Digital Image Processing

Kamil Ilyas —— i202371

September 2023

1 Assignment 1

1.1 Question 1

1.1.1 Proposed Solution

1. **Grayscale Conversion (Intensity Transformation):** The `convert to grayscale` function takes an input color image and converts it to grayscale using the `cv2.cvtColor` function. Grayscale conversion simplifies intensity-based analysis by reducing the image to a single channel.

2. **Image Segmentation:** The `segment_image` function performs image segmentation to identify and extract contours from the input image. It starts by converting the input image to grayscale using the previously defined function. Then, it applies binary thresholding using `cv2.threshold` to create a binary image where objects are separated from the background. Finally, it detects and returns a list of detected contours using the `cv2.findContours` function. These detected contours represent the boundaries of objects or regions in the image.

3. **Shape Analysis:** The `analyze_shape` function analyzes the shape represented by a given contour. It calculates an approximate polygon for the contour using `cv2.approxPolyDP` and checks if it has four vertices, indicating a potential rectangle or square. If a four-sided polygon is identified, it calculates the perimeter, centroid coordinates, and aspect ratio of the bounding rectangle. Depending on the aspect ratio, it categorizes the shape as a "Square," "Rectangle," or "Neither Square nor Rectangle." The function returns a tuple containing the shape type, perimeter, and centroid coordinates for further analysis and printing.

4. **Loop Through Contours:** The code iterates through the detected contours obtained from the image segmentation step. For each contour, it calls the `analyze_shape` function to determine the shape type, perimeter, and centroid coordinates. If a shape is identified, it prints the shape type, perimeter, and centroid information.

5. **Visual Representation:** The code draws the detected contours on the original image using `cv2.drawContours` and displays the result with contours outlined in green using `cv2.imshow`.

1.1.2 Relevance with topics

The code performs digital image processing tasks, starting with the loading of an image and its conversion to grayscale to simplify intensity-based analysis. It then delves into several fundamental concepts in digital image processing:

- **Grayscale Conversion:** The code showcases the transformation of a color image into grayscale, a foundational step for many image processing tasks, aiding in the simplification of image analysis based on pixel intensity.
- **Image Segmentation:** Through binary thresholding and contour detection, the code segments the image into distinct regions, highlighting object boundaries. This process is vital in isolating and identifying objects within an image.
- **Shape Analysis:** The code proceeds to analyze the detected contours, particularly focusing on four-sided polygons. It calculates essential attributes, including the perimeter, centroid, and aspect ratio, enabling the identification of shapes as "Square," "Rectangle," or "Neither Square nor Rectangle."

1.2 Question 2

1.2.1 Proposed Solution

1. **Template Loading:** The function loads two grayscale templates: "boy template" and "girl template". These templates represent patterns of a boy and a girl character, respectively.
2. **Image Loading:** It reads two grayscale images, "left image" and "right image", specified by the input paths "left image path" and "right image path". These images contain characters to be classified.
3. **Error Handling:** The code checks if any of the templates or images fail to load. If any of them are not successfully loaded, an error message is returned, indicating that one or both of the images or templates couldn't be loaded.
4. **Template Matching:** Template matching is performed using "cv2.matchTemplate" to compare the character in the left image with the boy and girl templates. The result is a similarity score for each template match.
5. **Matching Threshold:** A "matching_threshold" is set (0.8 in this case) to determine the classification. If the similarity score for the boy template

exceeds the threshold while the girl template score is below it, the character in the left image is classified as a boy, and vice versa for the girl character. If neither condition is met, it indicates that the gender cannot be determined accurately.

6. **Display Images:** The left and right images are displayed using "cv2.imshow" to provide a visual representation of the characters being classified.

7. **Result:** The function returns a result string indicating the classification outcome, such as whether the left image character is classified as a boy, girl, or if it couldn't be determined.

1.2.2 Relevance with topics

Firstly, the code demonstrates the fundamentals of image processing by employing techniques such as image loading, grayscale conversion, and template matching. These fundamental operations are crucial in manipulating and analyzing images.

