ME5405: Machine Vision Project

Project Group No. 22:

Praveen Krishnapur - A0304688U

Pan Shijiao - A0304336M

Zhang Yutao - A0304300E

November 20, 2024

Contents

Abstract

This project report presents a MATLAB-based machine vision system for processing and analysing images. Key tasks include contrast enhancement, character straightening, smoothing, high-pass filtering, and segmentation of text images. Additionally, an unsupervised classification model was developed and trained to recognize characters, incorporating preprocessing techniques and hyperparameter tuning. The report evaluates each processing step and compares various algorithms and methods.

Introduction

Machine vision has become an essential technology in recent times, playing a critical role in fields like automation, quality inspection, and artificial intelligence applications. This project aims to explore fundamental machine vision techniques, including reading, displaying, processing, segmenting, and identifying characters in digital images. Using MATLAB, various image processing and analysis methods are applied to demonstrate the power and versatility of machine vision in real-world applications. There are two sets of tasks associated with two images: Image 1: Hello.jpg and Image 2: hello_world.jpg. The tasks are as follows:

Image 1 Hello.jpg

1. Display the original image on screen. Convert the image to grayscale image. Experiment with contrast enhancement of the image. Comment on the results.
2. Straighten the characters “HELLO!” in the image using the three different interpolation methods mentioned in class. Compare and comment on the results.
3. Implement and apply a 5x5 averaging filter to the image. Experiment with filters of different sizes. Compare and comment on the results of the respective smoothing methods.
4. Implement and apply a high-pass filter on the image in the frequency domain. Compare and comment on the resultant image in the spatial domain.
5. Segment the image to separate and label the different characters as clearly as possible

Image 2 hello_world.jpg

1. Display the original image on screen.

2. Create an image which is a sub-image of the original image comprising the middle line – HELLO, WORLD.
3. Create a binary image from Step 2 using thresholding.
4. Determine a one-pixel thin image of the characters.
5. Determine the outline(s) of characters of the image.
6. Segment the image to separate and label the different characters.
7. Using the training dataset provided, train the (conventional) unsupervised classification method of your choice (i.e., self-ordered maps (SOM), k-nearest neighbors (kNN), or support vector machine (SVM)) to recognize the different characters ("H", "E", "L", "O", "W", "R", "D").
8. Throughout step 7 (training of the classifier), also experiment with pre-processing of the data (e.g., padding/resizing input images) as well as with hyperparameter tuning. In your report, discuss how sensitive your approach is to these changes.

Figure 1 *Image 1 HELLO.jpg*

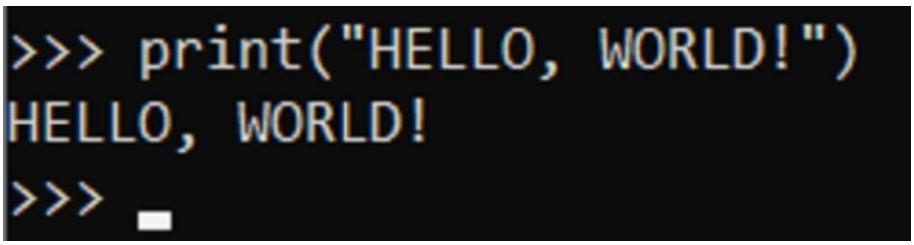

Figure 2 *Image 2: hello_world.jpg*

Image 1: HELLO.jpg Processing

Task 1 Display and Enhance Image Contrast

The original image is read in MATLAB using the **imread** function from the Image Processing Toolbox. The provided image has a resolution of 2236x1258, so it is read as a matrix of size 2236x1258, with each element representing the brightness or intensity value. Since the image is in color, the matrix includes a depth of 3 layers, each corresponding to the Red, Green, and Blue (RGB) channels. It is displayed using **imshow** function. (figure 1)

To convert the image to grayscale, a custom function called color2gray is used. This function calculates the grayscale intensity by taking a weighted average of the RGB values for each pixel, accounting for the human eye's greater sensitivity to green. The grayscale value is computed as:

$$gray\_img(i,j) = (R(i,j) \times 0.2989) + (G(i,j) \times 0.5870) + (B(i,j) \times 0.114)$$

Figure 3: *Gray scale image of HELLO.jpg*

With the grayscale image obtained, we can proceed with contrast enhancement. Using a custom function named **contrast_stretch**, the image's contrast is adjusted by applying different minimum and maximum intensity pairs. This function takes the image and specified contrast limits to scale the pixel values accordingly. Three contrast pairs were applied: (0, 255), (40, 200), and (80, 150).

