Department of Mechanical Engineering

ME5411 Computer Vision and AI

Project Report

Group 6

Name : Madkaikar Atharva Atul **ID Number :** A0268376M

Name : Thong Chee Wai Benjamin **ID Number :** A0268544U

Name : Tiancheng Zhang **ID Number :** A0268347R

Contents

1 Introduction

The field of computer vision, image processing, and artificial intelligence has gained significant attention in recent years, owing to its potential applications in various domains. In this report, we present the results of a series of tasks performed on a given image using different techniques and algorithms. The tasks included image smoothing, sub-image creation, thresholding, outlining characters, segmenting images, and finally, classifying characters using both CNN-based and non-CNN-based (SVM) approaches. The aim of this report is to compare the effectiveness and efficiency of these approaches and to investigate the sensitivity of the methods to changes in preprocessing and hyperparameters. We present the results obtained from each task, provide a comparison of the two classification approaches, and offer an explanation for any differences in the outcomes. Overall, this report presents a comprehensive analysis of different computer vision and artificial intelligence techniques applied to a real-world problem.

2 Part I: Image Preprocessing

2.1 Task 1: Display the Source Image

At the beginning of the experiment, we loaded the noisy source image, as shown in Fig 1.

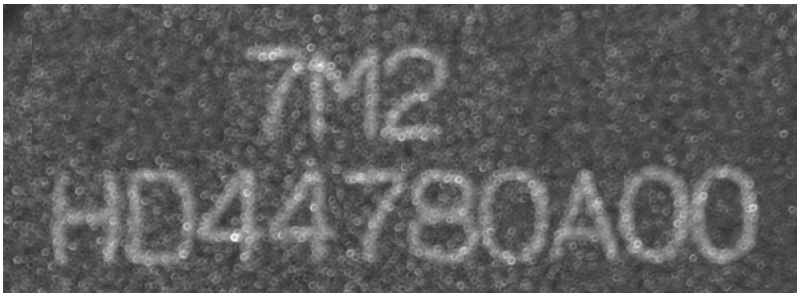

Figure 1: The source image

2.2 Task 2: Averaging Mask

Averaging mask is one of the common operations used for blurring and smoothing a noisy image. It works by replacing each pixel in the image with the average value of all its neighboring pixels within a predefined kernel. In this project, we implemented two methods to construct an averaging mask for image processing: the convolution-based method and the integral map-based method. The convolution-based method is straightforward, while the integral map-based method is more computationally efficient. While both methods achieve the same goal of averaging the neighboring pixels in an image, the integral map-based method is preferred for larger images or where larger kernels will be used due to its reduced computational complexity.

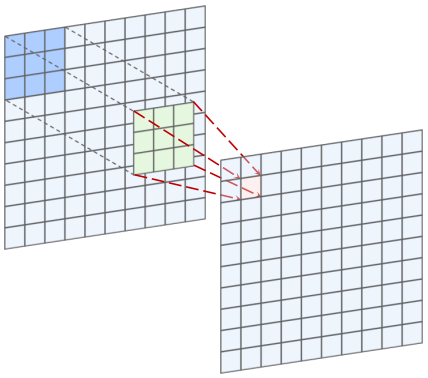

Figure 2: convolution-based averaging mask

Convolution-based Method

The convolution-based method implemented simply convolve the image with an $n \times n$ kernel with all its elements being $\frac{1}{n^2}$, as illustrated in Fig 2. For an $x \times y$ image, the computational time required by this method with $n \times n$ kernel is $\Theta(xyn^2)$

Integral Map-Based Method

The integral map-based method for achieving an averaging mask is much more computationally efficient than the convolution-based method. As opposed to the convolution-based method, this method does not rely on any convolutional operations but instead relies on calculating the integral map of a given image. The integral map used in this method is simply the cumulative sum of all pixels in the image, as shown in Fig 3. Afterwards, the summation of the values of surrounding pixels can be calculated within a very small number of operations. Taking the example image in Fig 3 as an example, if we want to calculate the sum (denoted as S) of pixels within the 3×3 region around the point located at $(3, 3)$, then S can be calculated as:

$$S = (S1 + S2 + S3 + S4) + S1 - (S1 + S3) - (S1 + S2) \tag{1}$$

in which $S1$, $S2$, $S3$ and $S4$ refer to sum of pixels in the regions denoted by different colors, respectively.

The advantage of this method is that its computational time is invariant as the kernel size n is changed, and this feature will be more prominent as the kernel sizes increases. Such efficiency is mainly due to the reason that the integral map can be pre-computed once for an entire image, and then used to compute the sum of pixel values within any rectangular region of the image, regardless of its size or position, with just a few simple operations. This allows for very fast computation of averages or other types of smoothing operations across large images.

The effect of the averaging mask

After designing the averaging mask, we tried to denoise the source image with averaging masks with different mask size, the result obtained is shown in Fig 4. Fig 4 shows that the more noise will be eliminated as the mask size increases, but at the same time the image itself becomes more blurry and might lose some important features.

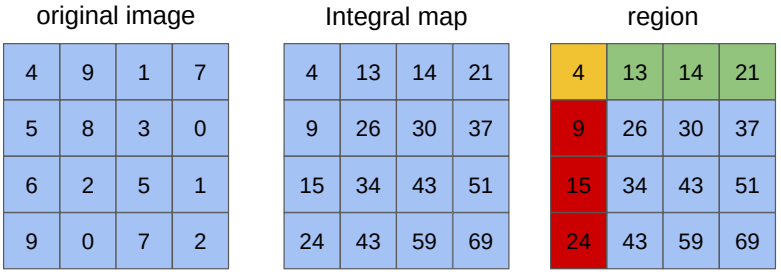

Figure 3: Integral map-based averaging mask

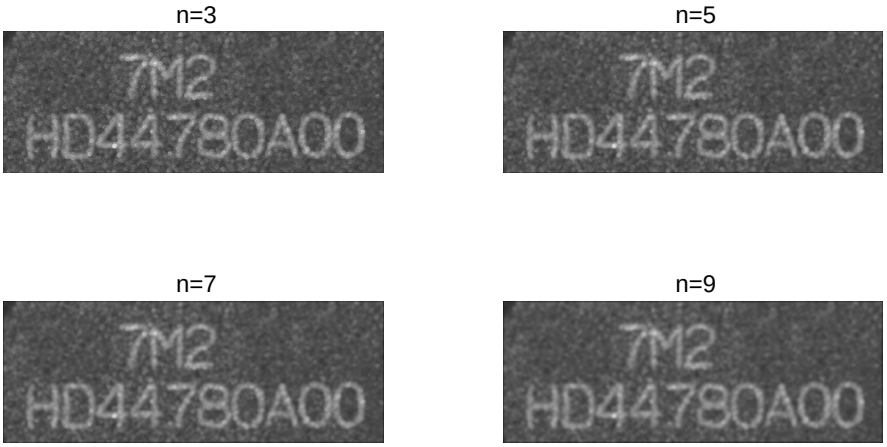

Figure 4: The effect of different size of the averaging mask

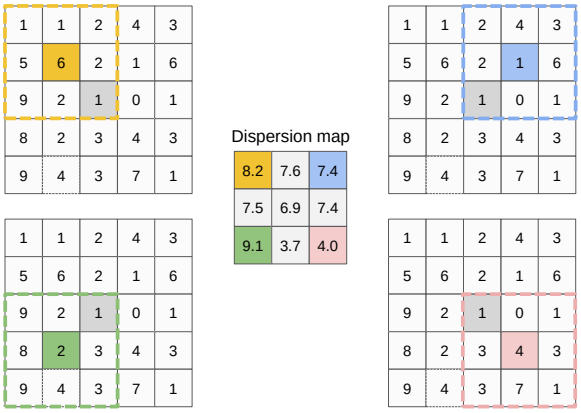

Figure 5: Illustration of the rotating mask method, where the surrounding four figures denote four different masks applied to the image, and the central figure shows the dispersion map obtained from these masks

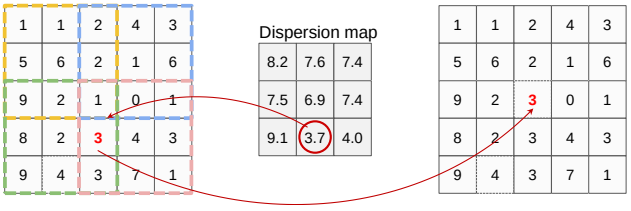

Figure 6: Illustration of how the rotating mask will be applied to the image. The pixel with minimum dispersion will be selected to replace the original pixel

2.3 Task 2: Rotating Mask

Rotating mask is another method used to smooth noisy images. Unlike the averaging mask, a rotating mask replaces each pixel with one of its neighboring pixels instead of taking the average of all neighboring pixels. The pixel to be replaced is that has minimum surrounding dispersion (variance) values. This process is displayed in Fig 5, where 4 different 3×3 rotating mask is applied to the central pixel (denoted as gray color) of a sample 5×5 image and their corresponding dispersion maps are also illustrated.

After obtaining the dispersion map, the pixel corresponding with the minimum dispersion within the map will be selected to replace the original pixel, as shown in Fig 6.

Although rotating masks can usually achieve (satisfying) results, the conventional methods can be very slow in speed, since the calculation of dispersion is time-consuming. To alleviate this problem, we also invented a way to support fast calculation to avoid calculating the dispersion pixel-by-pixel. We present our algorithm for fast computation of dispersion map in Fig 15. In our implementation, the previously designed integral map-based averaging mask is also used in rotating mask.

We also tested with the rotating mask with different mask sizes, as shown in Fig 8. The result shows that larger mask size could make the rotating mask more effective in terms of denoising images without blurring the image.

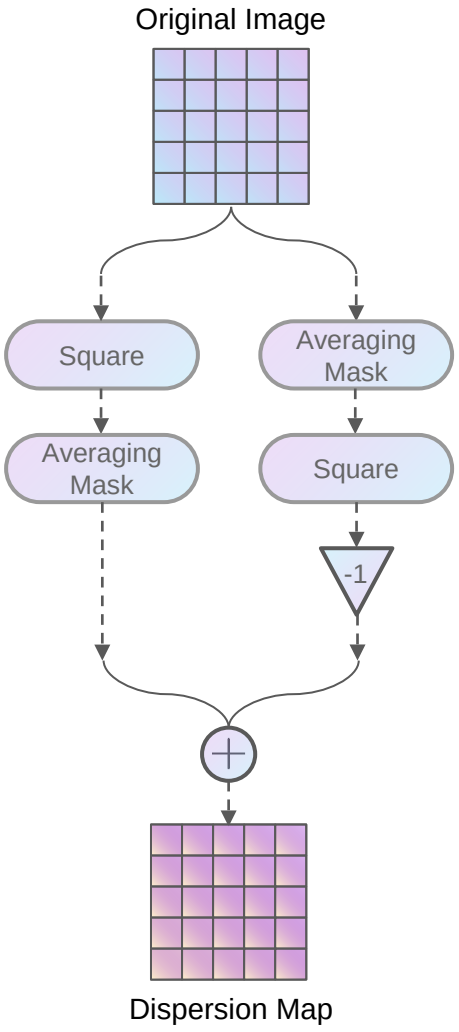

Figure 7: Flow chart of computing dispersion map

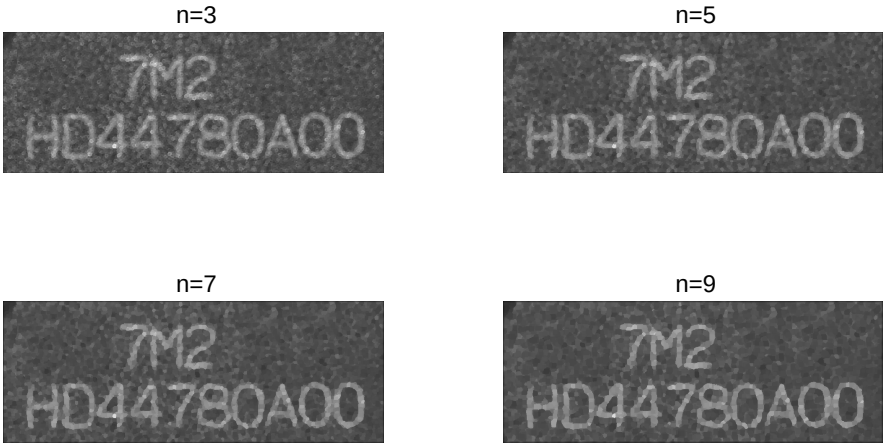

Figure 8: The effect of different size of the rotating mask

2.4 Image Denoising

After testing both the averaging mask and rotating mask implemented by ourselves, we tried and applied different combinations of both masks to the source image, and the denoised image is illustrated in Fig 9.

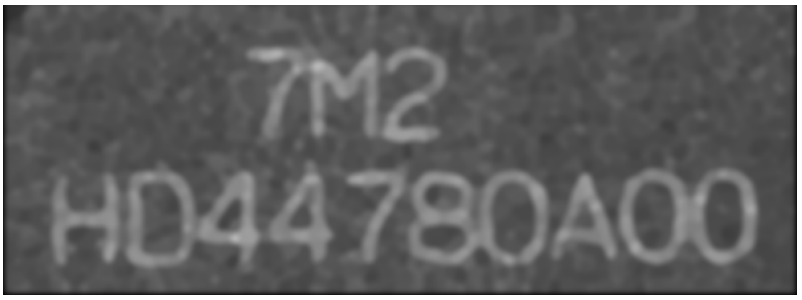

Figure 9: The image after applying both the averaging mask and rotating mask

2.5 Task 3: Subimage

The problem statement given in Task 3 was to extract a sub-image with the line "HD44780A00" from the original image. The original image is a $367 \times 990 \times 3$ colour image of two rows of text of which we are expected to extract the second row from. The image has a somewhat uniformly noisy black background with white foreground text. We observed that the rows with text would have a higher count of bright pixels as opposed to background and used that as a basis for the algorithm. A high level flowchart of our algorithm is shown in Fig 10.

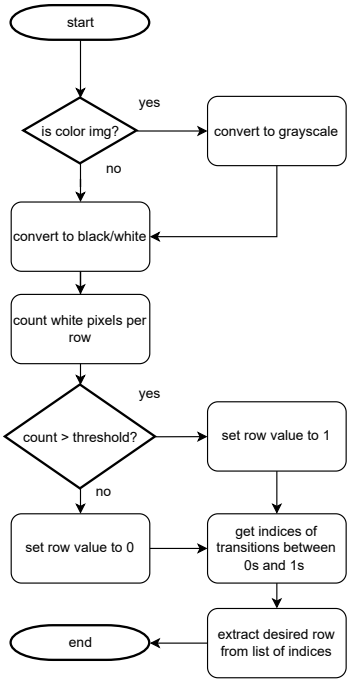

Figure 10: High level flowchart for extracting row from original image

Fig 11 shows the result of counting white pixels of each row in the image. From the count a desired threshold value can be chosen to separate the rows of text from the image. From the extracted text row we conclude that this approach is robust and can be extended to handle more complex scenarios. For example, extraneous rows of strong horizontal noise can be filtered out by further considering only rows with a configurable minimum height matching the font used. Alternatively, if the estimated vertical position of the text is known prior, the region containing that position can be selected.

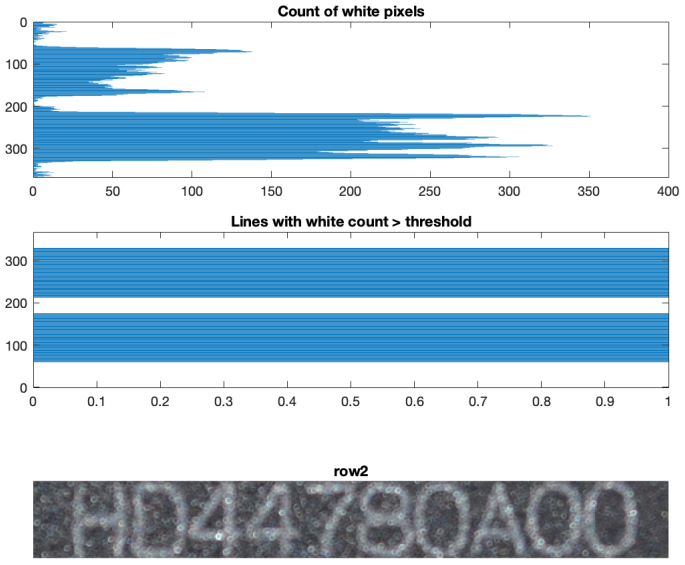

Figure 11: Extracting 2nd row from original image

2.6 Task 4: Binary Image

To convert the original grey image to binary, we simply set a threshold t value and all the pixels with value larger than t to be 255 while others to be 0. For our project, we found out that setting the threshold as 127 can produce relatively optimal result, so we used this value throughout the whole project.

The binary image is displayed in Fig 12, which shows that the digits can be perfectly separated from the background.

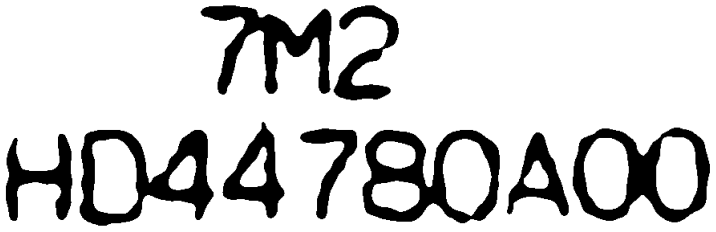

Figure 12: The binary image

2.7 Task 5: Determine Image Outline

Given the binary image from Task 4, the outline of the characters were extracted by applying Sobel filters to the image followed by an optional edge linking. The flowchart is shown in Fig 13. Sobel filters are a type of linear filter that calculate the gradient magnitude of the image intensity at each pixel. It consists of two kernels, one for each direction. The gradient magnitude of the pixel is then approximated by the sum of each direction's absolute value. Edge linking helps to connect boundaries between regions for improved edge continuity.

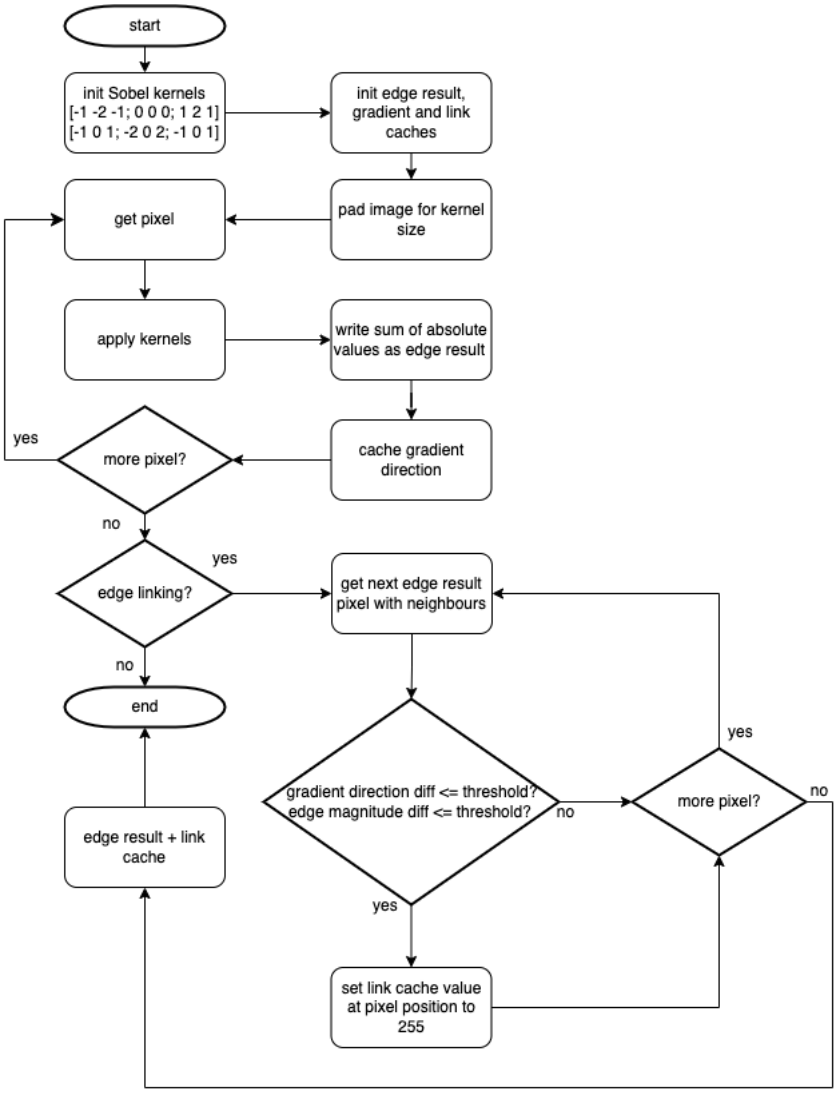

Figure 13: High level flowchart for extracting edge from binary image

The resultant extracted image along with edge linking is shown in Fig 14. The edges are noticeably stronger with edge linking. Due to the source image being binary, coupled with the gradient magnitude approximation, the unique gradient magnitude values are 0, 510, 1020 and 1530. Through our tests, changing the magnitude threshold between these values or varying the edge-linking-angle value of above 10 had no noticeable effect. We conclude that this is due to the source being a binary image and these parameters would have been noticeable on images with more varying pixel intensities.

Figure 14: Top: Sobel operator without edge linking. Bottom: With edge linking.

2.8 Task 6: Image Segmentation

The purpose of doing image segmentation is to assign the same label to the foreground pixels that connect with each other. In this case, since we are dealing with binary images, we can leverage the disjoint set data structure to segment the image. At the initialization stage of our image segmentation algorithm, all the pixels will be separated to different sets. Then the algorithm will search over through all the foreground pixels and connect them with all the neighboring foreground pixels. At last, all the foreground pixels within the same region will be included to the same set by taking unions of all interconnected pixels.

To speed up the process of taking union, we used the weighted quick union algorithm suggest by [1], whose time complexity is $O(\log N)$

Figure 15: Illustration of the segmentation algorithm: The left figure shows a sample image, in which foreground pixels are denoted as ”1” and background pixels are denoted as ”0”, and this is an unsegmented image with all its pixels labelled with different colors. The right figure shows the segmented image in which the connected foreground pixels are labelled with the same color

The segmented result of the whole image is displayed in Fig 12, where pixels belong to different regions are labelled with different colors (randomly assigned).

Since the primary goal of this project is to recognize the characters on the second line.

Figure 16: The segmented image

We also processed and segmented the seperated image obtained in Task 3, the result is shown in Fig 17.

Figure 17: The second line of the segmented image

However, due to the close proximity between some characters, it can be seen the character pairs “D4”, “80” and “00” at the second line appeared as contiguous segments. Therefore a further preprocessing step is required before it is suitable as input to the classifier, which further expects 128×128 sized images with black foreground on a white background. Fig 18 outlines this preprocessing step. The columns chosen to split contiguous pairs is shown in Fig 19 and the resultant individual characters ready for classifier is shown in Fig 20.

3 Part II

3.1 Subtask 1: CNN based character recognition model

3.1.1 Introduction

The aim of this task is to develop a Convolutional Neural Net (CNN) in MATLAB to classify each character segmented in part 1 of the BMP image of a label on a microchip. CNNs have been successfully applied to character recognition tasks, achieving high accuracy rates in recognizing both printed and handwritten characters. CNNs learn and identify relevant features from images automatically, without requiring manual feature engineering. Moreover, CNNs can be trained end-to-end, which means that the input image can be directly processed and classified without the need for extra processing steps. This enhances the efficiency and speed of character classification process.

3.1.2 Dataset

The dataset comprised a total of 1778 binary images of seven different alphanumeric characters: 0, 4, 7, 8, A, D, and H, as shown in Fig 21. All these images were of size

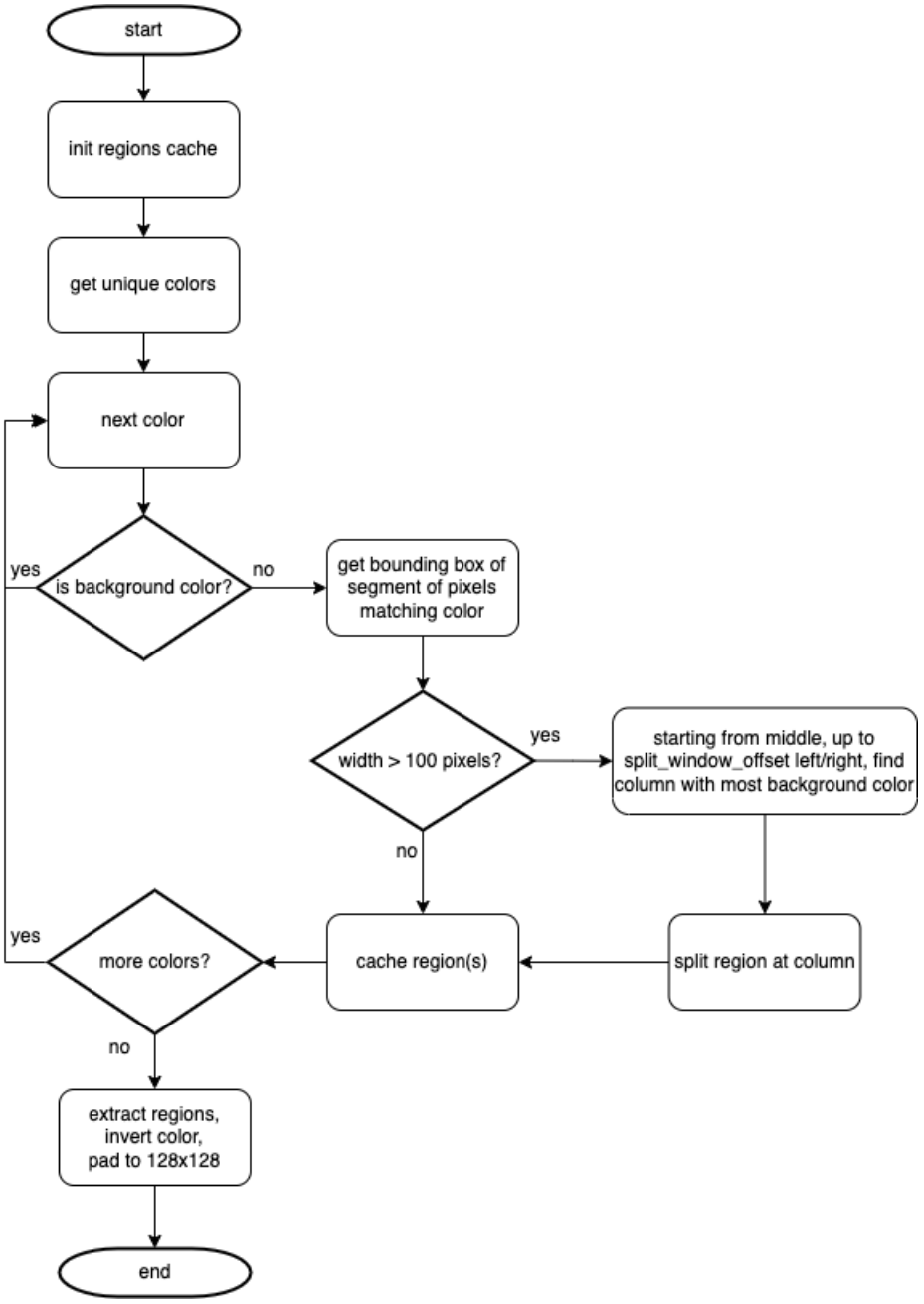

Figure 18: Flow chart of splitting contiguous pairs with preprocessing for classifier

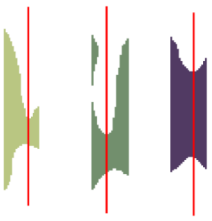

Figure 19: The manually segmented digits with red lines indicating the chosen split between character pairs

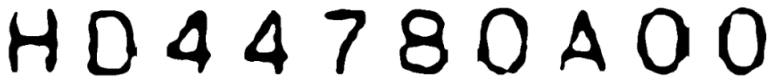

Figure 20: Resultant preprocessed characters ready for use as input to classifier

128×128 pixels. The dataset was randomly split into two parts for training and validation. The training set consisted of 75% of the data, while the remaining 25% was reserved for validation purposes. This random division of the dataset serves to minimize potential biases that may arise if the data is not selected randomly. Furthermore, this approach reduces the risk of overfitting and promotes model accuracy by providing more robust and reliable metrics for testing. The dataset was used as is, without any pre-processing or data augmentation applied.

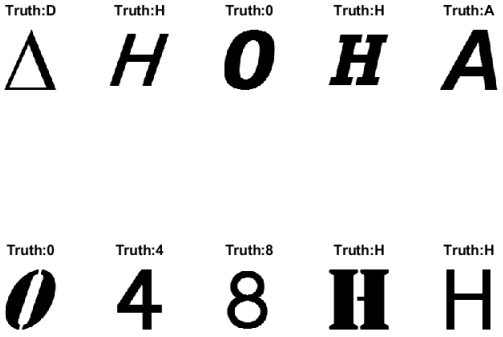

Figure 21: Sample of the Dataset

3.1.3 Model Architecture

The CNN architecture used comprised of the following layers, as visualized in Fig 22:

- Input layer: This layer received binary images of size 128×128 pixels.
- Convolutional layers: Our network consists of seven convolutional layers. Every convolution operation was followed by a batch normalization step to minimize internal covariate shift and keep the outputs of each convolution operation within a minimum range for better convergence and performance. Each convolution layer uses a ReLU activation function. Pooling layers have been included for some convolutions while excluded for others.
    1. Layer 1: 8 kernels of size 3×3 with a stride of 1 and padding 1 were used for the convolution operation to give an output volume of size $128 \times 128 \times 8$. Keeping the number of kernels low for the initial layers helps the model to not memorize irrelevant patterns or noise. Finally, a max pooling layer of size 2×2 with a stride of 2 was used to reduce spatial dimensionality of the input volume. Pooling layers also help get rid of the noise in the images. The resulting output volume had a size of $64 \times 64 \times 8$.
    2. Layer 2: 16 kernels of size 3×3 with a stride of 1 and padding 1 were used for the convolution operation to give an output volume of size $64 \times 64 \times 16$. A max

pooling layer of size 2×2 with a stride of 2 was used to give an output feature map of size $32 \times 32 \times 16$.

3. Layer 3: 32 kernels of size 3×3 with a stride of 1 and padding 1 were used for the convolution operation to give an output volume of size $32 \times 32 \times 32$. As you can observe, the number of kernels was increased gradually as with the increase in the depth of the network, the filters are able to learn more sophisticated and specialized features, making it suitable to use a higher number of filters. Pooling layer was skipped to prevent loss of spatial information. This also, allows us to build deeper networks by preserving the spatial dimensionality of the layers.
4. Layer 4: 64 kernels of size 3×3 with a stride of 1 and padding 1 were used for the convolution operation to give an output volume of size $32 \times 32 \times 64$. An average pooling layer of size 2×2 with a stride of 2 was used instead of max pooling layer. Since deeper layers hold more valuable high-level feature data, using average pooling minimizes the loss of important spatial information. The output feature map for this layer was of size $16 \times 16 \times 64$.
5. Layer 5:128 kernels of size 3×3 with a stride of 1 and padding 1 were used for the convolution operation to give an output volume of size $16 \times 16 \times 128$. No pooling layer was used for reasons mentioned above. The output feature map for this layer was of size $16 \times 16 \times 128$.
6. Layer 6: 28 kernels of size 3×3 with a stride of 1 and padding 1 were used for the convolution operation to give an output volume of size $16 \times 16 \times 128$. An average pooling layer of size 2×2 with a stride of 2 was used to give an output volume of $8 \times 8 \times 128$.
7. Layer 7: 256 kernels of size 3×3 with a stride of 1 and padding 0 were used for the convolution operation to give an output volume of size $6 \times 6 \times 256$. An average pooling layer of size 2×2 with a stride of 2 was used to give an output volume of $3 \times 3 \times 256$.

- Dropout Layer: A Dropout layer was included in the model for regularization of the network to minimize overfitting of the network.
- Output Layers: The output consists of 3 layers. The fully connected layer had seven neurons followed by a SoftMax activation layer. A classification layer has been added to give out corresponding labels for the given input images.

3.1.4 Training

With the given training parameters and hyperparameters in Table 3.1.4, the CNN was able to maintain an accuracy of over 99% on the validation set throughout the training process.

- Optimizer: After conducting experiments with both Stochastic Gradient Descent (SGDM) and Adam optimizers, we concluded that the Adam optimizer resulted in higher accuracy on our CNN model. Also, training was faster using the Adam optimizer.
- InitialLearnRate: Setting initial learning rate to 0.001 gave us the best results. Having a sufficiently lower learning rate ensures that the model converges without overshooting. Although, if the learning rate is too low, the model might take forever to converge.

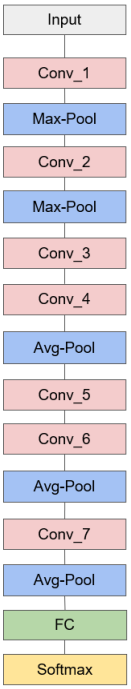

Figure 22: CNN Architecture

Table 1: Hyperparameters used for our network

Parameters/Hyperparameteres	**Value**
Optimizer	'adam'
InitialLearnRate	0.002
LearnRateSchedule	'piecewise''
LearnRateDropPeriod	3
LearnRateDropFactor	0.5
MaxEpochs	20
shuffle	'every-epoch'
ValidationData	'testset'
ValidationFrequency	10
MiniBatchSize	64
Verbose	False
ExecutionEnvironment	'gpu'

Table 2: Effect of different optimizers with all other hyperparameters fixed

OPTIMIZER	**Adam**	**SGDM**
Accuracy (%)	99.32	97.28

Table 3: Effect of different initial learning rate with all other hyperparameters fixed

INITIAL LEARNING RATE	**0.001**	**0.01**
Accuracy (%)	99.32	95.69

- LearnRateDropPeriod & LearnRateDropFactor: These two hyperparameters are usually used together. Decreasing the learning rate after a certain period (say after 2 epochs or 3 epochs) by some factor makes the optimization fine-grained allowing model to gradually approach optimal weights and biases. Setting 'LearnRateDropPeriod' to 3 and 'LearnRateDropFactor' to 0.5 worked best for our CNN.
- MaxEpochs: After 20 epochs, our network achieved convergence, indicating that the model has learned the underlying patterns in the data and is no longer improving significantly with further training.
- Shuffle: While shuffling data is generally considered a best practice in machine learning as it ensures that the model sees different batches of samples during training, making it more robust, our experiments did not reveal any noticeable changes in the performance of our CNN when we did not shuffle the data. This is likely because the model was trained to recognize only a small amount of characters, which means that the dataset is relatively small, and the model is less likely to learn patterns that are specific to the order of the data. If the dataset were larger or more complex, shuffling the data may have a greater impact on the model's performance.
- ValidationData & ValidationFrequency: We used 25% of the dataset as our validation set, and set the validation frequency to 10 to ensure that the CNN's performance was validated every 10 iterations. This approach allowed us to monitor the model's progress during training.
- MiniBatchSize: We conducted experiments with three different mini-batch sizes: 16, 32, and 64. Our results showed that the algorithm performed well with all three values. However, considering the trade-off between computational cost and convergence speed, we decided to use a mini-batch size of 64, as it resulted in faster convergence and reduced computational cost.

We have trained our CNN model using an Nvidia RTX 3060 GPU. The training process is illustrated in Fig 23.

3.1.5 Results

Our trained CNN model achieved an accuracy of 99.32% on the validation set, which is comparable to the state-of-the-art performance of various character/digit recognition models that typically achieve around 99.8% accuracy. While there is still room for improvement, our model's performance is commendable and demonstrates its high level of accuracy in recognizing characters and digits. To better illustrate the performance of the trained CNN, we plotted its confusion matrix and confusion chart on the validation dataset, as shown in Fig 24 and Fig 25. Fig 26 shows some specific correct predictions on the validation dataset.

At last, we tested our CNN with the segmented characters from the source image, and it turns out that the CNN achieved 100% accuracy on these data, as shown in Fig 27.

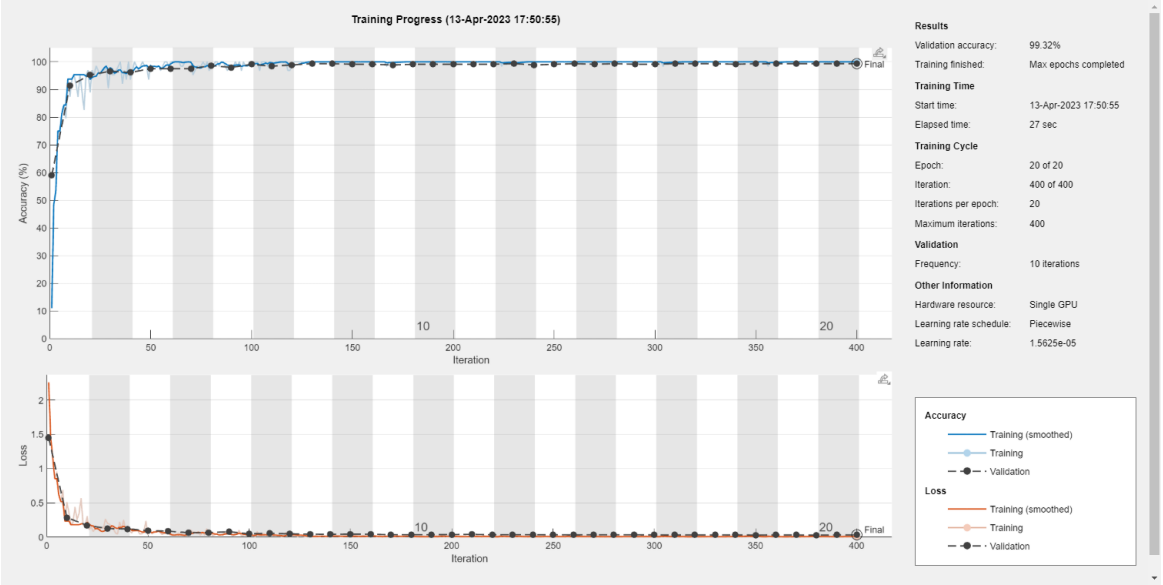

Figure 23: Training process of our CNN

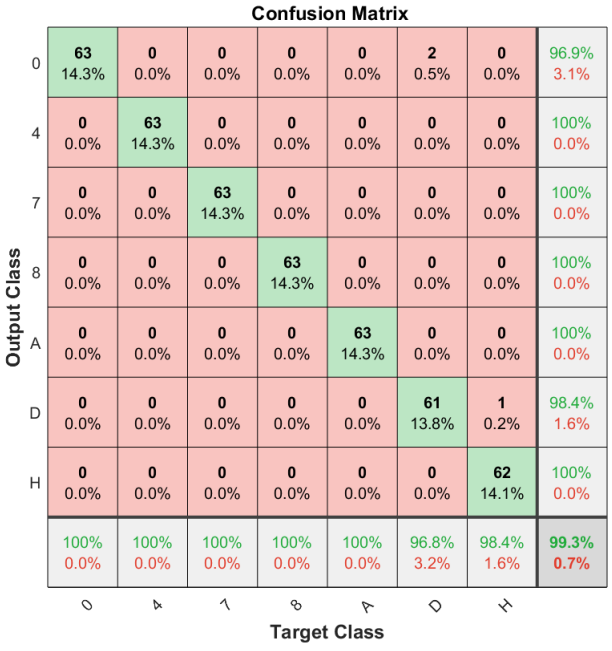

Figure 24: Confusion Matrix of the trained CNN

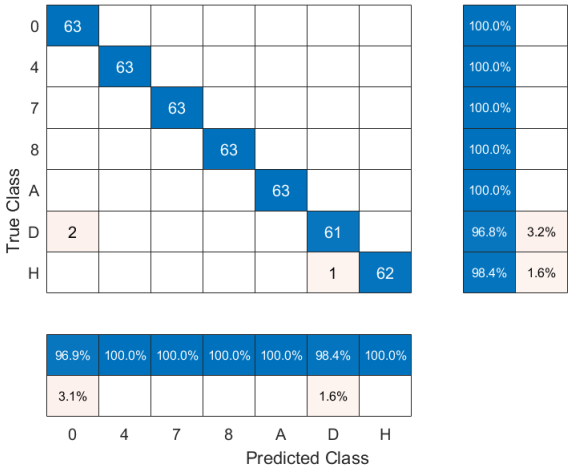

Figure 25: Confusion Chart of the trained CNN

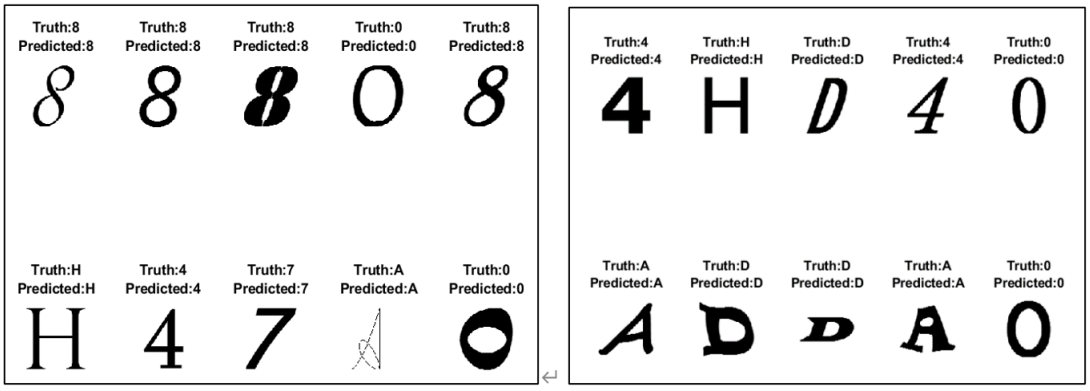

Figure 26: Predicted Labels on Validation set

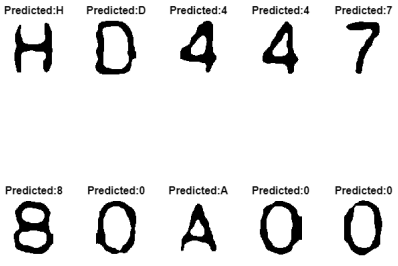

Figure 27: Predicted Labels for the segmented characters

3.2 SVM based character recognition model

3.2.1 Introduction

The aim of this task is to develop a non-CNN based character recognition model in MATLAB to classify each character segmented in part 1 of the BMP image of a label on a microchip. We chose to use SVM with binary encoders to build our classifier because our dataset consists of binary images. Using binary encoding simplifies the classification process for SVM by allowing it to more easily identify the optimal boundary or hyperplane. Moreover, successful multi-class classifiers have been built using SVMs.

3.2.2 Dataset

The same dataset that was used to train the CNN based classifier was used. The dataset comprised a total of 1778 binary images of seven different alphanumeric characters: 0, 4, 7, 8, A, D, and H. All these images were of size 128×128.

3.2.3 Model Architecture

The classifier architecture consists of two main components: feature extraction and classifier training. The first component focuses on extracting relevant features from the input data, while the second component leverages the extracted features to train the classifier.

1. Feature Extraction: For feature extraction, we utilized MATLAB's computer vision toolbox. After evaluating three popular feature extraction algorithms, including Speeded-up Robust Features (SURF), Histogram of Oriented Gradients (HOG), and Local Binary Pattern (LBP), we ultimately selected the HOG algorithm for our approach. The HOG algorithm segments the data into several grids cells and computes the gradient and magnitude of the pixel clusters in each grid cell. Given that our dataset comprised binary images, we thought this algorithm was particularly well-suited for our needs.

   We set the grid size, or cell size, to [20 20] for the HOG algorithm. While a smaller grid size is generally preferred, we opted for this larger size due to the relatively large size of the images in our dataset. This choice allowed us to strike a balance between computational cost and performance. The process of feature extraction using the HOG algorithm can be observed in the images shown in Fig 28.

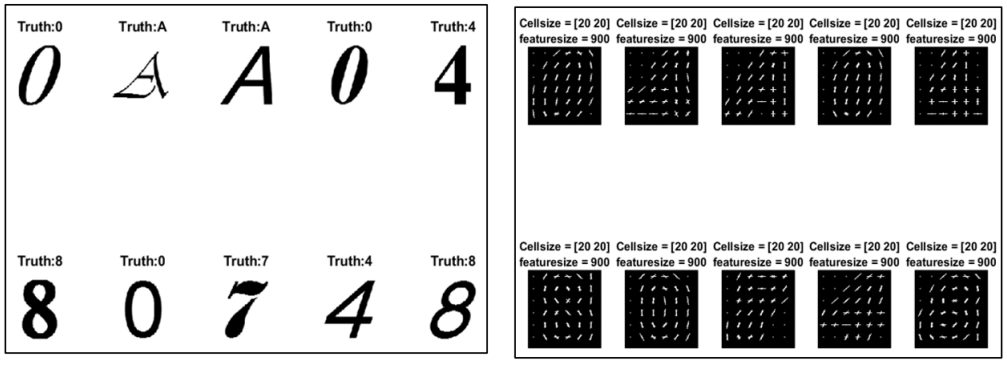

Figure 28: left: Sample training data for SVM; right: HOG Features extracted from sample data

2. Training the Classifier: The dataset was split into two subsets for the purpose of training and validating a model. Specifically, 75% of the data was allocated to the training set, and the remaining 25% was used as the validation set. To train the dataset for multi-class classification and obtain corresponding labels, we leveraged MATLAB's Machine Learning Toolbox. Specifically, we used the 'fitcecoc' function within the toolbox. This function is designed for training support vector machines (SVM) using the error-correcting output codes (ECOC) method, which is a popular approach for multi-class classification. With this function, we were able to train the dataset and obtain labels for each class.

3.3 Results

The trained classifier using the SVM achieved an accuracy of 98.4% on the validation set which is comparable to the accuracy of our trained CNN on the same dataset. This time, We also used Confusion matrices and Confusion charts as metrics for performance evaluation, as shown in Fig 29 and Fig 30.

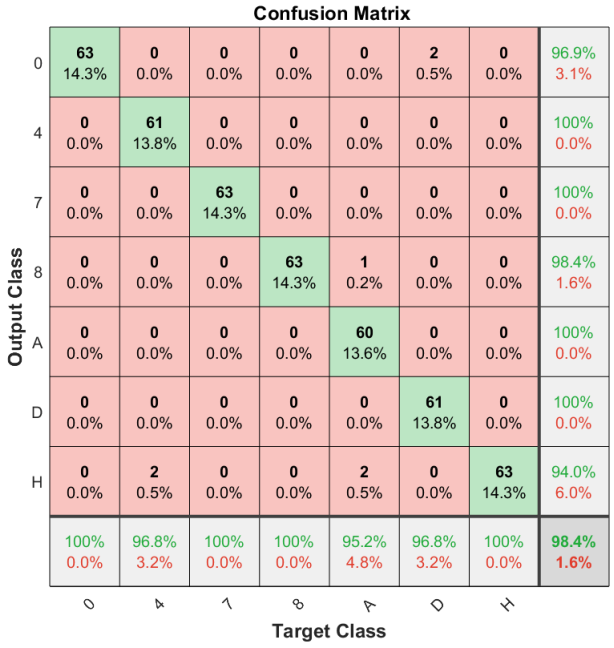

Figure 29: Confusion Matrix of the trained SVM

3.3.1 Character recognition on Validation Data

lthough the SVM-based classifier generally performs well, as shown in the figures above, it is worth noting that it did misclassify three instances in a row during evaluation on the validation set.

For the segmented characters from the source image, the SVM based classifier misclassified character 'h' as '8' but classified all other characters correctly.

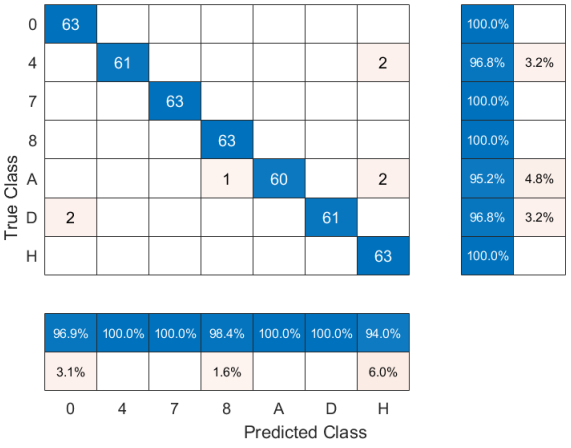

Figure 30: Confusion Chart of the trained SVM

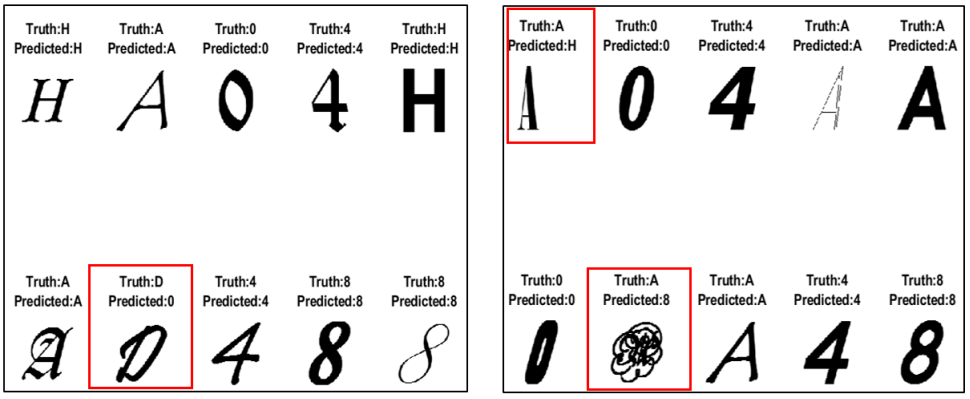

Figure 31: Predicted Labels of SVM on Validation set, where the wrong predictions are denoted with red squares

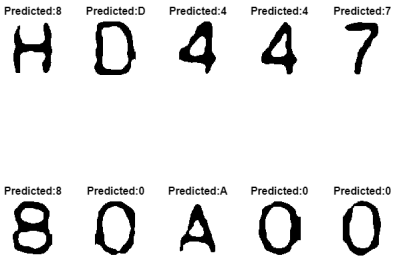

Figure 32: Predicted Labels of SVM for the segmented characters

3.4 Subtask 3: Comparison of CNN and SVM based Models

The performance of both trained classifiers on the provided dataset is strong; however, the CNN-based classifier outperforms the SVM-based classifier by a slight margin. Based on our analysis, we have made the following observations and inferences regarding the accuracy and efficiency of the two approaches.

1. While the CNN and the SVM demonstrate similar performance on the validation sets, it is notable that the CNN performs better on the segmented dataset. This can be attributed to the CNN's capability to learn both low-level and high-level features, which enables it to extract more meaningful information from the segmented data compared to the SVM.

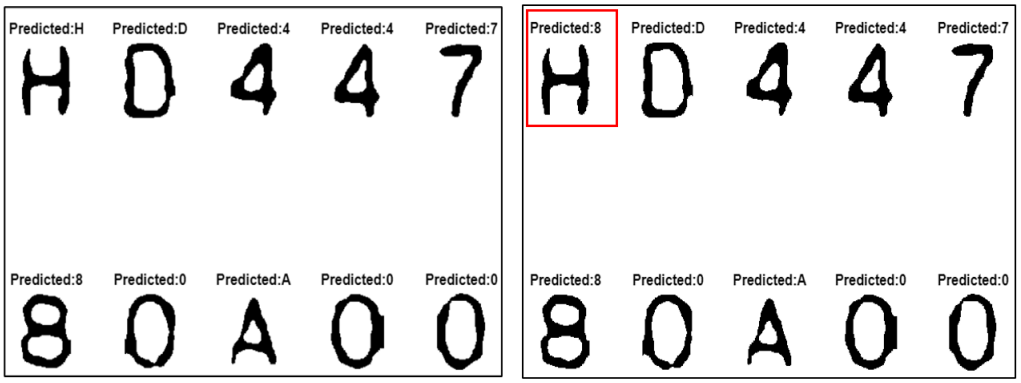

Figure 33: left: Predicted labels for the segmented data by CNN; right: Predicted labels for the segmented data by SVM

2. Furthermore, in this specific scenario, the performance of the SVM and the CNN is comparable since the dataset is small, binary, and with small classes numbers. However, CNNs have the potential to excel on higher-dimensional and more complex datasets due to their ability to learn hierarchical features from raw data.

3. The SVM on the other hand trains faster on the given dataset since the number of trainable parameters is far less than that of the CNN.

4. The performance of the SVM on this dataset can further be improved by decreasing the grid size for feature extraction. Although, that will lead to increase in computational complexity of the algorithm.

5. CNNs are better than SVMs for character recognition since they don't require explicit feature extraction during training and can automatically identify the most relevant features.

3.5 Discussions

In this section, we will discuss why CNNs can have better performance than other methods. One of the reasons that makes CNN so powerful is that its output is equivariant to the shift in inputs. This means that if we shift the input image with some distance, the

corresponding feature maps produced by the CNN will also be shifted with in the same way. However, for other machine learning models such as neural networks and support vector machines, they could not achieve shift equivariance, which means that there is no direct relationships between the shifted input images and corresponding output feature maps. This is why CNN usually has more advantages than other non-CNN based models in terms of image processing.

However, CNN also has its limitations despite its power feature of shift equivariance. First of all, CNN cannot achieve rotational equivariance, which makes it could hardly recognize a rotated image. The researches conducted by Cohen and Welling later solved this problem by introducing group convolution [2] [3]. Also, the pooling operation used in CNN might also result in loss of important information, thus causing wrong prediction. To solve this problem, Sabour and Hinton proposed the capsule network structure [4], which can achieve better performance than CNN.

4 Conclusion

In summary, we performed image smoothing, sub-image creation, thresholding, edge-detection and character segmentation on the given BMP image of a microchip to get distinct characters. Appropriate labels were then generated for these segmented characters using trained character recognition models. We used two different models for character recognition. The first model was a CNN-based model, which has a strong track record in character recognition. The second classifier was based on SVM, which was commonly used for classification tasks before the development of CNNs. Both models performed well on the small dataset with a small number of classes, accurately identifying the segmented characters. The CNN-based model had a slight edge over the SVM-based model due to its ability to learn more complex features, and its end-to-end architecture provided practical advantages. It is worth noting that the performance of both models was comparable due to the small dataset size and number of classes. However, it is expected that the CNN-based model will perform considerably better and more efficiently on larger and more complex datasets.

References

[1] “9.4 Weighted Quick Union (WQU) · Hug61B.” [Online]. Available: https://joshhug.gitbooks.io/hug61b/content/chap9/chap94.html

[2] T. S. Cohen and M. Welling, “Group Equivariant Convolutional Networks,” Jun. 2016, arXiv:1602.07576 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1602.07576

[3] ——, “Steerable CNNs,” Dec. 2016, arXiv:1612.08498 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1612.08498

[4] S. Sabour, N. Frosst, and G. E. Hinton, “Dynamic Routing Between Capsules,” Nov. 2017, arXiv:1710.09829 [cs]. [Online]. Available: http://arxiv.org/abs/1710.09829