in Figure **??**, showcasing perfectly separated characters.

The following code is our main implementation code in Matlab.

```
if j - temp > set3 && j - temp < set4
    [x1,x2,y1,y2] = deal(temp,j,0,m);
    I1 = imcrop(I,[x1,y1,abs(x1-x2),abs(y1-y2)]);z
    temp = j;
    figure
    imshow(I1)
elseif j - temp > set4
    [x1,x2,y1,y2] = deal(temp,(j+temp)/2,0,m);
    I1 = imcrop(I,[x1,y1,abs(x1-x2),abs(y1-y2)]);
    figure
    imshow(I1)
    [x1,x2,y1,y2] = deal((j+temp)/2,j,0,m);
    I1 = imcrop(I,[x1,y1,abs(x1-x2),abs(y1-y2)]);
    temp = j;
end
```

We firstly set some thresholds and parameters. *set1* is the minimum value of the RGB change. *set2* is cut part of the maximum RGB value. And *set3* is minimum character spacing. *set4* is maximum distance between characters.

Then we divide the characters according to the RGB value change of the column. We use *col_A = sum (I)* to calculates the sum of the RGB values for the image column. Then loop through the RGB values of the column, find the point of change of the column according to the set thresholds *set1* and *set2* , and record their position.

Finally we finish image segmentation and display according to character spacing. We use the *imcrop()* function to crop the image according to the change points found earlier to get a single character image. Depending on the set minimum character spacing *set3* and maximum character spacing *set4*, determine the different cases of characters (possibly two characters are joined together or single characters), and different clipping operations are performed. Finally, the *imshow* function is used to display the segmented character image.

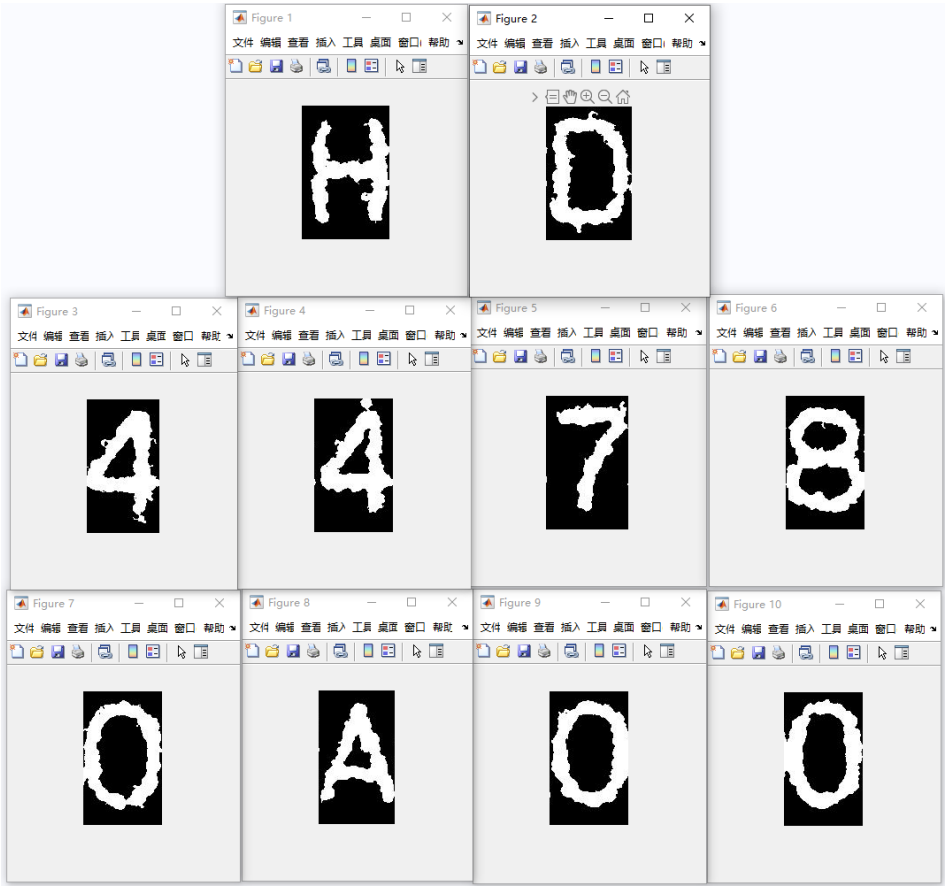


Figure 8: Single Characters after cropping

6.3 Discussion and Conclusion

This process of splitting an entire line of text into a single character is the first step in optional character recognition (OCR), which separates a line of text containing multiple char-

acters into separate character images for subsequent character recognition processing.

At the same time, single character image is more suitable for character recognition. By cropping the image into a single character, each character can be entered independently into the character recognition system for recognition and analysis. This can improve the accuracy and reliability of character recognition.

In our experiment, although most of the characters can be divided, two or more characters will still be connected. In this case, we will crop the characters according to the normal value after detecting the abnormal horizontal length. This cropping method can be improved in the future experiment.

7 Task 7

7.1 Sub-task 1

When recognizing or classifying a character within an image, an intuitive approach involves leveraging its features. Prior to the emergence of convolutional neural networks (CNN), traditional pattern recognition entailed designing a series of handcrafted feature extractors to gather relevant information from the input. Subsequently, this information was utilized for manual character classification. The advent of trainable multi-layer perceptrons (MLPs) enhanced the simplicity of recognition tasks, thanks to their excellent function-fitting capabilities. However, MLPs struggle when confronted with larger datasets, such as high-resolution images, due to their limited parameter count and lack use of spatial information. In such scenarios, CNNs prove instrumental in implementing character recognition within larger image formats.

7.1.1 Convolutional Neural Network Design

The convolutional neural network usually involves a sequence of cascaded convolutional layers for extracting features. Following the last convolutional layer, an MLP is added to merge features and aid in classification. In contrast to traditional rule-based feature classifiers, a notable aspect of CNN is the use of trainable convolution kernels employing techniques like Stochastic Gradient Descent (SGD). This signifies the network's adaptability to the dataset, enabling the automatic and inherent extraction of desired input features. Each convolutional layer comprises multiple channels with randomly initialized parameters, promoting the learning of features from different "sub-spaces." Furthermore, convolutional layers are commonly succeeded by a pooling layer, employed for sub-sampling feature maps to reduce parameters and improve recognition efficiency.

Given the effectiveness of LeNet [2] in manuscript recognition, we embraced a comparable network architecture. Indeed, we investigated diverse network designs, experimenting with parameters such as layer count, convolutional kernel size, and channel quantity. Ultimately, we selected the most potent configuration, and the details are as follows.

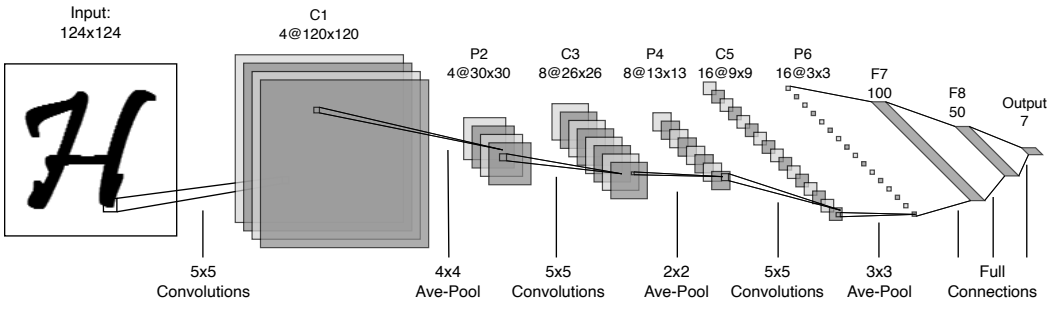


Figure 9: Neural network architecture.