With contrast stretching applied, noticeable differences emerge in the clarity between foreground and background. When the contrast is set to the full range (0, 255), the separation between the foreground and background is sharp, making details stand out clearly. As the

contrast range is reduced to (40, 200) and then further to (80, 150), the image progressively darkens, and finer details become less distinct. Lowering the contrast also reduces intensity variations, causing the image to appear more uniform and limiting depth perception in textured regions

Figure 4: *Contrast stretched images (0,255); (40,200); (80,150)*

Task 2 Straighten “HELLO!” with Interpolation

To straighten the characters "HELLO" in the image, we applied a rotation of 2 degrees using three different interpolation techniques: nearest neighbour, bilinear, and bicubic interpolation.

- Nearest Neighbour Interpolation: This method assigns the value of the nearest pixel to the new pixel, resulting in a faster computation but potentially producing a blocky or jagged image.
- Bilinear Interpolation: It calculates the new pixel value using a weighted average of the four nearest pixels, offering smoother results than nearest neighbour interpolation.
- Bicubic Interpolation: This technique uses a weighted average of the 16 nearest pixels, providing even smoother and more precise images with higher computational complexity. Custom functions **rot_img_nn, rot_img_bilinear**, and **rot_img_bicubic** were implemented to perform the rotation using these interpolation methods. The results are in figure 5.

The comparison of the three interpolation techniques—nearest neighbor, bilinear, and bicubic—reveals noticeable differences in image quality and computational efficiency. Bicubic interpolation produces the smoothest and most detailed results, followed by bilinear interpolation, with nearest neighbor being the least smooth. However, these improvements in image quality come at the cost of increased computation time: bicubic interpolation takes the longest, followed by bilinear, with nearest neighbor being the fastest. This demonstrates a trade-off between computational time and the smoothness and detail in the resulting images.

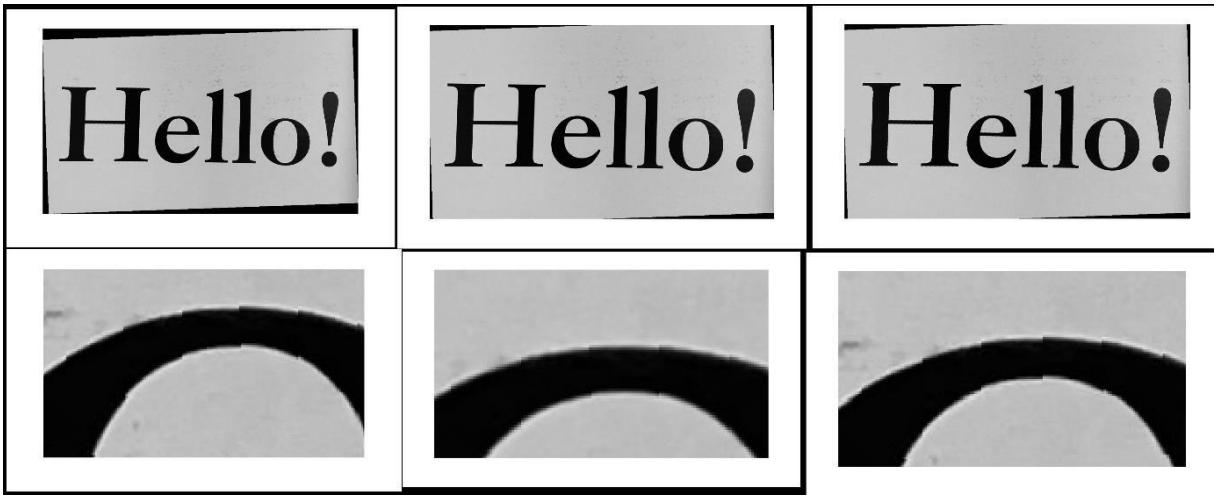

Figure 5: *Bicubic, Bilinear, NN interpolations and their zoomed in parts respectively*

Task 3 Smoothing with Averaging Filter

Use the **imread** function to read the specified image file “HELLO.jpg” and store it in the variable “img”. Create a 5x5 averaging filter using the **fspecial** function, then apply this filter to the original image using the **imfilter** function to obtain the processed image “smoothedImg5x5”. Finally, use **subplot** to display the original image alongside images processed with average filters of different sizes in the same window for comparison and analysis. The results are in figure 6.

