Canny Edge Detector Algorithm in Image Segmentation

Alp Tuna - 2019400288
Oğuzhan Demirel - 2018400165

June 6, 2023

1 Introduction

1.1 What is Image Segmentation?

Image segmentation is the process of dividing a digital image into distinct regions or segments based on certain characteristics or criteria. The aim of segmentation is to group pixels or regions within an image that share similar properties, such as color, texture, intensity, or spatial proximity. By partitioning the image into segments, it becomes easier to analyze and understand its contents at a more granular level.

In image segmentation, each pixel in the image is assigned a label or category, indicating which segment it belongs to. This allows for the identification and differentiation of objects, boundaries, and other meaningful elements within the image. The segmentation can be binary, where pixels are classified as either foreground or background, or it can involve multiple classes, distinguishing between various objects or regions of interest.

Image segmentation plays a crucial role in computer vision, as it serves as a fundamental step for many tasks and applications. It is widely used in fields such as object recognition, tracking, medical imaging, autonomous driving, image editing, and scene understanding. By segmenting an image, it becomes possible to extract valuable information from specific regions and facilitate subsequent analysis, interpretation, or manipulation of the visual data.

1.2 Edge Detection

Edge detection is a fundamental process in computer vision that aims to identify and locate the boundaries or edges of objects within an image. It involves detecting significant changes in intensity or color between adjacent pixels, which typically correspond to object boundaries or transitions between different regions.

In the context of image segmentation, edge detection is often used as a preprocessing step or as a feature extraction technique. It helps to highlight the boundaries between different objects or regions, which can then be further processed using segmentation algorithms to partition the image into distinct segments.

Once edges are detected, image segmentation methods can utilize this edge information along with other features, such as color, texture, or spatial proximity, to separate the image into meaningful regions or segments. The edges often serve as boundaries or cues for region-growing, region-merging, or other region-based segmentation techniques.

1.3 Common Edge Detection Algorithms

There are several common edge detection algorithms used in computer vision and image processing. Here are some of the widely used edge detection methods:

1. Canny Edge Detector: (In the next section, we are going to discuss this algorithm in detail.)
2. Sobel Operator: The Sobel operator is a gradient-based method that calculates the gradient magnitude at each pixel in the image. It uses a set of convolution kernels to approximate the derivatives in the horizontal and vertical directions. The Sobel operator is relatively simple and computationally efficient, making it commonly used in edge detection tasks.
3. Laplacian of Gaussian (LoG): The Laplacian of Gaussian operator applies the Laplacian operator to an image that has been convolved with a Gaussian kernel. The LoG operator detects edges by finding zero crossings in the second derivative of the image intensity. It is sensitive to edges at different scales and can help detect edges with varying levels of detail.
4. Gradient-based Methods: Besides the Sobel operator, other gradient-based methods, such as Prewitt, Roberts, and Scharr operators, can be

used for edge detection. These methods calculate the gradients in the horizontal and vertical directions and combine them to estimate edge strength and orientation.

You can refer to more detailed comparison of the algorithms in the links given below:

- https://medium.com/@nikatsanka/comparing-edge-detection-methods-638a2919476e
- https://anirban-karchaudhuri.medium.com/edge-detection-methods-comparison-9e4b75a9bf87

2 Canny Edge Detector Algorithm

The Canny edge detector is a popular algorithm known for its accuracy and robustness. It is computationally more expensive compared to Sobel, Laplacian operator and many other edge detection algorithms. However, the Canny's edge detection algorithm performs better than all these operators under almost all scenarios, even under noisy conditions. It involves multiple stages, including Gaussian smoothing to reduce noise, calculating gradient magnitude and direction, non-maximum suppression to thin out edges, and hysteresis thresholding to detect and link edges.

2.1 Steps Of the Algorithm

2.1.1 Step 1: Convert the image to grayscale

The algorithm starts by converting the input image from color to grayscale. This simplifies the edge detection process as we only need to consider changes in intensity rather than color.

```
grayImage = rgb2gray(image);
```

2.1.2 Step 2: Apply Gaussian smoothing

To reduce noise and prevent false detections, a Gaussian filter is applied to the grayscale image. The smoothing operation helps to blur the image slightly and suppress small intensity variations that may not correspond to actual edges.

```
filteredImage = imgaussfilt(grayImage, sigma);
```

