DIP Project Report

Project Overview

The goal of this project is to develop an automated system capable of detecting and decoding multiple barcodes from a single image using advanced image processing techniques. The system is designed to process images captured using a mobile camera sensor, ensuring accessibility and convenience. It aims to support the detection of both 1D and 2D barcode formats, including commonly used standards such as EAN-13, Code-128, and QR codes.

A key feature of the system is its ability to operate in real-world conditions, making it robust against variations in barcode orientation. The system will be angle-invariant, enabling accurate detection and decoding regardless of the alignment or perspective of the barcodes in the image. Additionally, the design minimizes user interaction, providing a seamless and efficient user experience.

Existing Problems

The automated barcode detection and decoding system addresses several key challenges commonly faced in real-world barcode reading scenarios. It is designed to handle variations in barcode orientation, ensuring accurate detection regardless of the angle or perspective. The system is robust against environmental factors like motion blur, poor lighting, and occlusion, using advanced image processing techniques to decode barcodes even in suboptimal conditions. It supports the detection of multiple barcodes in a single image and is compatible with various 1D and 2D barcode formats, including EAN-13, Code-128, and QR codes. Additionally, the system is optimized for mobile devices, allowing users to capture images using their smartphones, thus enhancing accessibility and convenience. By minimizing the need for user interaction and automatically handling barcode detection and decoding, it provides a seamless, efficient user experience.

Algorithms

1. Barcode Validation Algorithm (`check(image)`)

The barcode validation algorithm checks whether a given image contains barcode-like structures based on features such as strong horizontal gradients and rectangular regions.

Algorithm 1 Barcode Validation Algorithm

Input: Image `image`
Output: Binary result indicating the presence of a barcode (1: True, 0: False)
Convert the image to grayscale: `gray` ← `cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)`.
Compute horizontal gradient: `gradX` ← `cv2.Sobel(gray, ddepth, dx=1, dy=0)`.
Compute vertical gradient: `gradY` ← `cv2.Sobel(gray, ddepth, dx=0, dy=1)`.
Calculate the gradient difference: `gradient` ← `cv2.subtract(gradX, gradY)`.
Convert gradient to absolute values: `gradient` ← `cv2.convertScaleAbs(gradient)`.
Blur the gradient image: `blurred` ← `cv2.blur(gradient, (9, 9))`.
Apply binary threshold: `thresh` ← `cv2.threshold(blurred, 225, 255, cv2.THRESH_BINARY)`.
Define a rectangular kernel: `kernel` ← `cv2.getStructuringElement(cv2.MORPH_RECT, (21, 7))`.
Apply morphological closing: `closed` ← `cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)`.
Perform erosion and dilation:

- `closed` ← `cv2.erode(closed, None, iterations=4)`.
- `closed` ← `cv2.dilate(closed, None, iterations=4)`.

Find contours: `cnts` ← `cv2.findContours(closed)`.
if `len(cnts) == 0` **then**
    return 0.
end if
Select the largest contour: `c` ← `max(cnts, key=cv2.contourArea)`.
Compute minimum bounding rectangle: `rect` ← `cv2.minAreaRect(c)`.
Visualize the bounding box: `box` ← `cv2.boxPoints(rect)`.
return 1.

2. Image Rotation and Horizontal Alignment Algorithm (`check_horizontal(img)`)

This algorithm aligns the input image horizontally by detecting its inclination angle using Hough Line Transformation and rotating it accordingly.

Algorithm 2 Horizontal Alignment Algorithm

Input: Image `img`
Output: Horizontally aligned image `rotated_img`
Convert the image to grayscale: `gray` ← `cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)`.
Apply Canny edge detection: `edges` ← `cv2.Canny(gray, 50, 150, apertureSize=3)`.
Detect lines using Hough Line Transformation:

- `lines` ← `cv2.HoughLines(edges, 1, π/180, 300)`.

if `lines` is not `None` **then**
    Initialize an empty list for angles: `angles`. `(, )` in `lines[:, 0]`
    Compute endpoints of the line:

- x_1, y_1, x_2, y_2 based on (ρ, θ).

    Plot the detected line.
    Append θ (converted to degrees) to `angles`.

    Compute the mean angle: `mean_angle` ← `np.mean(angles)`.
    if $|$`mean_angle` $- 90| < 5$ **or** $|$`mean_angle` $- 270| < 5$ **then**
        return the input image (already aligned).
    else
        Calculate the rotation angle: `angle` ← `mean_angle + 180`.
        Compute rotation matrix: `M` ← `cv2.getRotationMatrix2D(center, angle, 1.0)`.
        Apply rotation: `rotated_img` ← `cv2.warpAffine(img, M, (w, h))`.
    end if
else
    No lines detected; `rotated_img` ← `img`.
end if
return `rotated_img`.

3. Segment Extraction Algorithm (`extract_segments(image_path)`)

This algorithm processes an input image to extract valid barcode segments using a segmentation model and additional validation steps.