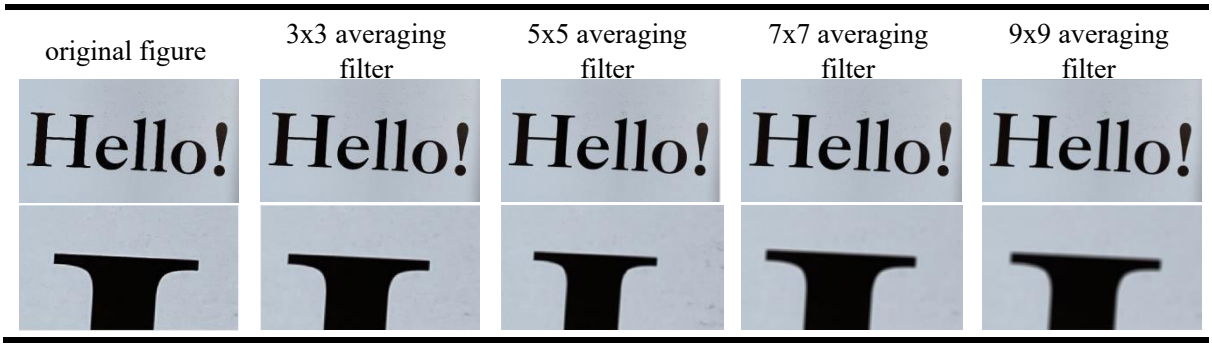

Figure 6: *The results processed with different sizes of averaging filters*

The original image features crisp edges, clear text, and high contrast. Applying a 3x3 averaging filter produces a slight smoothing effect, removing a small amount of noise while preserving edge details and maintaining sharp text. Compared to the 3x3 filter, the 5x5 averaging filter increases the smoothing effect, slightly blurring edges and reducing text sharpness, though the text remains legible. The 7x7 filter further smooths the image, with more noticeable blurring of edges, reduced text contrast, and some loss of detail. The 9x9 filter produces the strongest smoothing effect, significantly reducing image details. Edges become blurred, text sharpness diminishes, and portions of the text start to lose clarity.

In summary, larger filter sizes produce stronger smoothing effects but result in greater detail loss. A 3x3 or 5x5 averaging filter is ideal for removing minor noise while keeping the image sharp. If the image has more noise and edge accuracy is less critical, a 7x7 or 9x9 filter may be used, though it will lead to more pronounced blurring.

Task 4 Apply High-Pass Filter in Frequency Domain

In MATLAB, a variety of high-pass filters can be used to enhance high-frequency details in an image. These include the ideal high-pass filter, the Butterworth high-pass filter, and the Gaussian high-pass filter. For the strongest retention of high-frequency detail, the ideal highpass filter is given primary consideration, directly removing low-frequency components and retaining high-frequency components. Hence, the result of applying a high pass filter to the image in the frequency domain is as follows.

Figure 7: *ideal high-pass filtered image (Cutoff frequency d0=30)*

Next is the result of applying the Butterworth high-pass filter：

Figure 8: *Butterworth High-Pass Filtered Image(the order n=10)*

The last one is the result of applying a Gaussian high-pass filter：

Figure 9: *Gaussian High-Pass Filtered Image*

By comparing the results produced by the above three applications of different high-pass filters, it can be seen that each of the three methods has its own advantages and disadvantages. For an ideal high-pass filter, the resulting high-frequency detail is retained the most, but a significant ringing effect may be produced. Image edges are sharp but noisy. However, it can be adjusted by changing the cutoff frequency d0 to select the image that best satisfies the requirements.

For Butterworth high-pass filters, it smooths the transition to the high-frequency component, thereby reducing the ringing effect and providing a good balance between high-frequency retention and smoothing. Image edges are sharp and less noisy. And different effects can be obtained by changing the order n.

For Gaussian high-pass filters, it provides smooth high-frequency retention and usually does not introduce significant ringing effects. It is usually suitable for images that are more sensitive to noise.

Task 5 Segment and Label Characters

To segment the characters in the image "HELLO!", the straightened grayscale image was first thresholded using Otsu's thresholding method. This was implemented using the custom function called **otsu_threshold**. Otsu's thresholding is an adaptive, global thresholding technique that automatically determines the optimal threshold by maximizing the variance

between the foreground and background classes. Other global thresholding methods, such as Median thresholding and Constant thresholding, were implemented as custom functions (**Median_threshold, constant_threshold**) but were not used for final segmentation. Median thresholding uses the median intensity value as the threshold, while constant thresholding applies a fixed threshold value. Although these methods are effective in certain situations, Otsu's method provided a more accurate separation for this image. For more robust thresholding in images with varying lighting, local thresholding methods can be used, which adapt to small regions within the image.