2.1.3 Step 3: Compute gradients

Using Sobel operators, we calculate the horizontal and vertical gradients of the smoothed image. These gradients represent the rate of change of intensity in the x and y directions, providing information about the image's edges' orientations and strengths.

```
    sobelX = [-1 0 1; -2 0 2; -1 0 1];
    sobelY = [-1 -2 -1; 0 0 0; 1 2 1];
    gradientX = imfilter(filteredImage, sobelX);
    gradientY = imfilter(filteredImage, sobelY);
    gradientX = double(gradientX);
    gradientY = double(gradientY);
```

2.1.4 Step 4: Compute gradient magnitude

The gradient magnitude is computed by combining the horizontal and vertical gradients using the Euclidean distance formula. This represents the overall strength of the edges at each pixel, combining information from both gradient directions.

```
    gradientMagnitude = hypot(gradientX, gradientY);
    gradientOrientation = atan2(gradientY, gradientX);
```

2.1.5 Step 5: Perform non-maximum suppression

This step helps refine the detected edges by thinning them to single-pixel width. It involves comparing the gradient magnitude with its neighboring pixels along the gradient direction and keeping only the local maxima. This ensures that only the most significant edges are preserved while suppressing weaker or spurious responses.

```

function suppressedImage = nonMaxSuppression(gradientMagnitude,
    gradientOrientation)
    [rows, cols] = size(gradientMagnitude);
    suppressedImage = zeros(rows, cols);

    % Convert orientation values to angles between 0 and 180
    degrees
    angle = rad2deg(gradientOrientation);
    angle(angle < 0) = angle(angle < 0) + 180;

    % Perform non-maximum suppression
    for i = 2:rows-1
        for j = 2:cols-1
            q = 255;
            r = 255;

            % Find the appropriate neighboring pixels based on the
    orientation
```

```
            if (0 <= angle(i,j) && angle(i,j) < 22.5) || (157.5 <= angle(i,j) && angle(i,j) <= 180)
                q = gradientMagnitude(i, j + 1);
                r = gradientMagnitude(i, j - 1);
            elseif 22.5 <= angle(i,j) && angle(i,j) < 67.5
                q = gradientMagnitude(i + 1, j - 1);
                r = gradientMagnitude(i - 1, j + 1);
            elseif 67.5 <= angle(i,j) && angle(i,j) < 112.5
                q = gradientMagnitude(i + 1, j);
                r = gradientMagnitude(i - 1, j);
            elseif 112.5 <= angle(i,j) && angle(i,j) < 157.5
                q = gradientMagnitude(i - 1, j - 1);
                r = gradientMagnitude(i + 1, j + 1);
            end

            % Perform suppression
            if gradientMagnitude(i,j) >= q && gradientMagnitude(i,j) >= r
                suppressedImage(i,j) = gradientMagnitude(i,j);
            else
                suppressedImage(i,j) = 0;
            end
        end
    end
end
```

2.1.6 Step 6: Double thresholding

A double thresholding technique is applied to classify the remaining edge candidates as strong, weak, or non-edges based on their gradient magnitude values. Pixels with magnitudes above a high threshold are considered strong edges, while those below a low threshold are considered non-edges. Pixels between the two thresholds are marked as weak edges and will undergo further analysis in the next step.

```

function thresholdedImage = doubleThresholding(suppressedImage, lowThreshold, highThreshold)
    [rows, cols] = size(suppressedImage);
    thresholdedImage = zeros(rows, cols);

    % Perform thresholding
    thresholdedImage(suppressedImage >= highThreshold) = 1;
    figure;
    imshow(thresholdedImage);

    % Apply hysteresis thresholding
    visited = zeros(rows, cols);
    stack = [];

    % Find strong edges and start the edge tracking process
    [strongRows, strongCols] = find(suppressedImage >= highThreshold);
    numStrong = numel(strongRows);

```

```
    for i = 1:numStrong
        if visited(strongRows(i), strongCols(i)) == 0
            stack = [stack; strongRows(i), strongCols(i)];
            visited(strongRows(i), strongCols(i)) = 1;
        end
    end

    % Perform edge tracking
    while ~isempty(stack)
        currentPixel = stack(1, :);
        stack(1, :) = [];
        neighbors = getNeighbors(currentPixel(1), currentPixel(2),
    rows, cols);

        for k = 1:size(neighbors, 1)
            row = neighbors(k, 1);
            col = neighbors(k, 2);

            if visited(row, col) == 0 && suppressedImage(row, col)
    >= lowThreshold
                stack = [stack; row, col];
                visited(row, col) = 1;
                thresholdedImage(row, col) = 1;
            end
        end
    end
end
```

2.1.7 Step 7: Edge tracking by hysteresis

This final step aims to connect weak edges to strong edges and suppress remaining non-edges. Starting from each strong edge, a connectivity search is performed to link adjacent weak edges that are likely part of the same edge structure. This process is often performed using depth-first search or similar techniques, ensuring that connected edges are retained while non-edges are suppressed.

```

function edgeImage = edgeTrackingByHysteresis(thresholdedImage)
    [rows, cols] = size(thresholdedImage);
    edgeImage = zeros(rows, cols);


    % Perform edge tracking by hysteresis
    for i = 2:rows-1
        for j = 2:cols-1
            if thresholdedImage(i, j) == 1 && edgeImage(i, j) == 0
                edgeImage(i, j) = 1;

                % Perform depth-first search to find connected
    edges
                stack = [i, j];
                while ~isempty(stack)
                    currentPixel = stack(1, :);
                    stack(1, :) = [];
```

```
18
19                      % Get neighbors of current pixel
20                      neighbors = getNeighbors(currentPixel(1),
    currentPixel(2), rows, cols);
21
22                      for k = 1:size(neighbors, 1)
23                          row = neighbors(k, 1);
24                          col = neighbors(k, 2);
25
26                          % Check if neighbor is a candidate edge
27                          if thresholdedImage(row, col) == 1 &&
    edgeImage(row, col) == 0
28                              edgeImage(row, col) = 1;
29                              stack = [stack; row, col];
30                          end
31                      end
32                  end
33              end
34          end
35      end
36  end
```

2.2 Inputs of the Algorithm

2.2.1 Sigma (sigma)

Sigma is the standard deviation of the Gaussian filter applied in the image smoothing step. Gaussian smoothing helps to reduce noise in the image and improve the quality of edge detection. A higher value of sigma results in a stronger smoothing effect, which can help to remove more noise but may also blur the edges. The appropriate value of sigma depends on the characteristics of the image and the level of noise present. Experimentation and fine-tuning are often necessary to determine the optimal sigma value.

2.2.2 Low Threshold: lowThreshold

Low threshold is used in the double thresholding step of the Canny edge detector. After non-maximum suppression, the gradient magnitude values are classified as strong, weak, or non-edges based on their relationship to the high and low thresholds. The low threshold determines the lower bound for weak edges. Any gradient magnitude value below the low threshold is considered a non-edge and suppressed. Weak edges are those above the low threshold but below the high threshold. The low threshold helps to distinguish weaker edges from noise or non-edges.

2.2.3 High Threshold (highThreshold)

High threshold is the upper bound for strong edges in the double thresholding step. Gradient magnitude values above the high threshold are classified as strong edges, while those below the high threshold but above the low threshold are considered weak edges. Strong edges are typically associated with pronounced transitions or sharp changes in intensity, while weak edges represent less significant or uncertain edge information. The high threshold helps to define the intensity threshold above which an edge is considered strong and reliable.

3 Results

Input1

Output 1

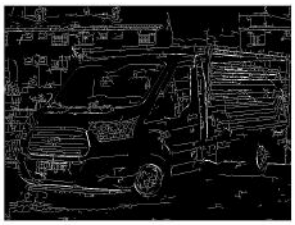
Input 2

Output 2

Input 3

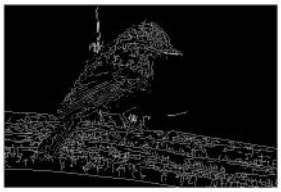
Output 3

Input 4



Output 4

Input 5



Output 5

4 Conclusion

The implementation of the Canny edge detection algorithm in this project has yielded promising results. The algorithm effectively detects and highlights edges in images, providing clean edge maps by incorporating Gaussian smoothing and non-maximum suppression. The inclusion of weak edges through double thresholding and edge tracking enhances the comprehensiveness of the results.

However, there are a few shortcomings that should be acknowledged. One

limitation of the Canny edge detector is its sensitivity to parameter selection. The values of parameters such as sigma, low threshold, and high threshold can significantly impact the quality of the edge detection results. Fine-tuning these parameters for each image and application is essential to achieve optimal results.

Additionally, the Canny edge detector may struggle in cases where edges are ambiguous or faint, or when there is a high level of noise in the image. In such situations, additional preprocessing steps or alternative edge detection methods may be necessary to improve the results.

Overall, the Canny edge detection project has provided valuable insights into the algorithm's principles and implementation. Understanding its strengths and weaknesses allows for informed parameter selection and lays the foundation for future advancements in edge detection.

5 References

- https://datagen.tech/guides/image-annotation/image-segmentation/
- https://anirban-karchaudhuri.medium.com/edge-detection-methods-comparison-9e4b75a9bf87
- https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vi
- https://en.wikipedia.org/wiki/Canny_edge_detector