Secondly, it touches upon intensity transformation and spatial filtering through the template matching process. The matching is based on intensity patterns, which is an example of spatial filtering, as it analyzes pixel values in local neighborhoods to identify patterns of interest. In this case, it identifies gender-specific features through intensity patterns.

Furthermore, the code indirectly relates to histogram processing. Although not explicitly applied in this context, histogram equalization and adjustments could enhance the accuracy of gender classification by improving image contrast and highlighting relevant features.

Lastly, the relevance of this code to image segmentation lies in its underlying concept of distinguishing characters from the background. While not a segmentation code per se, the differentiation between a character (boy or girl) and the image background represents a simplified form of segmentation, as it isolates the character for gender classification.

1.3 Question 3

1.3.1 Proposed Solution

1. **Image Analysis:** The function takes two grayscale images, original_image and blurred_image, as input. It calculates the Laplacian variance for each image using cv2.Laplacian, which quantifies the intensity transitions in the images.

2. **Thresholding:** The code defines a gradient_threshold value (set to 100), which serves as a threshold for classifying the images. If the Laplacian variance of the blurred_image is below this threshold, it is labeled as a "Blurred Image." Otherwise, the original_image is classified as the "Original Image."

3. **Image Display:** The titles corresponding to the classifications are stored in blurred_title and original_title. These titles indicate which image is original and which is blurred. Both images are displayed using "cv2.imshow," allowing visual confirmation of the classifications.

4. **Error Handling:** Before processing, the code checks if the input images are successfully loaded. If either of the images cannot be loaded, an error message is printed.

1.3.2 Relevance with topics

The provided code distinguishes between an original grayscale image and a potentially blurred version using fundamental image processing concepts. It relies on Laplacian variance, measuring intensity transitions, as its core principle. This demonstrates the essence of image processing, focusing on feature extraction and analysis.

Intensity transformation and spatial filtering play a vital role, enhancing edges via the Laplacian operator. This showcases spatial filtering's ability to emphasize critical image features, facilitating attribute identification.

Thresholding implicitly applied through a predefined threshold (gradient_threshold), it aids in categorizing images as original or blurred, illustrating the importance of thresholding in image classification.

In terms of segmentation, the code categorizes images as "Original Image" or "Blurred Image." While not a direct segmentation technique, it implies a basic form of image categorization based on image characteristics, emphasizing segmentation's importance in image analysis.

1.4 Question 4

1.4.1 Proposed Solution

1. **Image Loading:** The function starts by reading the input image specified by the `image_path` parameter using OpenCV's `cv2.imread` function. The image is stored in the `image` variable.

2. **Image Dimensions:** The dimensions of the image (height, width, and number of color channels) are extracted using the `.shape` attribute of the `image` array.

3. **Bar Width Calculation:** The code calculates the width of each colored bar by dividing the total width of the image by 4. This assumes that there are four equally spaced bars in the image.

4. **Area Calculation:** For each of the four bars, the code iterates in a loop. It calculates the starting (`x_start`) and ending (`x_end`) pixel positions for each bar based on the calculated bar width. It then extracts the region of interest (ROI) for each bar.

5. **Pixel Counting:** The area of each bar is estimated by counting the number of non-zero pixels in the ROI using `np.count_nonzero`. This provides an approximation of the area covered by each colored bar.

6. **Centroid Calculation:** The code calculates the centroid of each bar, determining the x-coordinate as the midpoint between `x_start` and `x_end`, and setting the y-coordinate as the vertical center of the image (`height // 2`).

7. **Labeling:** Using `cv2.putText`, the calculated area of each bar is added as text near the centroid of the bar. This text includes the area in pixels squared (px^2).

8. **Display:** The modified image, with the labeled areas at the centroids, is displayed using `cv2.imshow`. The code waits for a key press (`cv2.waitKey(0)`) before closing the display window.

1.4.2 Relevance with topics

Fundamentals of Image Processing: The code demonstrates fundamental image processing concepts throughout. It loads an image, extracts its dimensions, and processes individual regions of interest (ROIs). These are fundamental operations in image analysis.