After thresholding, the complement of the binary image was taken, and the **bwlabel** function from the Image Processing Toolbox was applied to label the connected components. The **bwlabel** function scans through the binary image, identifying connected regions by assigning unique labels to each component. This allows each character or disconnected part to be treated as a separate entity. Following labeling, the **regionprops** function was used to obtain bounding boxes for each labeled region. **regionprops** calculates properties of the labeled regions, such as area, centroid, and bounding box, which are useful for isolating and analyzing individual components.

In total, 7 regions were identified, corresponding to 6 characters. The exclamation mark is segmented into two regions due to the disconnected line and dot components, which remain distinct. While smoothing could bridge this gap, it would also blur critical details in other characters, so it is preferable to treat them as separate regions.

Below you can find the segmented images from the original image:

Figure 10: *Segmented Characters of HELLO.jpg*

Image 2: hello_world.jpg Processing

Task 1 Display Original Image

The original image is read in MATLAB using the imread function from the Image Processing Toolbox. where Image 2 is a JPEG color image, comprising three lines of characters. The following figure shows the result of the code implementation.

Figure 11: *Output of helloworld.jpg*

Task 2 Extract Middle Line Sub-Image

Since there are only three rows of the original image, and only the middle row is the sub-image being asked to be rendered in this task, it only needs to be indexed manually based on the exact location of “HELLO, WORLD”.

Figure 12: *sub-image of the original image*

Task 3 Creat a Binary Image

In this task, the extracted sub-images are first converted to grayscale and then the thresholding method is applied to create a binary image. In this process, the threshold needs to be adjusted according to the contrast and brightness of the image. The figure below presents the result with a threshold value of 130.

Figure 13: *binary image with a threshold value of 130*

Task 4 Generate One-Pixel Thin Image

The Zhang-Suen algorithm is a classic thinning algorithm designed to progressively thin characters or shapes down to a single-pixel width. This algorithm iteratively checks each foreground pixel in the image to determine if it meets the conditions for deletion, removing unnecessary edge pixels until the image reaches a stable state. Each iteration consists of two sub-steps. In each sub-step, the algorithm scans all foreground pixels in the image and marks "deletable" pixels according to certain rules. Then, it sets these marked pixels as background (deletes them). This process repeats until no more pixels can be deleted.

The specific steps are as follows:

Step 1: Traverse each foreground pixel. If the pixel meets the following four conditions, it is marked as "deletable":

- The number of foreground pixels in its 8-neighborhood is between 2 and 6.
- The number of transitions from background to foreground in its 8-neighborhood is At least one of its neighboring pixels at the top, right, or bottom is a background pixel.
- At least one of its neighboring pixels at the right, bottom, or left is a background pixel.

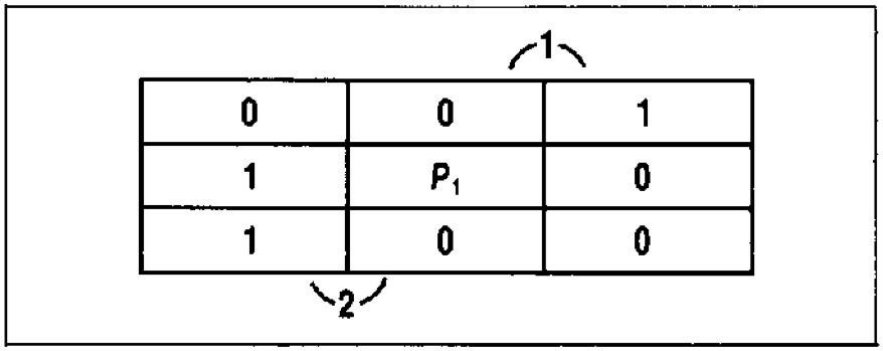

Figure 14: *Counting the 01 Patterns in the Ordered Set P2,P3,P4,...P8,P9*

Step 2: Traverse each foreground pixel again. This time, it needs to meet the following four conditions to be marked as "deletable":

- The number of foreground pixels in its 8-neighborhood is between 2 and 6. The number of transitions from background to foreground in its 8-neighborhood is At least one of its neighboring pixels at the top, right, or left is a background pixel.
- At least one of its neighboring pixels at the top, bottom, or left is a background pixel.