The overall architecture of our network is depicted in Fig. 9. Comprising three convolutional layers, each followed by a pooling sub-sampling layer, the network maintains consistent 5x5 convolutional kernel sizes. The initial convolutional layer starts with 4 channels, doubling

sequentially. Pooling steps are set at 4, 2, and 3, respectively. Following the last convolutional layer, the resulting 32 3x3 feature maps are flattened into a 144-dimensional vector. This vector is then fed into a Multi-Layer Perceptron (MLP) with two hidden layers, featuring 100 and 50 units, respectively. The MLP concludes with a softmax output layer containing 7 units aligned with our classification task. We applied a ReLU activation function to the output of each layer to enable the non-linear fitting ability of the network.

To ensure compatibility between the input image and our designed network, the original 128x128 dimensions are resized to 124x124. The reasons that we designed such a CNN are as follows.

Firstly, we observe that many characters in the given dataset have fine-grained features like the small changes in curvature of handwritten $\mathcal{H}$ as shown in Fig. 9, which necessitate a network that preserves input dimensionality. Consequently, a relatively large feature map is maintained at the network's inception. This design extends to the convolutional kernel, which must be compact to facilitate fine feature extraction.

Secondly, the dataset's scale is smaller than common ones like MNIST [3]. To prevent overfitting, the model's size should be kept moderate. Considering both model scale and fine features, we introduced a large average pooling layer (P2) with a step of 4, followed by layer C1. This strategic choice significantly reduces the feature map size, resulting in a more compact model architecture for subsequent layers. Consequently, the final 16 3x3 feature maps are flattened into a concise feature vector (P6).

The design of MLPs (F7)-(F8) adheres to common principles. Additionally, to mitigate overfitting, a dropout layer is incorporated after the full-connected layer (F7).

7.1.2 Training Process

As we implemented the CNN based on MATLAB without using any toolboxs, some important derivations must be described as follows. For clarity, the notations used are listed in Table 2.

Table 2: Notations

l	Layer index	N^l	Size of input image
M^l	Size of kernel	D^l	Size of feature map
$\mathbf{X}^l$	Input image of layer l	x_{ij}^l	An element of $\mathbf{X}^l$
$\mathbf{W}^l$	Kernel of layer l	w_{ij}^l	An element of $\mathbf{W}^l$
$\mathbf{V}^l$	Feature map of layer l	v_{ij}^l	An element of $\mathbf{V}^l$

- **Forward Propagation**

  Forward propagation is a direct process, involving computations for convolution, average-pooling, dropout, and the weighted sum in MLP. For a certain convolutional layer v_{ij}^l, each element of the current feature map can be written as:

$$v_{i,j}^l = \sum_{p=1}^{M^l} \sum_{q=1}^{M^l} w_{p,q}^l \cdot x_{i+p-1,j+q-1}^{l-1} + b_{i,j}^l, \quad x_{i,j}^l = \varphi\left(v_{i,j}^l\right) \tag{6}$$

where $\varphi(\cdot)$ represents the activation function. In our case, the ReLU activation function we used is:

$$\text{ReLU}(x) = \begin{cases} 0, & \text{if } x < 0, \\ x, & \text{if } x \geq 0. \end{cases} \tag{7}$$

The forward propagation for MLP is actully the weightd sum of the last layer's output plus a bias term. That is

$$v_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} x_i^{(l-1)} + b_i^l, \quad x_j^{(l)} = \varphi\left(v_j^{(l)}\right) \tag{8}$$

- **Loss Computation**

  We adopted cross-entrophy loss function considering the given classification task, which can be written as:

$$E(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i) \tag{9}$$

  where y_i is the true probability of class i, $\hat{y}_i$ is the predicted probability of class i.

- **Backward Propagation**

  The backward propagation aims to propagate the gradient of loss w.r.t. to each trainable weights. The general form of gradients can be represented as

$$\nabla_\theta E = \left(\frac{\partial E}{\partial \theta_1}, \frac{\partial E}{\partial \theta_2}, \ldots, \frac{\partial E}{\partial \theta_n}, \ldots\right) \tag{10}$$

  Where θ denotes the collection of all trainable parameters in neural networks. Calculating gradients entails determining the partial derivative of the loss concerning each parameter. A comprehensive derivation of these partial derivatives will be provided in the subsequent sections.

  As we adapted softmax output layer and the cross-entrophy loss, the gradient w.r.t to the output layer is actually

$$\begin{aligned} \frac{\partial E_i}{\partial v_j} &= -\frac{\partial}{\partial v_j}(y_i \log(\hat{y}_i)) \\ &= -\frac{\partial}{\partial v_j}(y_i \log(\frac{e^{v_j}}{\sum_{m=1}^K e^{v_m}})) \\ &= -\frac{\partial}{\partial v_j}(y_i \cdot (v_j - \log(\sum_{m=1}^K e^{v_m}))) \\ &= -y_i + y_i \cdot \frac{e^{v_j}}{\sum_{m=1}^K e^{v_m}} \\ &= \hat{y}_i - y_i. \end{aligned} \tag{11}$$

  Where K is the number of the class to be predicted. Then, the partial derivative of loss

w.r.t the hidden layers of MLP can be computed as:

$$\begin{aligned}\frac{\partial E}{\partial w_{ji}^{(l)}} &= \left(\frac{\partial E}{\partial v_j^{(l)}}\right)\left(\frac{\partial v_j^{(l)}}{\partial w_{ji}^{(l)}}\right) \\ &= \delta_j^{(l)} x_i^{(l-1)}\end{aligned} \tag{12}$$

Where

$$\delta_j^{(l)} = \left(\sum_{m=0}^{n_{l+1}} \delta_m^{(l+1)} w_{mj}^{(l+1)}\right) \varphi'\left(v_j^{(l)}\right) \tag{13}$$

For the partial derivative of convolutional layers w.r.t the weights in the convolutional kernels, the formula adheres to a comparable chain rule paradigm, distinguished by the convolution computing. We then have

$$\begin{aligned}\frac{\partial E}{\partial w_{h,k}^l} &= \left(\frac{\partial E}{\partial \mathbf{V}^l}\right)\left(\frac{\partial \mathbf{V}^l}{\partial w_{h,k}^l}\right) \\ &= \sum_{i=1}^{D^l}\sum_{j=1}^{D^l}\left(\frac{\partial E}{\partial v_{i,j}^l}\right)\left(\frac{\partial v_{i,j}^l}{\partial w_{h,k}^l}\right) \\ &= \sum_{i=1}^{D^l}\sum_{j=1}^{D^l}\delta_{i,j}^l \varphi'\left(v_j^{(l)}\right)\end{aligned} \tag{14}$$

Where

$$\delta_{i,j}^l = \frac{\partial E}{\partial v_{ij}^l} = \sum_{m=1}^{M^{l+1}}\sum_{n=1}^{M^{l+1}} \delta_{i-m+1,j-n+1}^{l+1} \cdot w_{m,n}^{l+1} \cdot \varphi'\left(v_{i,j}^l\right) \tag{15}$$

Similarly, the partial derivitive w.r.t. the bias term can be written as:

$$\frac{\partial E}{\partial b_{h,k}^l} = \sum_{i=1}^{D^l}\sum_{j=1}^{D^l}\frac{\partial E}{\partial v_{i,j}^l}\frac{\partial v_{i,j}^l}{\partial b_{h,k}^l} = \sum_{i=1}^{D^l}\sum_{j=1}^{D^l}\delta_{i,j}^l \tag{16}$$

- **Parameter Update**

  Once we obtain the partial derivative or the gradient of the loss with respect to the parameters, we can then formulate the following parameter update rule based on SGD:

$$\begin{aligned}\theta(k+1) &= \theta(k) + \Delta\theta(k) \\ &= \theta(k) + \eta\nabla_\theta E(k)\end{aligned} \tag{17}$$

  Where k and $k+1$ represent the current and the subsequent training iterations, respectively. η denotes the learning rate, a hyper-parameter. To expedite training convergence, surmount local minima, and alleviate shocks in the training process, we imple-

mented the momentum parameter updating rule. The formula is then modified to

$$\begin{aligned} v(k+1) &= \gamma v(k) + (1-\gamma)\nabla_\theta E(k) \\ \theta(k+1) &= \theta(k) - \eta v(k) \end{aligned} \tag{18}$$

Where γ is a hyperparameter, typically set between 0.9 and 0.99.

