Lab2: Edge Detection

Wu Han
Class: SEIEE 2301

December 3, 2024

1 Experiment Overview

This experiment explores edge detection techniques. An edge in an image refers to regions where there is a significant change in pixel intensity. Such regions can often be modeled as a step-function, where pixel values exhibit a sharp change over a small spatial area.

The purpose of edge detection is to identify these sharp transitions in intensity. By approximating the gradient of the image's grayscale matrix, we can locate the edges, and by connecting these points, we form the outline of the image's edges.

The Canny edge detection algorithm is one of the most effective methods for edge detection. It involves five main steps:

1. Smoothing the image with a Gaussian filter to remove noise.
2. Calculating the gradient strength and direction for each pixel.
3. Applying Non-Maximum Suppression (NMS) to remove spurious responses from edge detection.
4. Using Double Thresholding to classify pixels as real or potential edges.
5. Finalizing the edge detection by suppressing weak, isolated edges.

In OpenCV, there is a pre-built Canny function that performs edge detection directly. However, in this experiment, we aim to implement a similar function from scratch for learning purposes.

2 Problem Statement, Experiment Principles, and Code Explanation

In this experiment, we were tasked with implementing the Canny edge detection function. To simplify the process, we encapsulated the function into a python function called `Custom edge detection`, which organizes the workflow and makes it easier to conduct further experiments. The implementation is divided into five key tasks:

2.1 Grayscale Conversion

The principle of converting images to grayscale was covered in Experiment 1. For this experiment, we employed the standard method of grayscale conversion, which is based on the following weighted sum of RGB components:

$$\text{Gray} = 0.299R + 0.587G + 0.114B$$

This method is widely used in image processing for converting color images to grayscale.The code is shown as follows.

```
# Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

This function converts the image from BGR (Blue-Green-Red) to grayscale, making it easier to compute gradients and apply filters.

2.2 Gaussian Filtering

Gaussian filtering is an important step to smooth the image and remove noise before edge detection. It can be done by applying two 1D Gaussian kernels sequentially, or by applying a 2D Gaussian kernel in one convolution. The 2D Gaussian function is defined as:

$$H(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

After normalization, this kernel can be convolved with the image to reduce noise. For example, with a value of $\sigma = 1.4$, the kernel values are calculated and applied to the image.The code is shown as follows.

```
# Apply Gaussian Blur
    blurred = cv2.GaussianBlur(gray, (3, 3), 0)
```

Here, $(3, 3)$ defines the size of the Gaussian kernel, and 0 specifies that the standard deviation of the kernel is automatically calculated.

2.3 Gradient Calculation Using Sobel Operator And Other Operators

To detect edges, we need to calculate the gradient of the image. This is done using finite differences to approximate the first-order derivatives in the x and y directions. The Sobel operator is commonly used for this purpose. The Sobel kernels in the x and y directions are:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

By applying these kernels to the grayscale image, we calculate the gradients in both directions. The magnitude of the gradient is given by:

$$G(x, y) = \sqrt{g_x^2 + g_y^2}$$

The direction of the gradient is calculated as:

$$\theta = \tan^{-1}\left(\frac{g_y}{g_x}\right)$$

These values help identify the edges by highlighting areas with significant intensity changes.The key code is as follows:

```
    if operator == 'sobel':
        # Compute gradients using Sobel operator
        grad_x = cv2.Sobel(blurred, cv2.CV_16S, 1, 0)
        grad_y = cv2.Sobel(blurred, cv2.CV_16S, 0, 1)
```

This calculates the gradients in the horizontal and vertical directions, respectively.

Other operators like Roberts Operator will also be used for comparison.Here are the details of them.

1) Roberts Operator

The Roberts operator is one of the simplest methods to approximate the gradient of an image. It uses two 2×2 convolution kernels to compute the gradients in the x and y directions. The gradient magnitude G at each point can be computed as follows:

$$G = \sqrt{(G_x)^2 + (G_y)^2} \tag{1}$$

where G_x and G_y are computed using the following convolution kernels:

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{2}$$

The code to implement this function is as follows

```
    elif operator == 'roberts':
        # Define Roberts operator kernels
        kernel_x = np.array([[1, 0], [0, -1]],
           dtype=int)
        kernel_y = np.array([[0, 1], [-1, 0]],
           dtype=int)

        # Apply the kernels to get gradients
        grad_x = cv2.filter2D(blurred, cv2.CV_16S,
           kernel_x)
        grad_y = cv2.filter2D(blurred, cv2.CV_16S,
           kernel_y)
```

The gradients are calculated by applying these kernels using the 'cv2.filter2D' function.

2) Prewitt Operator

The Prewitt operator is similar to the Sobel operator, but it uses different convolution kernels. The Prewitt operator also calculates the gradient in the x and y directions, and the gradient magnitude G is given by:

$$G = \sqrt{(G_x)^2 + (G_y)^2} \tag{3}$$

The convolution kernels for the Prewitt operator are as follows:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{4}$$

The code to implement this function is as follows