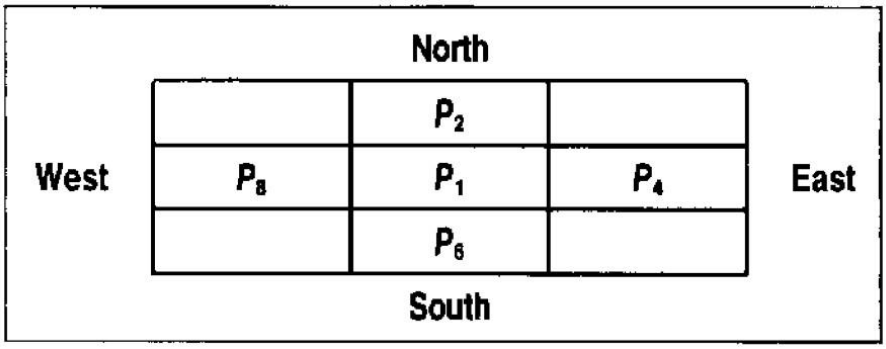

Figure 15: *Points under Consideration and their Location*

In each iteration, the "deletable" pixels marked in both sub-steps are set to the background. This process repeats until no pixels meet the conditions for deletion, resulting in a single-pixelwidth thinned image.

The implementation process of the algorithm is represented in the following logic flowchart.

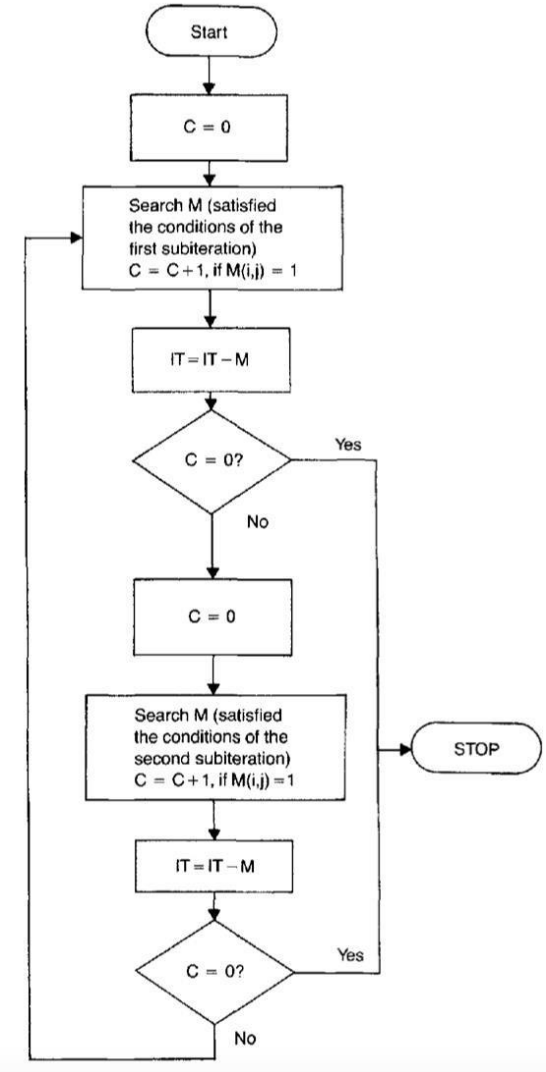

Figure 16: *Flowchart of Zhang-Suen Thinning Algorithm*

For the thinning process, choosing the appropriate thresholding method is crucial, as it significantly affects the outcome of the process. The choice of thresholding method depends on the specific application and the characteristics of the input image. Here, two different thresholds are used to invoke the Zhang-Suen algorithm function. The final effect is shown in the image below.

Figure 17: *One-pixel Thin Image of hello_world.jpg*

Task 5 Extract Character Outlines

In this problem, the morphological operation of "erosion" is used. The imerode function is called to perform this operation. The erosion operation scans the binary image with a structuring element (a 3x3 square in this case). If all the pixels within the structuring element are 1, the output pixel value is set to 1; otherwise, it is set to 0. The erosion operation shrinks the white areas (characters) in the image, helping to remove small noise. Then, by calculating the difference between the original binary image and the eroded image, the edges of the characters are extracted. ~erodedimg is the inversion of the eroded image, and binary_image & ~erodedimg takes the intersection of the binary image and the inverted eroded image, thereby obtaining the character's contour (edges). This step effectively extracts the external edges of the characters.The result is shown below.

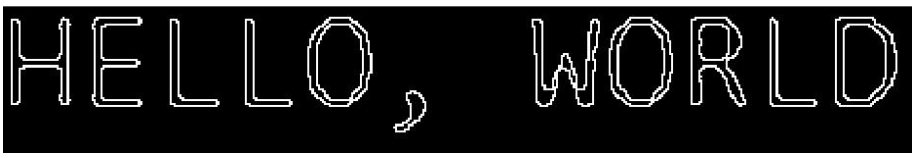

Figure 18: *Outlines of Characters of hello_world.jpg*

Task 6 Segment and Label Characters

The **bwlabel** function is used to label the connected regions in a binary image. **bwlabel** assigns different labels to the connected pixel regions in the image, where **numLabel** represents the number of connected regions, and labelImg is the image containing the labels for each connected region. Then, the **regionprops** function is used to calculate the properties of each

connected region, particularly the BoundingBox. The **Boundingbox** for each character is a rectangle that includes the starting position, width, and height of the character region, and this information is stored in the structure array **props**.