Intensity Transformation and Spatial Filtering: While the code doesn't explicitly perform intensity transformations or spatial filtering, it calculates the area of colored regions within the image. This involves assessing pixel intensity values within ROIs, which can be considered a form of spatial filtering in the context of extracting specific features or information from an image.

Histogram Processing: Histogram processing, which involves the analysis of pixel intensity distribution, is not explicitly used in this code. Instead, the code focuses on the direct measurement of areas of interest within the image.

Image Segmentation: Image segmentation, the process of dividing an image into meaningful regions, is indirectly applied. The code divides the image into four equally spaced regions, treating each colored bar as a segment. It then calculates the area of each segment based on the pixel count, effectively segmenting the image for analysis.

1.5 Question 5

1.5.1 Proposed Solution

1. **Image Loading:** The code loads the input image using OpenCV's `cv2.imread` function and stores it in the `image` variable.

2. **Color Conversion:** It converts the image from the default BGR format to RGB format using `cv2.cvtColor` for consistent color representation.

3. **Color Ranges:** The code defines lower and upper RGB color boundaries for four specific colors: yellow, light gray, gray, and dark gray.

4. **Percentage Area Calculation:** A function called `calculate_percentage_area` is defined to calculate the percentage area covered by each color. It uses `cv2.inRange` to create a binary mask for each color within the specified color range and calculates the area as a percentage of the total image pixels.

5. **Area Calculation:** The code calculates the percentage area covered by each color by calling the `calculate_percentage_area` function with the respective lower and upper color boundaries.

6. **Results Printing:** Finally, the code prints the results in the required format, displaying the percentage area covered by each color category.

1.5.2 Relevance with topics

Fundamentals of Image Processing: This concept is used throughout the code, as it involves loading an image, manipulating its colors, and performing calculations on pixel values to determine color areas.

Image Segmentation: Image segmentation is used indirectly in the code. Although the code does not perform explicit segmentation, it calculates the percentage area covered by different colors, which can be seen as a simple form of segmentation based on color regions.

1.6 Question 6

1.6.1 Proposed Solution

1. **Color Space Conversion:** It begins by converting the input image from BGR to the HSV color space using `cv2.cvtColor`. This conversion simplifies color-based segmentation.

2. **Color-Based Segmentation:** The code segments the image based on predefined HSV color ranges for each bone segment. This method isolates regions of interest by specifying the color characteristics of each bone segment.

3. **Binary Masking:** Using `cv2.inRange`, binary masks are created for each color segment. These masks highlight the pixels within the specified HSV range for each bone segment.

4. **Contour Detection:** Contours, representing connected components in the binary masks, are detected using `cv2.findContours`. This step identifies individual bone segments.

5. **Contour Filtering:** The code filters the detected contours based on area. Contours with an area less than 100 pixels are discarded, eliminating small noise segments.

6. **Bounding Box Analysis:** Using `cv2.boundingRect`, bounding boxes are calculated around valid contours. These boxes facilitate the visualization and measurement of segmented bone regions.

7. **Visualization:** The code draws green bounding boxes around each segmented bone region on the original image using `cv2.rectangle`, providing a clear visual representation.

8. **Region Extraction:** Extracted and segmented regions are stored as separate images in a list, enabling further analysis or processing of individual bone segments.

9. **Maximum Dimension Calculation:** The code calculates and prints the maximum width and height within each segmented region, offering insights into the dimensions of the bone segments.

1.6.2 Relevance with topics

The provided code predominantly employs image segmentation and the fundamentals of image processing to analyze and extract bone regions from an input image. It leverages segmentation by color, converting the image to the HSV color space and defining specific HSV color bounds for each bone segment. This approach effectively segments the image into distinct regions based on color information, crucial for distinguishing different bone regions.

The fundamentals of image processing are evident throughout the code, encompassing image loading, contour detection, bounding box calculation, and region extraction. These fundamental operations are essential for identifying and characterizing bone segments within the image. The code also incorporates contour filtering to remove small noise segments, ensuring that only significant bone regions are processed and visualized. The code primarily relies on image segmentation and fundamental processing operations to achieve its goal of bone region analysis.