```
    elif operator == 'prewitt':
        # Define Prewitt operator kernels
        kernel_x = np.array([[-1, 0, 1], [-1, 0, 1],
           [-1, 0, 1]], dtype=int)
        kernel_y = np.array([[1, 1, 1], [0, 0, 0], [-1,
           -1, -1]], dtype=int)

        # Apply the kernels to get gradients
        grad_x = cv2.filter2D(blurred, cv2.CV_16S,
           kernel_x)
```

```
        grad_y = cv2.filter2D(blurred, cv2.CV_16S,
            kernel_y)
```

These kernels are also applied using ‘cv2.filter2D‘ to compute the gradients in the x and y directions.

2.4 Non-Maximum Suppression (NMS)

Non-Maximum Suppression is a technique used to thin the detected edges. In this step, we compare the gradient magnitude at each pixel with the magnitudes along the direction of the gradient. If a pixel is not a local maximum in the gradient direction, it is suppressed (set to zero). This helps to eliminate spurious edges and ensures that only the strongest edges are retained.The code to implement this function is as follows

```
    # Convert gradients to absolute values and then to
        uint8
    abs_grad_x = cv2.convertScaleAbs(grad_x)
    abs_grad_y = cv2.convertScaleAbs(grad_y)

    # Combine the gradients to get the edge magnitude
        and direction
    gradient_magnitude = cv2.addWeighted(abs_grad_x,
        0.5, abs_grad_y, 0.5, 0)
    gradient_direction = np.arctan2(grad_y, grad_x) *
        (180 / np.pi)
    gradient_direction[gradient_direction < 0] += 180
```

Here, ‘cv2.convertScaleAbs‘ is used to convert the gradients to absolute values and scale them to an 8-bit unsigned integer. Then, the gradient magnitude is computed as a weighted sum of ‘abs grad x‘ and ‘abs grad y‘, with equal weights of 0.5. The gradient direction is computed using the ‘np.arctan2‘ function, which calculates the arctangent of the gradient's y and x components.

Next, the Non-Maximum Suppression (NMS) is applied:

```
    # Non-Maximum Suppression with Interpolation
    nms_result = np.zeros_like(gradient_magnitude,
        dtype=np.uint8)
    rows, cols = gradient_magnitude.shape

    for r in range(1, rows - 1):
        for c in range(1, cols - 1):
            angle = gradient_direction[r, c]
            mag = gradient_magnitude[r, c]
```

```
9
10              # Determine the neighboring pixels to
                   interpolate based on the gradient
                   direction
11              if (0 <= angle < 45):
12                  weight = math.tan(math.radians(angle))
13                  neighbor_1 = (1 - weight) *
                       gradient_magnitude[r + 1, c] +
                       weight * gradient_magnitude[r + 1, c
                       + 1]
14                  neighbor_2 = (1 - weight) *
                       gradient_magnitude[r - 1, c] +
                       weight * gradient_magnitude[r - 1, c
                       - 1]
15              elif 45 <= angle < 90:
16                  weight = 1 /
                       math.tan(math.radians(angle))
17                  neighbor_1 = (1 - weight) *
                       gradient_magnitude[r, c + 1] +
                       weight * gradient_magnitude[r + 1, c
                       + 1]
18                  neighbor_2 = (1 - weight) *
                       gradient_magnitude[r, c - 1] +
                       weight * gradient_magnitude[r - 1, c
                       - 1]
19              elif 90 <= angle < 135:
20                  weight = math.tan(math.radians(angle -
                       90))
21                  neighbor_1 = (1 - weight) *
                       gradient_magnitude[r, c + 1] +
                       weight * gradient_magnitude[r - 1, c
                       + 1]
22                  neighbor_2 = (1 - weight) *
                       gradient_magnitude[r, c - 1] +
                       weight * gradient_magnitude[r + 1, c
                       - 1]
23              elif 135 <= angle < 180:
24                  weight = math.tan(math.radians(180 -
                       angle))
25                  neighbor_1 = (1 - weight) *
                       gradient_magnitude[r - 1, c] +
                       weight * gradient_magnitude[r - 1, c
                       + 1]
```

```
                neighbor_2 = (1 - weight) *
                    gradient_magnitude[r + 1, c] +
                    weight * gradient_magnitude[r + 1, c
                    - 1]

            # Suppress non-maximum values using
                interpolation
            if mag >= neighbor_1 and mag >= neighbor_2:
                nms_result[r, c] = mag
            else:
                nms_result[r, c] = 0
```

The code loops through each pixel and checks its gradient direction:

```
for r in range(1, rows - 1):
for c in range(1, cols - 1):
```

For each pixel, the angle θ is used to determine which two neighboring pixels to compare. Depending on the angle, the code uses interpolation to compute the weights of the neighboring pixels.We have to use 'tan' function to determine the weight and calculate the gradient of dTmp1 and dTmp2(we call these two as "neighbors"). The gradients at these neighbors are compared to the current pixel's gradient magnitude.

If $G_{\text{pixel}} \geq \text{dTmp1}$ and $G_{\text{pixel}} \geq \text{dTmp2}$, then keep the pixel.

Otherwise, suppress it.

2.5 Double Thresholding and Edge Connectivity

The final step in Canny edge detection is double thresholding, which helps classify pixels as either strong edges, weak edges, or non-edges. Two thresholds are applied: a high threshold (TH) and a low threshold (TL). If a pixel's gradient is above the high threshold, it is considered a strong edge. If it is below the low threshold, it is discarded. If the gradient lies between the two thresholds, the pixel is considered a weak edge, and its status is determined by examining the surrounding pixels.

Once the edges have been detected, we perform edge connectivity to ensure that weak edges are connected to strong edges, forming continuous boundaries.The main code is as follows:

```
    # Apply double threshold to get edges
    _, strong_edges = cv2.threshold(nms_result,
       high_threshold, 255, cv2.THRESH_BINARY)
    _, weak_edges = cv2.threshold(nms_result,
       low_threshold, 255, cv2.THRESH_BINARY)

    # Combine strong and weak edges
    edges = cv2.bitwise_or(strong_edges, weak_edges)
```

In this step, pixels are classified into strong and weak edges using two threshold values: 'high threshold' and 'low threshold'.

The final edge detection is completed by combining the strong and weak edges.

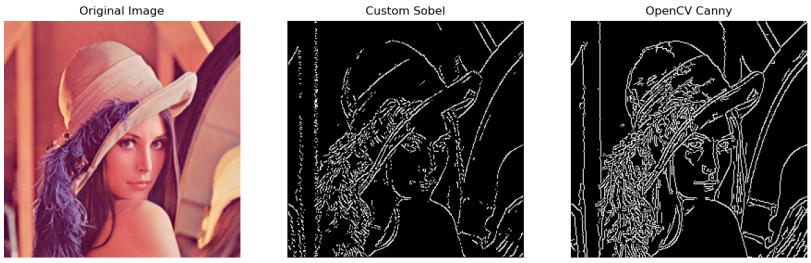
3 Results and Discussion

3.1 Sobel Operator

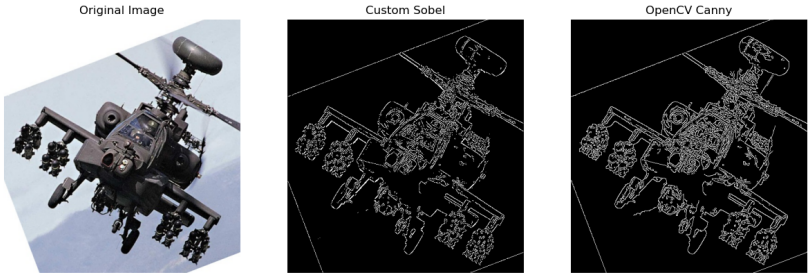
The edge detection results produced by our Canny implementation were compared with those obtained using the OpenCV Canny function.As it's shown in Figure 1. While the results were not identical, the overall structure and contours of the detected edges were similar. Our implementation was able to detect the general outlines, but with some noise and less sharpness compared to the built-in OpenCV function.

3.2 Other Operators Comparison

The edge detection results obtained using different operators were compared to evaluate their performance. As it's shown in Figure2, The built-in OpenCV Canny function produced the most accurate and precise edges, with well-defined contours and minimal noise. The Sobel operator also performed quite well, capturing the overall edge structure with reasonable sharpness, though it slightly lagged behind the Canny in terms of precision. The Prewitt operator produced somewhat less defined edges, with noticeable noise and slightly blurred contours. Finally, the Roberts operator yielded the least satisfactory results, as it struggled to detect fine edges and produced more noise, making the overall edge detection less reliable compared to the other methods.



(a)



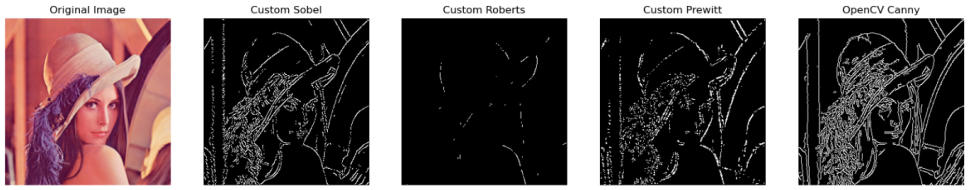
(b)

(c)

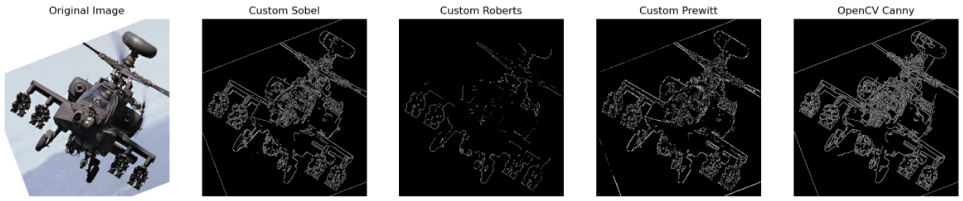
Figure 1: Sobel Operation Results

3.3 Different Threshold Comparison

The edge detection results obtained with different double threshold values were compared to assess their impact on the quality of the edges detected.As it's shown in Figure3, The combination of (40, 100) produced the best results, with clear and well-defined edges and minimal noise. The (50, 150) combination also yielded good results, though the edges were slightly less sharp and



(a)



(b)

(c)

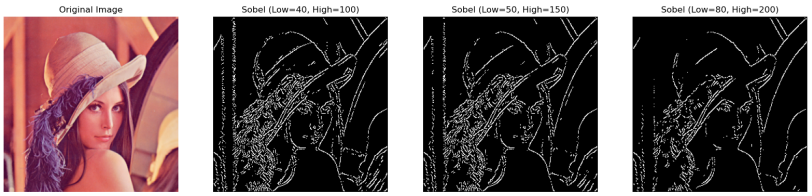
Figure 2: Different Operations Comparsion

there was a slight increase in noise compared to the (40, 100) combination. Finally, the (80, 200) threshold combination gave the least favorable results, with more pronounced noise, less distinct edges, and a considerable loss in edge detail.

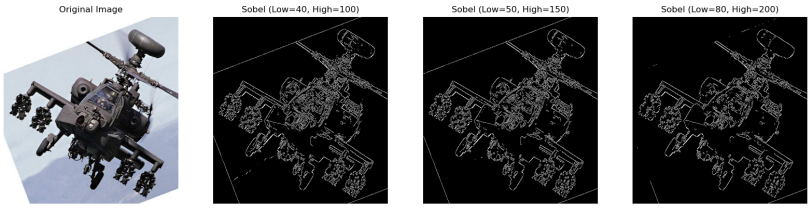
4 Further Analysis and Extensions

4.1 The Role of Gaussian Filtering

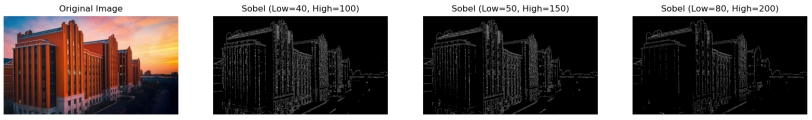
Gaussian filtering plays a critical role in removing noise and smoothing sharp transitions in the image. Without this step, the edge detection results would be significantly affected by noise, leading to distorted or incorrect edge maps.



(a)



(b)



(c)

Figure 3: Different Threshold Comparison

4.2 The Principle of Double Thresholding

Double thresholding helps to minimize false edges by classifying pixels based on their gradient magnitudes. This method ensures that only meaningful edges are preserved, and weak edges are either discarded or connected to strong edges to form continuous contours.

4.3 An Error in PPT

In section "Non-Maximum Suppression (NMS)", the ppt provides a formula:M(dTmp1) = w*M(g2) + (1-w)*M(g1), w = dis(dtmp, g2)/dis(g1, g2) However, from my perspective, the right form is M(dTmp1) = w*M(g1) + (1-w)*M(g2), w = dis(dTmp1, g2)/dis(g1, g2)

5 Personal Reflection

This experiment has been a great opportunity to improve my skills in several areas. First, my understanding of mathematics, especially in the context of image processing, has grown significantly. Implementing edge detection algorithms like Sobel and Prewitt helped me see how mathematical operations can be directly applied to solve real-world problems. My understanding of OpenCV has also deepened—working with its functions for gradient computation, filtering, and edge detection gave me a better appreciation of its capabilities. In terms of coding, I became more efficient at writing clear and structured code, breaking down complex tasks into manageable functions. Finally, the experiment enhanced my writing skills, as I had to explain technical concepts clearly and concisely. Overall, this experience has not only improved my technical abilities but also boosted my communication skills.