During processing, the edges of the characters may overlap with the bounding box, leading to unclear recognition. Therefore, the **imcrop** function is used to crop the region for each character. The **padding** is used to expand the bounding box to avoid the character's edges being too close to the bounding box, ensuring the cropped character region is complete and the edges are not lost. Additionally, **max(boundingBox,1)** is used to ensure that the coordinates of the bounding box do not go out of bounds (i.e., less than 1), preventing errors during cropping.

Due to character similarities or other factors, the order of the detected characters may become mixed up. Therefore, a sorting method is applied to ensure that the characters are arranged in the correct order based on their position in the image. By using **arrayfun**, the x coordinates of the top-left corner of each bounding box are extracted and the characters are sorted by their horizontal position (from left to right). The sorted indices, **idx**, are then used to reorder the props array, ensuring that the characters are arranged in the correct order.Following are the segmented characters of image2.

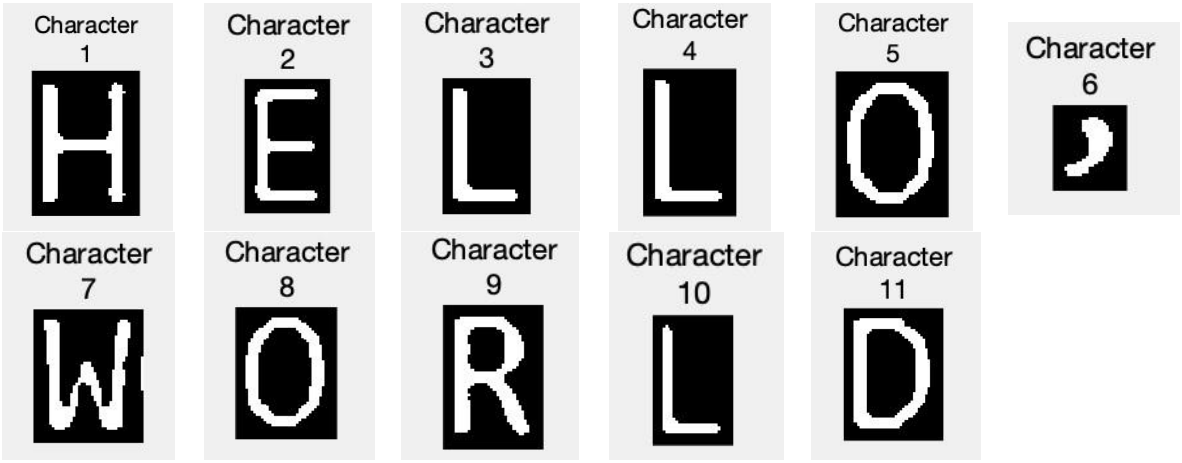

Figure 19: *Segmented Characters of hello_world.jpg*

Task 7 Train a Classifier to Recognize Characters

We selected the K-Nearest Neighbors (KNN) algorithm for classification. KNN classifies an input sample based on the majority class of its nearest neighbors in the feature space, using a

distance metric such as Euclidean distance. It is simple, non-parametric, and effective for various classification tasks. The provided dataset p_dataset_26 contains data for 7 classes: H, E, L, O, W, R, D, with 1016 images per class. As per the instructions, 75% of the dataset was used for training, i.e., 0.75×1016×7 = 5,322 images. The feature set for the model consisted of the pixel values of the binary-converted images. Since each image is of size 128x128 the number of features was **16384**.

A training matrix of size (5322, 16384) was formed and similarly the validation matrix size was (1774, 16384) (using 25% of the dataset). For each image in the validation set, distances to the training images were computed, and the **3 nearest neighbors (k=3)** were used to predict the class by majority voting. Accuracy was calculated by comparing these predictions against actual labels, achieving a **validation accuracy of 87.91%**.