Algorithm 3 Segment Extraction Algorithm

```
Input: Path to input image image_path
Output: List of valid segments with metadata
Load the image: original_image ← Image.open(image_path).
Convert image to NumPy array: image_np ← np.array(original_image).
Initialize segmentation model: model ← FastSAM('FastSAM-s.pt').
Run segmentation: results ← model(original_image, retina_masks=True, conf=0.4, iou=0.9).
if results[0].masks.data is None then
    Print "No segments detected".
    return [].
end if
Initialize list for valid segments: segments ← []. annotation in results[0].masks.data
Resize mask to match the image size.
Extract the segment using the mask.
Calculate bounding box dimensions.
Compute bounding box area.
if bbox_area is not within size constraints then
    Skip the segment.
end if
if check(cropped_segment) == 0 then
    Skip the segment.
end if
Save the valid segment to disk and append metadata to segments.

return segments.
```

Check for QR Code in an Image (`check_qr_code`)

This algorithm identifies potential QR codes in an image by detecting contours and approximating them as polygons with four corners. If enough four-cornered contours are found, the image is considered to contain QR codes.

Algorithm 4 Check for QR Code in an Image

```
Input: Image image
Output: True if a QR code is detected, False otherwise.
Convert the image to grayscale: gray ← cv2.cvtColor(image, cv2.COLOR_BGR2GRAY).
Apply Gaussian blur to the grayscale image: blurred ← cv2.GaussianBlur(gray, (5, 5), 0).
Apply Canny edge detection: edges ← cv2.Canny(blurred, 50, 150).
Find contours in the edges: contours ← cv2.findContours(edges, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE).
Initialize an empty list for potential QR codes: potential_qr ← [].
for each contour in contours do
    Calculate the contour length: epsilon ← 0.02 * cv2.arcLength(cnt, True).
    Approximate the contour to a polygon: approx ← cv2.approxPolyDP(cnt, epsilon, True).
    if the polygon has 4 vertices then
        Add the polygon to potential_qr.
    end if
end for
if the number of potential QR codes is greater than or equal to 3 then
    return True.
else
    return False.
end if
```

The pipeline for automated barcode detection and decoding consists of several key stages, each utilizing different algorithms for image processing and barcode recognition. The stages are as follows:

1. **Image Preprocessing:** In this initial stage, the input image undergoes basic preprocessing steps, such as grayscale conversion and edge detection, to improve the clarity and contrast of the barcodes. This step helps highlight the barcode features while reducing noise that could interfere with detection.

2. **Barcode Localization:** This step focuses on identifying regions of the image where barcodes are likely to be located. It involves contour detection, morphological operations like closing and erosion, and other techniques to detect the rectangular shapes of barcodes. The algorithm looks for shapes resembling barcodes, which are typically rectangular and have certain aspect ratios.

3. **Barcode Orientation Handling:** After localization, the system determines the orientation of each detected barcode. If the barcode is not aligned properly (e.g., rotated), the system applies geometric transformations (like rotation) to correct the orientation. This ensures that the barcode can be decoded regardless of its angle in the image.

4. **Barcode Decoding:** The final step involves decoding the detected barcodes. For 1D and 2D barcodes like QR codes, specialized algorithms detect and decode QR patterns and barcodes, extracting the embedded data.

Results

Figure 1: Original Image

Figure 2: Rotated - Segmented Image, followed by result

Figure 3: Original Image

Figure 4: Rotated - Segmented Image, followed by result

Figure 5: Original Image

Figure 6: Rotated - Segmented Image, followed by result

Advantages of the Proposed Method

The proposed method, which combines angle invariance and advanced segmentation models, offers several advantages over traditional approaches for barcode detection and decoding:

Angle Invariance

- **Existing Issues:** Traditional barcode scanners often require manual alignment of the barcode, as they perform poorly when the barcode is rotated or skewed.
- **Improvement:** By incorporating techniques like Hough Transform for angle detection and rectification, the proposed method ensures that barcodes are correctly oriented before decoding. This makes the system robust against real-world conditions where barcodes are rarely perfectly aligned.

Multi-Format Support

- **Existing Issues:** Older systems may be optimized for specific barcode formats (e.g., 1D or QR codes), limiting their versatility.
- **Improvement:** The pipeline is format-agnostic, capable of detecting and decoding both 1D and 2D barcodes (e.g., EAN-13, Code-128, QR codes).

Scalability

- **Existing Issues:** Traditional systems may struggle to handle multiple barcodes in a single image or require significant user input to select a region of interest.
- **Improvement:** The segmentation model enables automated detection of multiple barcode regions within the image, facilitating batch processing and reducing the need for manual intervention.

Real-World Usability

- **Existing Issues:** Standard barcode readers are often sensitive to variations in lighting, perspective distortions, and background interference.
- **Improvement:** The method integrates image preprocessing techniques like Sobel gradients, Gaussian blur, and Canny edge detection to mitigate lighting and noise issues, improving robustness under diverse conditions.

Automation and User Experience

- **Existing Issues:** Traditional methods often require users to position and align barcodes carefully, increasing the time and effort required for decoding.
- **Improvement:** This pipeline automates the process of rotation correction, segmentation, and decoding, delivering a seamless user experience with minimal manual input.

Conclusion

The proposed pipeline improves robustness, accuracy, and usability compared to conventional barcode detection methods. By addressing challenges such as orientation invariance, background clutter, and multi-format support, it provides a versatile solution for real-world scenarios.