- **Learning Rate Adjustment**

  The choice of an appropriate learning rate is crucial in shaping the training dynamics of neural networks, impacting factors such as convergence speed, overall performance, and generalization ability. Although fixed learning rates are commonly used, their limitations become evident when dealing with the complex and evolving landscapes of loss functions associated with deep neural networks. In contrast, learning rate schedules, like linear decay, offer a dynamic and adaptive approach to fine-tune the learning rate throughout the training process.

  When comparing the performance of fixed learning rates with dynamically scheduled ones, the drawbacks of a fixed approach become apparent. Fixed learning rates, while simple, may encounter challenges in navigating through regions of high curvature, plateaus, or abrupt changes in the loss landscape. The lack of adaptability can result in suboptimal convergence, oscillations, and difficulties in escaping local minima.

  Enter learning rate schedules, dynamically adjusting the learning rate, providing a more nuanced approach to exploration and exploitation during training. In our implementation, we adopted the linear decay schedule, gradually decreasing the learning rate over time. This aligns with the notion that a larger learning rate fosters exploration in the early training stages, while a smaller learning rate benefits precise exploitation as the model approaches convergence. The formula can be expressed as:

$$\eta(t+1) = \eta(t)\left(1 - \frac{t}{T_{\text{max}}}\right) \tag{19}$$

  Where t and $t+1$ denote the current and the next epoch. It exemplifies a linear reduction in the learning rate over iterations, providing a straightforward yet effective strategy for adjusting the step size.

  In essence, the adoption of learning rate schedules, illustrated by linear decay, emphasizes the importance of a thoughtful and dynamic approach to learning rate adjustments. The adaptive nature of these schedules empowers neural networks to navigate the intricate terrains of the loss landscape, facilitating more effective training and enhancing the robustness of the trained models.

Above all, the entire CNN training process we utilized can be concisely summarized as a pseudo algorithm 5. Guided by this pseudo algorithm, we instantiated the complete CNN architecture using MATLAB. Key APIs are outlined as follows:

```
>> train.m
...
# define the network architecture
```

```
cnn.layers = {
struct('type', 'input')
struct('type', 'Conv2D', 'filterDim', 5, 'numFilters',
4, 'poolDim', 4, 'actiFunc', 'relu')
struct('type', 'Conv2D', 'filterDim', 5, 'numFilters',
8, 'poolDim', 2, 'actiFunc', 'relu')
struct('type', 'Conv2D', 'filterDim', 5, 'numFilters',
16, 'poolDim', 3, 'actiFunc', 'relu')
struct('type', 'Linear', 'hiddenUnits', 100,
'actiFunc', 'relu', 'dropout', 0.2)
struct('type', 'Linear', 'hiddenUnits', 50,
'actiFunc', 'relu')
struct('type', 'output', 'softmax', 1)
};

# set training hyper-parameters
train_options.epochs = 80;
train_options.minibatch = 64;
train_options.lr_max = 0.1;
train_options.lr_min = 1e-5;
train_options.lr_method = 'linear';
train_options.momentum = 0.9;

# start training
trainMachine(cnn, dataset_options, train_options,
data_train, labels_train, data_test, labels_test);

>> trainMachine.m
function flag = trainMachine(cnn, dataset_options,
train_options, data_train, labels_train, data_test,
labels_test)
...
# initilize model parameters
cnn = initModelParams(cnn, data_train, numClasses);
cnn = learn(cnn, data_train, labels_train, data_test,
labels_test, train_options); # start learning
...

>> learn.m
function [cnn_final] = learn(cnn, data_train,
labels_train, data_test, labels_test, options)
...
# retrieve a minibatch of dataset
mb_data = data_train(:, :, :, rp(s:s + minibatch - 1));
mb_labels = labels_train(rp(s:s + minibatch - 1));

```

```
# forward propagation
cnn = forward(cnn, mb_data, options);
# calculate loss
[cnn, curr_loss] = calcuLoss(cnn, mb_data, mb_labels,
options);
# backward propagation
grad = backward(cnn, mb_data, options);

# parameters updating with momentum
velocity = mom * velocity + (1 - mom) * grad;
theta = theta - lr * velocity;
cnn = updateWeights(cnn, theta);
...
```

Algorithm 5 CNN Training

Input: Training dataset D, CNN model parameters θ, initial learning rate η_{max}, minimum learning rate η_{min}, momentum β, number of epochs N
Output: Trained CNN model
for $epoch \leftarrow 1$ to N **do**
    $\eta(t+1) = \eta(t)\left(1 - \frac{t}{T_{\text{max}}}\right)$ ▷ Linear Annealing
    for each mini-batch (X, y) in D **do**
        $h \leftarrow \text{ForwardPropagation}(X, \theta)$ ▷ Forward Propagation
        $E \leftarrow \text{ComputeLoss}(h, y)$ ▷ Compute Loss
        $\nabla_\theta E \leftarrow \text{BackwardPropagation}(X, y, \theta)$ ▷ Backward Propagation
        $\theta \leftarrow \text{UpdateParameters}(\theta, \nabla_\theta E, \eta, \gamma)$ ▷ Parameter Update with Momentum
    end for
end for

7.2 Sub-task 2

7.2.1 MLP Network

The non-CNN-based method selected in this project is MLP. We implemented MLP with different size in each layer and resizing input image is also implemented for comparisonn. To be specific, input images were resized from 128*128 to 64*64 and 32*32 in different experiments.

The following figure shows the structure of the MLP we found most effective. The network takes original 128*128 image as input but resize that to 16384*1. In the next 2 hidden layers, the network resizes the features from the previous layer to 1024*1 and 256*1. The output layer keeps resizing to finally reach output of 7*1.

The activation applied in each layer except the output layer was $Sigmoid$ and $Softmax$ was applied to the last layer for better performance in classification task.

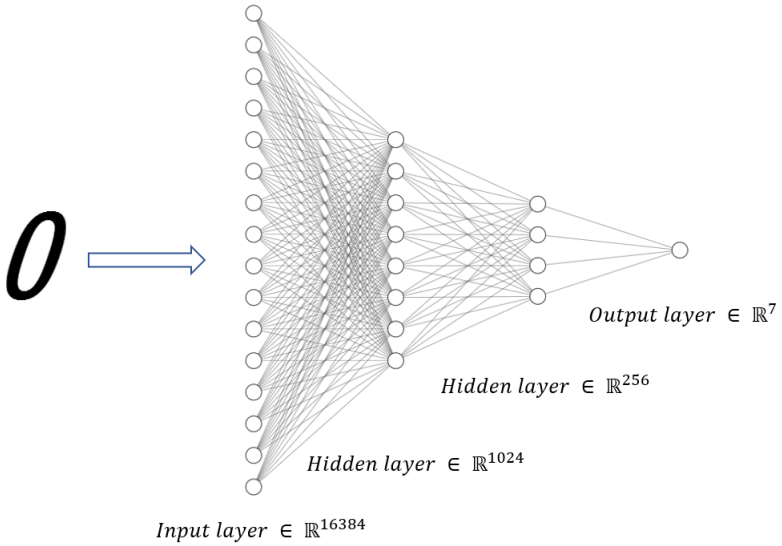


Figure 10: MLP Network Structure

7.2.2 MLP Training

The MLP network was implemented based on MATLAB without using any toolboxs. The steps described as follows. For clarity, the notations used are listed in Table 3.

Table 3: Notations

l	Layer index	L	the Last Layer
$\mathbf{b}^l$	Bias of the layer	$\mathbf{W}^l$	Weights of the layer
$\mathbf{Z}^l$	Linear output	$\mathbf{A}^l$	Activation output
$\hat{\mathbf{Y}}$	Final output	$\mathbf{Y}$	Ground truth

- **Forward Propagation**

  In this step, the input image was resized to a vector and processed by each layer to compute the final output. In each layer, the size of the input was changed by a matrix multiplication with the weights and bias was added in this process.

$$\mathbf{Z}^l = \mathbf{W}^l * \mathbf{A}^{l-1} + \mathbf{b}^l \tag{20}$$

  The result from the linear part passed through a $Sigmoid$ activation function and went into the next layer

$$\mathbf{A}^l = \text{Sigmoid}(\mathbf{Z}^l) \tag{21}$$

  where the $Sigmoid$ activation function can be denoted as:

$$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{22}$$

In the output layer, the result from the linear part passed through a $Softmax$ activation function and became output:

$$\hat{\mathbf{Y}} = \text{Softmax}(\mathbf{Z}^L) \tag{23}$$

where the $Softmax$ activation function can be denoted as:

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \quad for \quad i = 1, ..., K \tag{24}$$

- **Loss Computation**

  The label used for training was created by one-hot encoding. And the MLP was trained with and edited $Cross\ Entropy$ loss which can be expressed as:

$$E(Y, \hat{Y}) = -Y. * \log(\hat{Y}) - (1-Y). * \log(1-\hat{Y}) \tag{25}$$

  After the $Softmax$ activation, the output of the network can be considered as the possibility of each label on the image and $Cross\ Entropy$ represents Maximum-Likelihood-Estimation (MLE) which benefits the training process of such probability model.

- **Backward Propagation**

  In this process, the gradient of the loss function w.r.t each layer was computed.

  The edited $Cross\ Entropy$ loss remains in the form of a vector with the same size as the output, which significantly simplified the process of computing the gradient. The gradient of the loss function w.r.t output can be derived as:

$$\frac{\partial E}{\partial \hat{Y}} = -Y./\hat{Y} + (1-Y)./(1-\hat{Y}) \tag{26}$$

  and the gradient of the $Softmax$ activation function can be derived as:

$$\frac{\partial \hat{Y}}{\partial \mathbf{Z}^L} = \text{Softmax}(\mathbf{Z}^L). * (1 - \text{Softmax}(\mathbf{Z}^L)) \tag{27}$$

  The gradient of weight and bias in each linear layer can be calculated as:

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{W}^l} &= \frac{\partial E}{\partial \mathbf{Z}^l} \frac{\partial Z^l}{\partial \mathbf{W}^l} \\ &= \frac{1}{m} \frac{\partial E}{\partial \mathbf{Z}^l} (\mathbf{A^{l-1}})^T \end{aligned} \tag{28}$$

$$\begin{aligned}\frac{\partial E}{\partial \mathbf{b}^l} &= \frac{\partial E}{\partial \mathbf{Z}^l}\frac{\partial Z^l}{\partial \mathbf{b}^l} \\ &= \frac{1}{m}\text{sum}(\frac{\partial E}{\partial \mathbf{Z}^l}) \quad \text{dim=2}\end{aligned} \tag{29}$$

where m denotes the input batch size.

In previous layers where activation functions were $Sigmoid$, the gradient can be derived as:

$$\frac{\partial \mathbf{A}^l}{\partial \mathbf{Z}^l} = \text{Sigmoid}(\mathbf{Z}^l).*(1-\text{Sigmoid}(\mathbf{Z}^l)) \tag{30}$$

- **Parameter Update**

  Weight and bias of each layer were adjusted according to the computed gradient. Meanwhile, in order to prevent over-fitting, L2-Regularization was applied in the parameter update process, which can be denoted as:

$$d\mathbf{W}^l = d\mathbf{W}^l + \frac{\theta}{m} * \mathbf{W}^l \tag{31}$$

  where θ is a hyperparameter determining the weight of L2-Regularization.

7.3 Sub-task 3

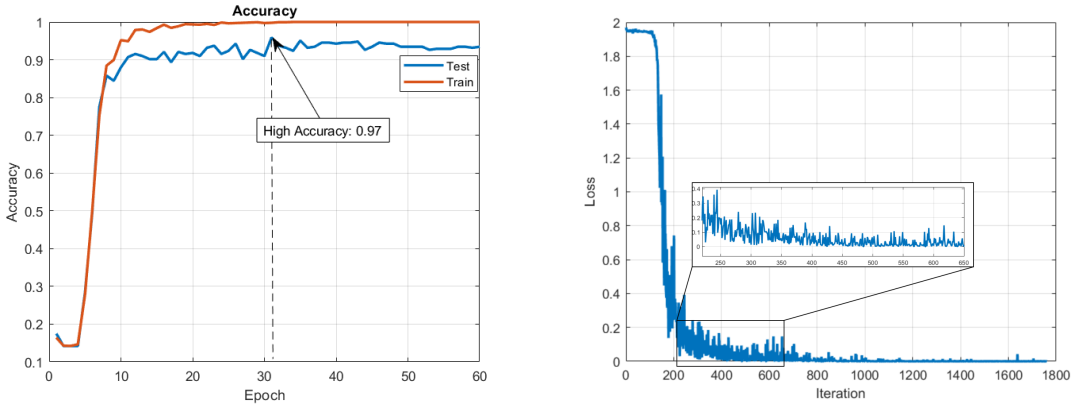
7.3.1 Experiment Results of CNN-based Model

In the implementation of our character recognition CNN, we incorporated a dropout rate of 0.2 to improve generalization and prevent over-fitting. The learning rate follows a linear annealing schedule, starting at a maximum of 0.1 and gradually decreasing to a minimum of 1e-5 over 80 epochs. We employed momentum with a value of 0.9 to enhance convergence speed. The entire training process was conducted on an Intel i7-10700H CPU, with a batch size of 64.

We finally achieved an accuracy **97%** on the test set, together with the loss **2.2484e-07**. The accuracy and loss curve of the whole training process are shown as Fig. 11.

To analyze failure patterns, we generated the corresponding confusion matrix as depicted in Fig. 12. In deep learning, a confusion matrix is a tabular representation illustrating the model's performance, detailing counts of true positive, true negative, false positive, and false negative predictions. This tool is invaluable for evaluating classification accuracy, revealing the model's ability to correctly classify instances and pinpoint areas of confusion. The matrix offers insights into precision, recall, and overall model effectiveness, facilitating the assessment of classification algorithms. Upon comparing some failure cases, we observed that images associated with true labels closely resembled those linked to predicted labels. Examples of these error pairs are presented in Fig. 13.

We also generated feature maps for each convolutional layer using misclassified samples. This exploration aimed to understand the features that the CNN model extracts and to assess the similarity of feature maps among the error samples. Fig. 14 depicts the feature maps corresponding to the misclassified sample *D*, the genuine *0*, and a correctly classified *D*.



(a) Accuracy curve (b) Loss curve

Figure 11: Training process for the best trial.

From Fig. 14, we observe that the feature maps of error pair samples exhibit striking similarities, while those of misclassified *D* and the correctly classified ones show significant differences. This discovery illuminated several limitations of CNN.

Firstly, the learning process of the convolutional kernel still possesses randomness even when SGD is employed to optimize the model parameters. This introduces issues where the trained CNN may fail to learn to extract certain ”special” features necessary to distinguish similar samples. However, this problem is also influenced by the scale and diversity of the dataset, revealing the second limitation.

Secondly, the performance of CNN is heavily contingent on the dataset's quality. Unlike how humans recognize new objects, which often requires only a few samples, or even none at all, a CNN model typically demands a substantial number of samples to identify a new input kind. In other words, the model performs well on unseen data only when exposed to a sufficiently large training set.

To delve deeper into how the dataset's scale affects CNN performance, we used a much larger MNIST [3] dataset, boasting 60,000 training samples and 10,000 testing samples—considerably larger than the provided dataset. For the input compatibility of MNIST with a 28x28 image size, we directly resized them to 124x124 and still used the same CNN framework. We increased the batch size to 128, considering the scale of MNIST while preserving other hyperparameters. The experimental result was extremely surprising, in which the accuracy on the test set can easily reach over **98%** as shown in Fig. 15, even when the model was only trained for **5** epochs. This inspired us that when using CNNs for image recognition, the quality of datasets such as diversity and size is critical to the accuracy of network predictions. A high-quality dataset is a prerequisite for high-accuracy identification. In order to investigate the effect of the dataset on our tasks, we adapted the data augmentation technique. More details about this are elaborated in section 8.1.2.

We evaluated the best model on test dataset using the supplied image *charact2.bmp*. Surprisingly, the results were unsatisfactory, as the model correctly identified only **3** characters out of **10**. After a comprehensive investigation, we determined that the misclassification resulted from differences in margin, size, and length-width ratio between segmented sub-

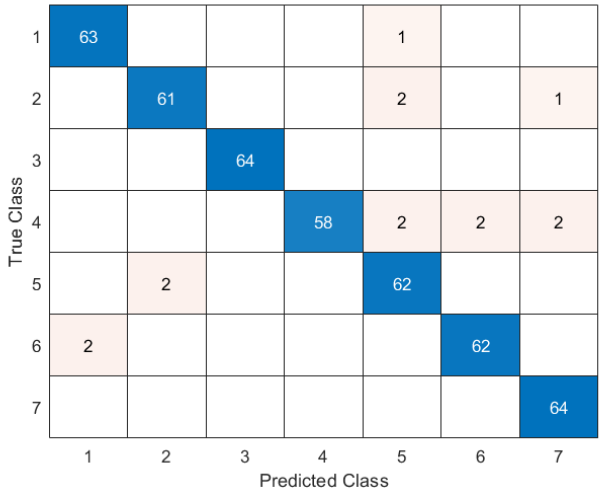


Figure 12: Confusion matrix for the best trial. Indexes 1 to 7 correspond to characters *0,4,7,8,A,D,H*, respectively.

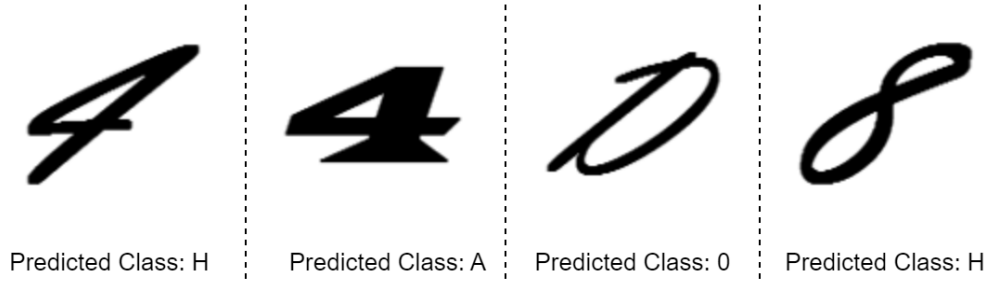


Figure 13: Misclassification samples.

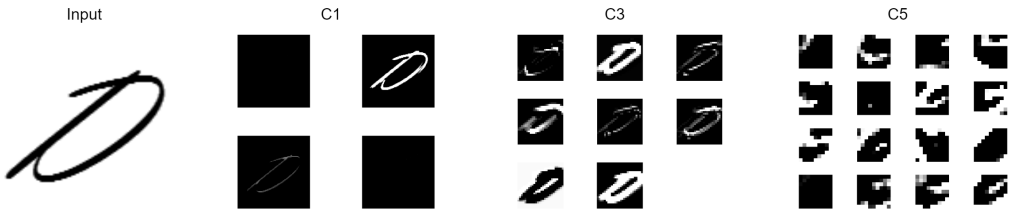
images of *charact2.bmp* and images in the dataset. Through the implementation of various strategies, we successfully improved the outcomes, enabling the model to accurately classify all characters. Further details on these strategies are elaborated in Section 8.1.

7.3.2 Experiment Results of Pure MLP-based Model

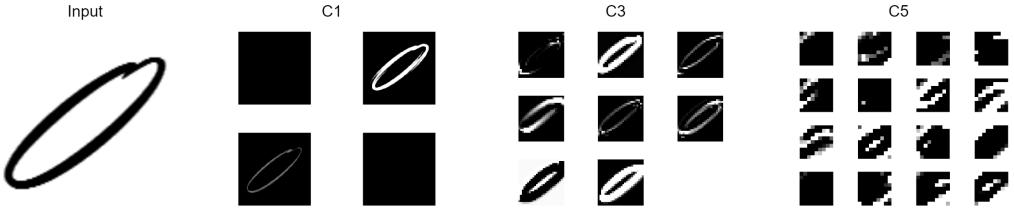
During the training process of the MLP, the learning rate was fixed to 0.02. In practice it was found practical that the whole dataset can be trained within one batch. In order to prevent overfitting, the training process was stopped when the accuracy on training data and test data are close.

The improvement of the network in first 350 epoches was not significant. After that there is a substaintial increase in the prediction accuracy in both training and test dataset. After finishing 1000 epoches of training, the MLP achieved accuracy of 88% on training data and 86% on test data.

To evaluate the model with more details, a confusion matrix was generated as depicted in Fig. 17. It is worth noticing that character *0,A,D* are more likely to be misclassified.



(a) Feature map of misclassified *D*



(b) Feature map of *0*

(c) Feature map of correctly classified *D*

Figure 14: Feature maps extracted by CNN.

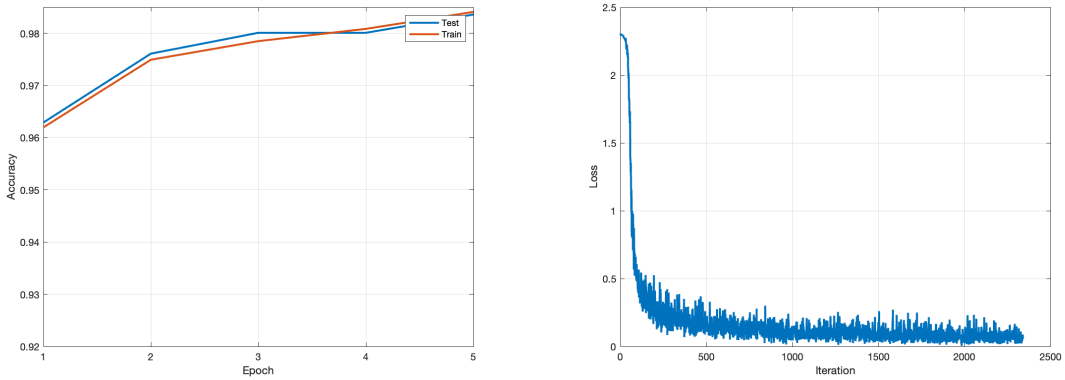
The MLP model was also evaluated by classifying each character in the second line of the given figure as shown in Fig. 1. At first the model could only correctly identified only 1 character out of 10. During our discussion, we considered the major reason for this problem as the fact that the images from the dataset contain balck character on white background while the images from the Fig. 1 contain white characters on black background. However, the model classified all the image with the reversed color into character *A*, which indicates there are factors except the reversed color. Our further discussion took margin, size, and length-width ratio between segmented sub-images into consideration. After adjusting the input image, the MLP model could successfully classify 6 characters out of 10.

7.3.3 Comparison

The CNN resulted in higher accuracy than MLP in both training and test dataset. The reasons are following

- Spatial Hierarchies and Local Receptive Fields:

  CNN is designed to recognize spatial hierarchies of features. They use convolutional layers with small filters and shared weights, which allows them to capture local patterns



(a) Accuracy curve (b) Loss curve

Figure 15: Training process on MNIST.

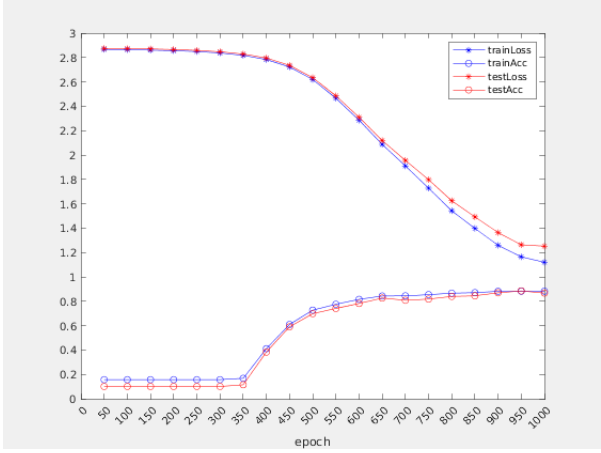


Figure 16: MLP Training Process and Result

and relationships in the input data. This is particularly important for images where pixels have spatial dependencies. MLP, on the other hand, treat each input feature as independent, which may not capture the spatial relationships in images effectively.

- Weight Sharing and Parameter Reduction:

  CNN uses weight sharing through convolutional kernels. This reduces the number of parameters compared to fully connected layers in MLPs. Weight sharing helps CNNs generalize better to variations in input data and reduces the risk of overfitting. MLP have a large number of parameters, and they may struggle with overfitting, especially when dealing with high-dimensional data like images.

- Pooling Layers:

  CNN includes pooling layers that downsample the spatial dimensions of the data, reducing computation and making the network more robust to variations in object positions

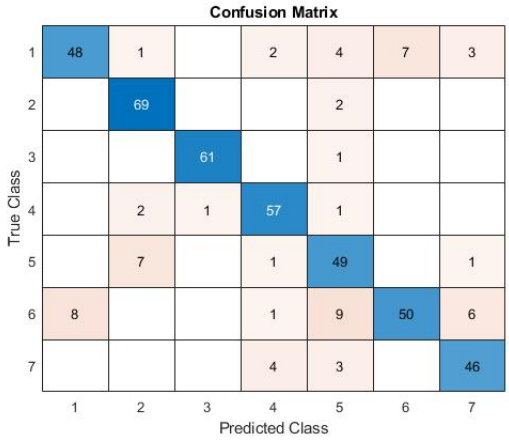


Figure 17: Confusion matrix of MLP. Indexes 1 to 7 correspond to characters *0,4,7,8,A,D,H*, respectively.

and sizes. MLP do not have pooling layers, and their architecture is not designed to capture hierarchical spatial information in the same way as CNNs.

Another significant difference between two types of networks is the amount of the parameter. As mentioned, the CNN can reduce its number of parameters by weights sharing through convolutional layers, which makes them more compact in compared with MLP designed for similar purposes. In the following part the amount of the parameter of the two network implemented in this task is calculated and compared. For simplify the calculation, only the number of the weights are taken into consideration.

The CNN used 5 5*5*4 convolutional kernels for the first 2 layers, 16 5*5*8 convolutional kernels were adopted in the next 2 layers, and the last 2 layers were formed with 32 5*5*16 convolutional kernels. The next part is the linear part, which includes a 144*100 matrix, a 100*50 matrix and a 50*7 matrix. Hence, the total amount of weights of the CNN is 36,250.

The structure of MLP is more straightforward. It was formed with 3 linear layers, which are essentially 3 matrices. The shape of each matrix is 16384*1024, 1024*256 and 256*7. The total amount of weights of the MLP is 17,087,232.

However, even given the fact that the parameters in MLP is hundreds of time of that in CNN, the training process of MLP is approximately 30 times faster than CNN. The reasons are following:

- Computational Complexity:

  CNN has a higher computational complexity due to the convolutional and pooling operations. Convolutional operations involve performing multiplication between the 3D convolutional kernel and each corresponding area of the input image which can hardly be paralleled and results in wasting more time on calling functions. On the contrary, each layer of MLP consists of only one matrix multiplication which is bigger but only one time of computation is required to pass the input through the layer. As a result, training a CNN requires more computational resources and time.

- Spatial Hierarchies and Feature Maps:

Although CNN contains fewer parameter than MLP, its expressiveness is higher than MLP, this is due to the fact that CNN is designed to capture spatial hierarchies of features in images through multiple layers of convolutions and the extracted features are stored in feature maps with high depth. One result is that even with fewer parameters, the total feature CNN needs to compute is higher than that of MLP. In this specific scenario, the total amount of feature of CNN can be calculated be adding the size of feature map in each layer, which is 124*124 at input, 4*120*120, 4*30*30, 8*26*26, 8*13*13, 16*9*9, 16*3*3 in convolutional layers and 100, 50 and 7 in linear layers. Hence the total feature the CNN calculates is 84,933. Moreover, there are overlaps between calculated area in each calculation, resulting in even more computational complexity. On the other hand, the total feature the MLP needs to compute is the size of each layer, which are 16384, 1024, 256 and 7 separately, which results in 17,671. This hierarchical feature learning requires more training iterations to converge compared to the simpler architecture of MLPs. MLPs, being fully connected, may converge more quickly because they lack the spatial hierarchy learning characteristic of CNN.

- Pooling Layers:

  CNN includes pooling layers to downsample spatial dimensions. While pooling helps in reducing the spatial resolution and computation, it adds an additional step to the forward and backward passes during training, contributing to increased training time.

- Backward Propagation

  Like the forward propagation, the gradient of the loss function w.r.t each layer of the MLP can be easily calculated with matrix multiplication. However, to calculate the gradient of the loss function w.r.t each convolutional kernels of the CNN requires more complicated calculation since the gradient needs to be calculated from different channels of feature map, which significantly reduces the speed of computation.

In conclusion, the main characteristic of CNN is that it can sense the surrounding area of each pixel, which improves the understanding of the whole image. Even though CNN has fewer parameters, through computation process with higher complexity, CNN is more expressive than MLP in image classification task.

8 Task 8

Considering the initial poor performance of letter recognition we mentioned at section 7.3.1, we first analyzed the failure patterns. Subsequently, we attempted image resizing and padding, and tuning hyper-parameters according to the instructions. The CNN model ultimately achieved flawless recognition of all the letters. Additionally, we conducted thorough experiments to assess the impact of network architecture.

8.1 Data Pre-processing

8.1.1 Resizing and Padding

Observing the low success in recognition ratio, we initially contrasted the split sub-images with the dataset images illustrated in Fig. 18. Notably, significant disparities exist between the two image groups. The most prominent dissimilarity lies in the Character color—specifically, the characters in the dataset are black with a black background, whereas the split characters from the provided image exhibit the inverse. Conversely, the character proportion within the image differs; in other words, the dataset images have larger margins towards the boundaries.

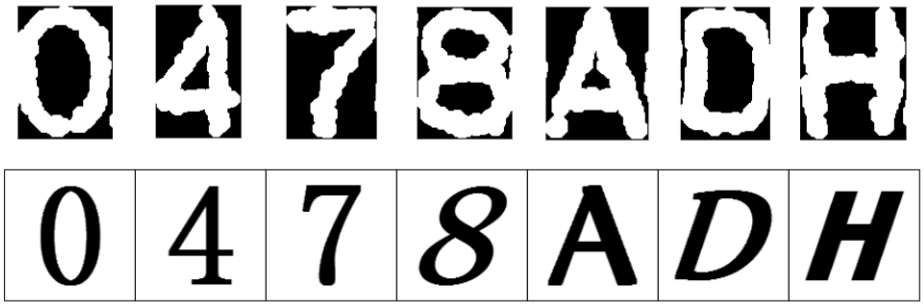


Figure 18: Comparison of images from the split results and the given dataset.

To address the aforementioned issues, we performed data pre-processing on the segmented images prior to inputting them into the CNN model. Specifically, we initially inverted the pixel values of the image. Subsequently, a MATLAB function was employed to achieve character scaling, maintaining the original image dimensions and automatically padding the surplus pixels with zeros. The preprocessing outcomes are depicted in Fig. 19.

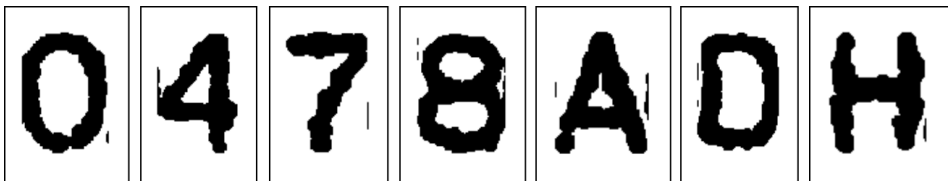


Figure 19: Pre-processed split images.

After resizing and padding the input, it is then adjusted to 124x124 for compatibility with the CNN input layer. The model successfully classified all letters except *8*, which was mistakenly

identified as *H*. To address this issue, we examined the pre-processed *8*, comparing it with dataset images and noting a slightly higher length-to-width ratio in the given image than in the dataset. In response, we unevenly scaled the letters to create a flatter appearance, setting the scale ratio to $(0.6, 0.8)$ for height and width, respectively. As a result, the model accurately recognized all letters.

8.1.2 Random Transformation

In section 7.3.1, we emphasize the importance of diversity and scale in training a CNN model. However, the dataset provided is limited to just 1778 samples. To enhance the dataset's quality, we employ the widely adopted data augmentation technique, applying random transformations to increase diversity and expand the data samples. Our approach involves introducing random translation, rotation, and scaling to the initial dataset, considering the characteristics of the letters; for example, many English letters exhibit an inclination of 15 to 30 degrees to the right.

To comprehensively explore the impacts of various random transformations, we conduct ablation experiments, investigating different combinations of these transformations. In our implementation, we apply random transformations to 20% of the training set in each trial to augment the training samples while keeping other hyperparameters constant. The random translation displaces a sample by up to 10%; the random rotation rotates it by a maximum of 25 degrees, and an image is randomly scaled by a factor between 0.8 and 1.2. Some transformation samples are depicted in Fig. 20.

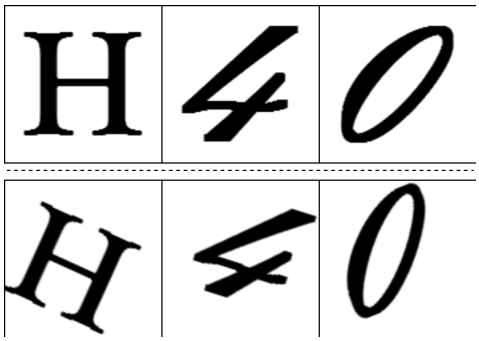


Figure 20: Illustration of random transformation. Original images (top), randomly transformed images (bottom)

The results are presented in Table **??**, indicating that the CNN model benefits more when applying only one type of random transformation. This is because the inclined characters in the dataset tend to confuse the CNN, leading it to "think" that an inclined character must represent a specific letter. Consequently, applying random rotation to the input samples counteracts this directional confusion pattern.

When applying two types of random transformations, the combination of rotation and scale demonstrates greater effectiveness. This is supported by our observations detailed in Section 8.1.1, where the scaling of characters influences the margin to the boundary for a given letter. Therefore, random scaling contributes significantly to enhancing data diversity.

Table 4: Impact of different random transformations.

Index	Applied Transformation	Best Accuracy (%)
1	None	88.25
2	Only translation	90.71
3	Only rotation	**90.98**
4	Only scale	90.16
5	Rotation and translation	91.26
6	Rotation and scale	**93.44**
7	Translation and scale	92.62
8	All	**93.72**

Finally, applying all random transformations resulted in the highest classification accuracy in the ablation studies, confirming the effectiveness of our applied data augmentation technique.

8.2 Hyper-parameters Tuning

8.2.1 Learning Rate

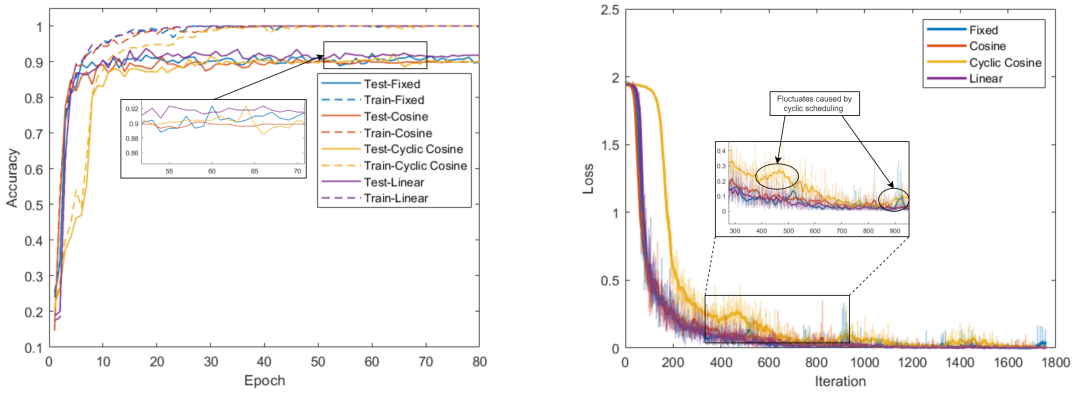
- **Learning Rate Scheduling Scheme**

  Various learning rate scheduling schemes exist, including linear annealing, cosine annealing, and cyclic cosine annealing [4]. These have proven more effective than a fixed learning rate. In recent years, researchers have introduced adaptive learning rate scheduling approaches such as Adam [5] and AdamW [6]—widely adopted in modern neural network training, significantly advancing deep learning. In our experiments, we compared contributions to the learning process from constant learning rate, linear annealing, cosine annealing, and cyclic cosine annealing. Adaptive learning rate schemes were not considered, as our CNN model is simple and doesn't require these methods. Comparison results are depicted in Fig. 21.

  As illustrated in the figure, linear annealing demonstrates superior performance, enabling swift convergence of the loss and attaining the minimum value compared to alternative schemes. Unexpectedly, the cyclic cosine scheme shows the least favorable performance in terms of both convergence speed and final value. Moreover, it introduces extra oscillations during training. We attribute this to the size and simplicity of our CNN model, rendering the global optimum easily discoverable. While implementing such a learning rate scheduling scheme accelerates the identification of the global optimum, it also gives rise to increased fluctuations.

  This experiment suggests that not all learning rate scheduling schemes that "appear" more efficient can benefit the training. The selection of the optimal learning rate scheduler should take into account the specific CNN model structure, dataset features, and the given task. Sometimes, a simple scheduling scheme yields better results.

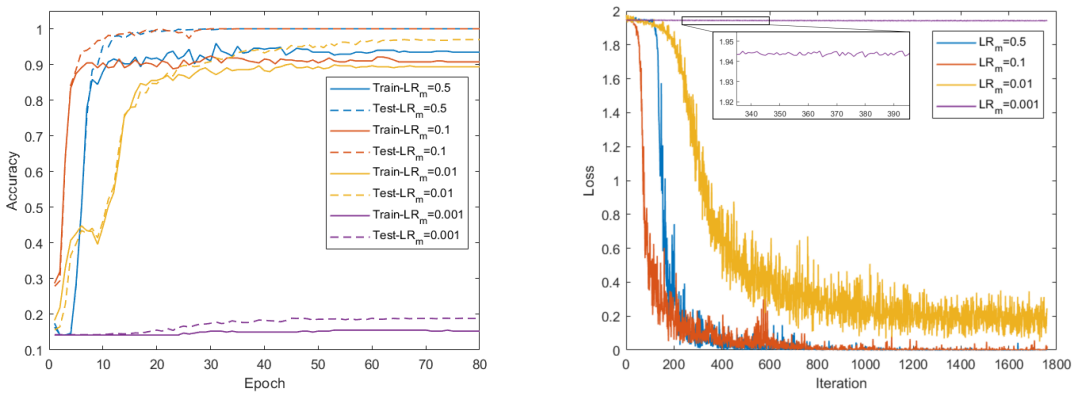
- **Optimizing Learning Rate Magnitude**



(a) Accuracy curve (b) Loss curve

Figure 21: Training process comparison at different learning scheduling schemes. The same color but thicker line represents the smoothed result of this set of data.

Employing a cosine annealing schedule with a fixed minimum magnitude, we examined the impact of various maximum learning rate magnitudes: 0.5, 0.1, 0.01, and 0.001, respectively. The results are illustrated in Fig. 22.



(a) Accuracy curve (b) Loss curve

Figure 22: Training process comparison at different learning rate.

The result suggests the fact that the larger learning rate causes quicker convergence but, conversely, brings about increased volatility during the optimization process. While the rapid reduction in the loss function is evident in the initial stages of training, the larger learning rate magnitudes often lead to erratic behavior characterized by frequent oscillations and fluctuations. These erratic patterns introduce challenges in maintaining a steady descent towards the optimal solution, and the training process becomes more susceptible to diverging or overshooting.

On the contrary, smaller learning rate magnitudes exhibit a more stable convergence pattern. Despite the initial trade-off of a slower convergence rate, the optimization tra-

jectory is characterized by smoother progress with fewer oscillations. This increased stability allows for a more reliable and predictable descent through the loss landscape. However, it is important to note that very small learning rates may also introduce challenges related to getting stuck in local minima as shown in Fig. 22, which exhibits the trapping of the training process, i.e., loss and accuracy don't get significantly improved until the end of training. The cautious exploration of the parameter space, while minimizing overshooting, might hinder the model's ability to escape shallow local minima.

In summary, our experimentation with cosine annealing and a minimum magnitude as the learning rate scheduling scheme highlights the nuanced effects of different learning rate magnitudes. Larger learning rates expedite convergence but introduce instability, while smaller learning rates foster stability at the cost of a potentially slower convergence and the risk of getting stuck in local minima. These insights contribute to the ongoing refinement of learning rate strategies, providing a deeper understanding of their impact on training dynamics.

8.2.2 Batch Size

In our exploration of the influence of different batch sizes on the performance of our CNN-based character recognition model, we conducted a series of experiments, aiming to understand the interplay between batch size, model convergence, and accuracy on the test set. Batch size, representing the number of training samples processed in each iteration, is a critical hyperparameter in deep learning, influencing both computational efficiency and the dynamics of model training.

Our experiments involved varying the batch size across different configurations: 16, 32, 64, and 128. The results, as summarized in Tab. 5 and Fig. 23, provide valuable insights into the nuanced relationship between batch size and model performance.

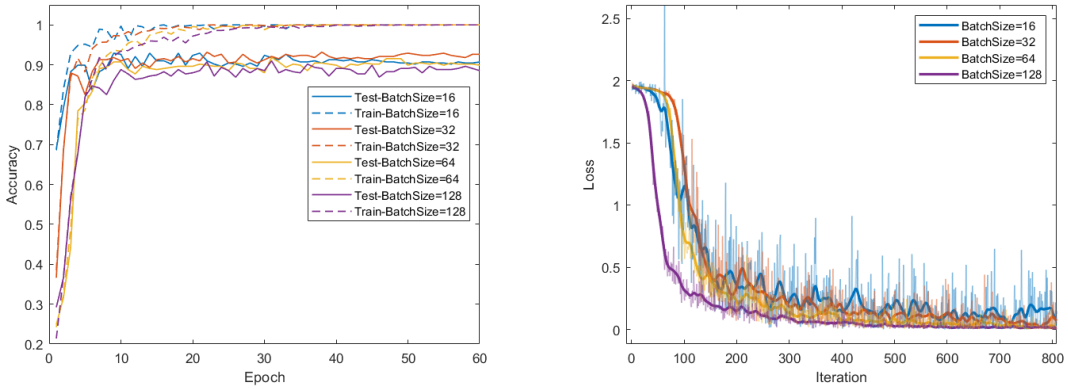
Table 5: Impact of Batch Size on Test Accuracy

Index	Batch Size	Best Accuracy (%)
1	16	92.62
2	32	**94.81**
3	64	91.26
4	128	90.71

When employing a smaller batch size of 16, the model achieved a test accuracy of 92.62%. This setting demonstrated a relatively stable training process, characterized by smooth and consistent convergence of accuracy and loss curves during training.

Increasing the batch size to 32 yielded an improved test accuracy of 94.81%. This adjustment resulted in a more rapid convergence, suggesting the potential benefits of a larger batch size in terms of accelerating the training process.

However, when utilizing a batch size of 64, the test accuracy decreased to 91.26%, accompanied by a noticeable oscillation in both accuracy and loss curves. This oscillatory behavior suggests that, while larger batch sizes can contribute to faster convergence, an excessively large batch size may introduce instability during the training process.



(a) Accuracy curve (b) Loss curve

Figure 23: Training process comparison at different batch sizes. The same color but thicker line represents the smoothed result of this set of data.

Further experimentation with a batch size of 128 led to a test accuracy of 90.71%. Interestingly, this setting exhibited a slower convergence rate compared to smaller batch sizes, indicating a potential trade-off between training speed and convergence stability.

In summary, our experiments underscore the importance of carefully selecting the batch size for training our CNN model. The choice involves a delicate balance, considering computational efficiency, convergence speed, and stability during training. This investigation contributes to a nuanced understanding of the impact of batch size on the training dynamics of character recognition models.

8.2.3 Epoch Number

Exploring the impact of different epochs on training outcomes reveals a nuanced relationship. Table 6 illustrates our model's evolving accuracy at various epochs. Initially, a noticeable improvement is observed, with an 88.25% accuracy at 10 epochs, signifying early learning. However, as epochs progress, accuracy shows diminishing marginal returns. Despite ascending up to 80 epochs, the incremental gain lessens, suggesting diminishing returns.

This phenomenon highlights the concept of diminishing marginal benefits in character recognition. Continuously increasing epochs may not yield substantial performance gains and could lead to overfitting. Therefore, selecting the optimal number of epochs is crucial to strike a balance. The model's generalization ability may be compromised with overfitting, emphasizing careful consideration in epoch selection. This nuanced exploration emphasizes the iterative nature of CNN-based character recognition, where finding the optimal epoch count is paramount. Beyond a certain point, returns may diminish, and the risk of overfitting may rise.

Table 6: Best accuracy at different epochs.

Index	Epoch Number	Best Accuracy (%)
1	10	88.25
2	20	91.53
3	40	91.80
4	60	92.26
5	80	**92.38**

8.3 Discussion and Conclusion

In this section we focused on improving the performance of CNN in classifying character from the segmented sub-image from the chip. We tried data augmentation and hyper-parameters tuning separately.

In the data augmentation part, resizing, transformation and rotation were adopted. The result showed that when applying only one data augmentation method, the results of training were similar but the one applied rotation ranked top. When applying two data augmentation methods, the difference between of the training results gradually became clear. The one applied rotation and scaling ranked highest in accuracy. The combination of all three data augmentation ranked the highest above all training experiments. One reason of such improvement is that the network was exposed to a broader range of variations in the input data, which significantly improve the generalization of the network.

In the hyper-parameters tuning part, learning rate, batch size and epoch number were taken into consideration. The result revealed that the loss with near annealing demonstrated superior performance during the training process. The magnitude of learning rate also played an important role in training process. Higher learning led to faster convergence but the subsequent learning may not be stable, while lower earning rate resulted in slower but more stable learning process. However, it is worth mentioning that very small learning rates may also introduce challenges related to getting stuck in local minima, which will impede further learning of the network. When considering batch size, the batch size of 32 resulted in highest accuracy of testing. A possible reason is that for smaller batch size, the gradient may be easily influenced by specific data and deviate from the optimal direction, while bigger batch size may result in unstable training process. The number of epoch of the whole training directly influence the result of the network parameters. Generally, the performance of the network was improved by increasing epoch number. However, as the epoch number increased, the training gain began to decrease gradually and a too large number of epoch may result in the overfitting of the network.

References

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

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[4] I. Loshchilov and F. Hutter, “Sgdr: Stochastic gradient descent with warm restarts,” *Learning*, vol. 10, p. 3.

[5] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[6] Z. Zhuang, M. Liu, A. Cutkosky, and F. Orabona, “Understanding adamw through proximal methods and scale-freeness,” *Transactions on Machine Learning Research*, 2022.