When this model was tested on segmented letters from **Image 2 (Figure 19)**, performance dropped significantly, correctly classifying only 2 out of 7 letters:

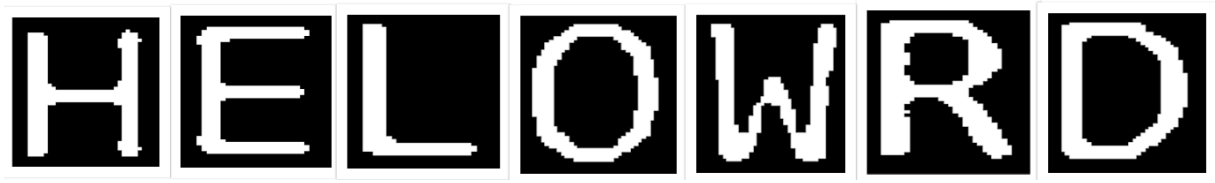

Figure 20:*Test images used for classification*

This loss of accuracy was attributed to the noise in the edges of the test images. The presence of noise likely disrupted the feature representation, leading to misclassifications. From the validation accuracy, it was evident that the model performed well for images without noise. To improve the performance, a combination of additional features and pre-processing techniques must be employed. This could include extracting HOG features or edge features, resizing images in the training data to artificially introduce noise in the edges, and smoothing the test images before classification. Such approaches would enhance the robustness of the model to handle noisy and imperfect test images effectively.

Table1:*Prediction of test images*

Predicted	H	D	H	O	O	L	E
Actual	H	E	L	O	W	R	D

Task 8 Experiment with Preprocessing and Hyperparameter Tuning

Based on the conclusions from Task 7, we decided to enhance the feature extraction process by using **Histogram of Oriented Gradients (HOG) features** and **edge features** instead of the raw pixel intensities of the image. Additionally, we resized the original **128x128 images to 32x32** using bicubic interpolation. These modifications were aimed at addressing the issues identified in Task 7 and improving the accuracy of the model.

Feature Selection and Image Resizing:

1. **HOG Features:**

HOG features were selected as they effectively capture the general shape and structure of the image. By focusing on the distribution of gradient orientations, HOG emphasizes the essential patterns and ignores irrelevant noise, making it ideal for recognizing characters with distinct shapes.

2. **Edge Features:**

Including edge features helped train the model to specifically identify the edges of the characters, which are critical for distinguishing letters. This feature set made the model more robust against variations in image intensity and noise.

3. **Image Resizing:**

Resizing the images to 32x32 reduced the dimensionality of the feature set, simplifying the computation and reducing overfitting. Furthermore, resizing introduced minor noise at the edges, which helped make the model more robust to similar distortions in real-world test images.

Training and Validation:

- The **training matrix** was constructed with a size of 5322×1168 (5322 images, with 144 H OG features + 1024 edge features from the resized 32x32 images).
- The **validation matrix** had a size of 1774×1168.
- Using KNN with k=3 the model achieved a **validation accuracy of 91.06%**, showing a notable improvement compared to the raw pixel-intensity model.

Testing Results:

When tested against the segmented letters from **Figure 20**, the model achieved **100% accuracy**, successfully classifying all test images.

Table2:*Prediction of test images*

Predicted	H	E	L	O	W	R	D
Actual	H	E	L	O	W	R	D

Hyperparameter Tuning:

For the **K-Nearest Neighbors (KNN)** algorithm, the only hyperparameter to tune is **K**, the number of neighbours used during classification. We selected K as an odd number to avoid ties when determining the majority class during the voting process.

To identify the optimal K-value, we tested the model with various K-values and recorded the validation accuracy. The plot below shows the relationship between K-values and validation accuracy. The model achieved its highest validation accuracy of **91.79% (~92%)** at K=7. This demonstrates that by carefully tuning the hyperparameters, we can optimize the model's performance to its maximum potential.

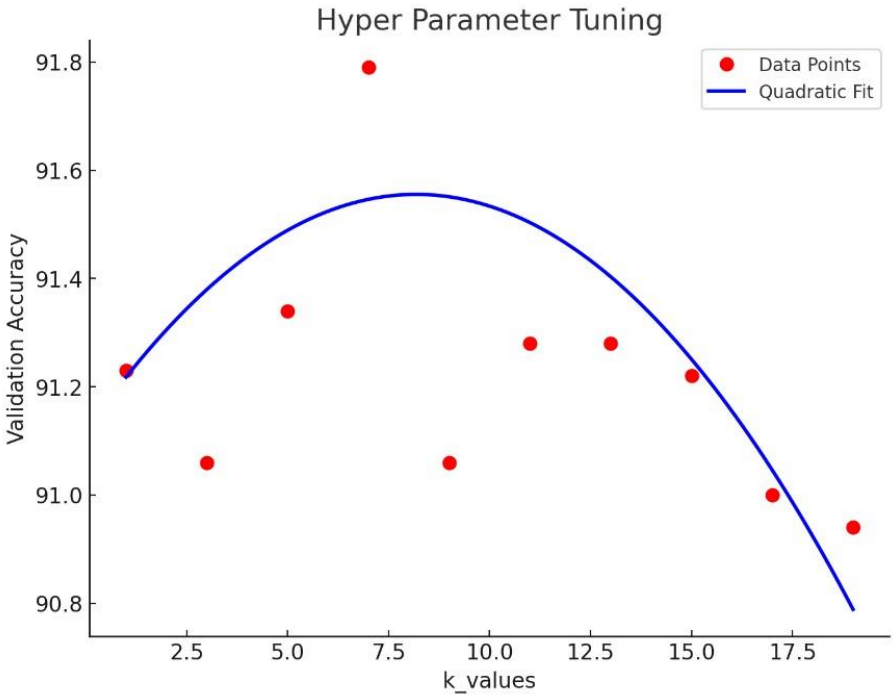

Figure 21:*K vs val_acc for HPT*

By incorporating HOG features, edge features, resizing images, and tuning the hyperparameter K, we significantly improved both the validation accuracy (from 87.91% to 91.79%) and the

test accuracy (from 28.57% to 100%). These enhancements ensured that the model effectively captured the essential features of the characters while being robust to noise and distortions. This emphasizes the importance of feature selection, pre-processing, and hyperparameter tuning in optimizing machine learning models for image classification tasks.

Bonus Task

For the bonus task, the letters **'e', 'H', 'l',** and **'o'** from **Image 1** were tested using the same KNN model with K=7. The model successfully classified all the letters correctly:

Table3:*Prediction of test images*

Predicted	E	H	L	O
Actual	E	H	L	O

It is not surprising that the model correctly labeled **H** and **O**, as these characters have distinct shapes. For 'e' and 'l', the success can be attributed to the fact that the training dataset contained images resembling small **'e'** and **'l'**, which closely matched the test samples (as shown below).

Figure 22: *Similar looking letters to small E and L*

This result reinforces the effectiveness of adding HOG features and edge features, as they significantly improve the model's robustness and accuracy. These features capture the structural essence of the characters, enabling the model to generalize well even for variations like lowercase letters.

Appendix

Custom Functions

The following custom functions were utilized in the **main.m** file to perform specific tasks required for the overall implementation. Each function is described below to provide clarity on its purpose and contribution to the script.

1. **color2gray**: Converts a color image to grayscale by combining color channels based on luminance weights. This simplifies the image for further analysis or processing.
2. **constant_threshold**: Applies a constant thresholding technique to an image, segmenting it into binary form based on a fixed intensity value. It is useful for creating binary masks in image segmentation.

3. **contrast_stretch**: Enhances the contrast of an image by stretching its intensity range to a specified minimum and maximum. This technique improves visibility in images with poor lighting.
4. **otsu_threshold**: Implements Otsu's method to determine the optimal threshold for binarizing an image, maximizing inter-class variance. It automates the process of choosing a threshold for segmentation.
5. **rot_img_bicubic**: Rotates an image by a specified angle around a pivot point using bicubic interpolation for smooth resampling. This ensures high-quality image rotation for detailed visualization.
6. **rot_img_bilinear**: Rotates an image by a specified angle around a pivot point using bilinear interpolation for moderate resampling quality. It balances processing time and output quality.
7. **rot_img_nn**: Rotates an image by a specified angle around a pivot point using nearestneighbor interpolation for fast but low-quality resampling. It is ideal for quick rotations where precision is less critical.
8. **custom_hog**: Extracts the Histogram of Oriented Gradients (HOG) feature descriptor for a binary image, useful for edge and object detection tasks.
9. **Grad_func**: Calculates the gradient of an image at each pixel using forward differencing, providing edge information for further processing. The gradient values can highlight boundaries and transitions in the image.

In addition to the main script, we implemented some custom functions that were not directly utilized in **main.m** but are presented here for completeness and future reference.

1. edge_detect.m

This is a custom function designed to detect edges in an image using Sobel gradients. The function calculates the gradient magnitude and applies edge linking with a threshold of **0.25 * max(gradient_magnitude)** to determine significant edges. The approach ensures effective edge detection while minimizing noise.

Below is an example of edge detection applied to **Image 1** using the edge_detect.m function:

2. median_threshold.m

Median thresholding is a technique that converts an image to binary by using the median pixel intensity as the threshold. Pixels with intensity greater than the median are set to white, while those below are set to black. This method is robust to outliers and adapts to the overall intensity distribution of the image.

The median_threshold.m function implements this technique, and the resultant binary image is shown below: