Hood College

CS 428: Artificial Intelligence

Image Processing Filters Explained

(Code is in Your Program)

Instructor:
Prof. Qian

Date:
November 6, 2024

Contents

Chapter 1

Introduction

This document provides detailed explanations of various image processing filters using OpenCV. Each filter is discussed in terms of its purpose, how it works, and how to adjust its parameters effectively. The variable `roi` is used as the source image in all examples.

Chapter 2

Image Filters

2.1 Converting to Grayscale

```
roi = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
```

What It Does

- **Purpose**: Converts a color image from the BGR (Blue, Green, Red) color space to a grayscale image.
- **Result**: The output `roi` is a single-channel image where each pixel represents the intensity of light (brightness), ranging from black to white.

Why Convert to Grayscale?

- **Simplification**: Many image processing algorithms work more effectively or only accept single-channel images.
- **Performance**: Reduces computational complexity by eliminating color information.

Adjusting Parameters

- `cv2.cvtColor` function parameters:
  - `src`: Source image (`roi` in this case).
  - `code`: Color space conversion code (`cv2.COLOR_BGR2GRAY`).
- **Adjustment**: There are no adjustable parameters here unless you want to convert to a different color space.

2.2 Global Thresholding

```
roi = cv2.threshold(roi, 75, 255, cv2.THRESH_BINARY)[1]
```

What It Does

- **Purpose**: Converts a grayscale image into a binary image using a fixed threshold value.
- **Process**:
  - **Threshold Value (75)**: Any pixel intensity below this value is set to 0 (black).
  - **Max Value (255)**: Any pixel intensity equal to or above the threshold is set to this value (white).
- **Result**: A binary image where pixels are either black or white.

Adjusting Parameters

1. **Threshold Value (75)**
   - **Effect**: Lower values make more pixels white; higher values make more pixels black.
   - **Adjustment**: Experiment with different values to suit the brightness and contrast of your image.
2. **Max Value (255)**
   - **Typically Left at 255**: Represents the maximum intensity in an 8-bit image.
   - **Adjustment**: Rarely changed unless working with images of different bit depths.
3. **Threshold Type (`cv2.THRESH_BINARY`)**
   - **Options**:
     - `cv2.THRESH_BINARY`: Pixels above the threshold are set to max value.
     - `cv2.THRESH_BINARY_INV`: Inverse of `THRESH_BINARY`.
   - **Adjustment**: Choose based on whether you want the foreground to be white or black.

Note

- **Accessing the Thresholded Image**: The function returns a tuple `(retval, dst)`. We use `[1]` to get the thresholded image (`dst`).

2.3 Canny Edge Detection

```
roi = cv2.Canny(roi, 100, 255, apertureSize=5)
```

What It Does

- **Purpose**: Detects edges in an image by looking for areas of rapid intensity change.
- **Process**:
  1. **Noise Reduction**: Applies a Gaussian filter (internally) to reduce noise.
  2. **Gradient Calculation**: Finds intensity gradients in the image.
  3. **Non-Maximum Suppression**: Thins out the edges.
  4. **Double Thresholding**: Determines potential edges.
  5. **Edge Tracking**: Finalizes edge detection by suppressing weak edges.

Adjusting Parameters

1. **Threshold1 (100) & Threshold2 (255)**
   - **Threshold1 (Lower)**: Edges with intensity gradient below this are discarded.
   - **Threshold2 (Upper)**: Edges with intensity gradient above this are considered strong edges.
   - **Adjustment**:
     - *Decrease Threshold1*: Detects more edges (including noise).
     - *Increase Threshold2*: Only detects very strong edges.
2. **Aperture Size (`apertureSize=5`)**
   - **Represents**: Size of the Sobel kernel used to find image gradients.
   - **Possible Values**: 3, 5, or 7.
   - **Adjustment**:
     - *Smaller Aperture*: Detects finer edges.
     - *Larger Aperture*: Detects broader edges but may miss fine details.

2.4 Adaptive Thresholding

```
roi = cv2.adaptiveThreshold(
    roi, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 5
)
```

What It Does

- **Purpose**: Applies thresholding to small regions of the image, allowing for variations in lighting across the image.
- **Process**:
  1. **Divides Image**: Splits the image into smaller regions (blocks).
  2. **Calculates Threshold for Each Block**: Uses the weighted sum of neighborhood pixel values minus a constant.
  3. **Applies Threshold**: Converts pixels to black or white based on local thresholds.

Parameters Explained

1. **Max Value (255)**
   - **Purpose**: The value assigned to pixels exceeding the threshold.
   - **Typically Left at 255**.
2. **Adaptive Method (`cv2.ADAPTIVE_THRESH_GAUSSIAN_C`)**
   - **Options**:
     - `cv2.ADAPTIVE_THRESH_MEAN_C`: Uses the mean of the neighborhood.

  - `cv2.ADAPTIVE_THRESH_GAUSSIAN_C`: Uses a weighted sum (Gaussian window) of the neighborhood.
- **Adjustment**: Choose based on the image characteristics.

3. **Threshold Type (`cv2.THRESH_BINARY`)**
   - **Same as in Global Thresholding**.
4. **Block Size (11)**
   - **Represents**: Size of the neighborhood considered for threshold calculation.
   - **Must Be**: An odd number greater than 1.
   - **Adjustment**:
     - *Smaller Block Size*: Threshold adapts to smaller variations; useful for fine details.
     - *Larger Block Size*: Threshold adapts to broader variations; useful for smoother regions.
5. **Constant (5)**
   - **Subtracted From the Weighted Sum**: Fine-tunes the threshold.
   - **Adjustment**:
     - *Increase Constant*: Makes the threshold more stringent; more pixels may become black.
     - *Decrease Constant*: Less stringent; more pixels may become white.

Why Use Adaptive Thresholding?

- **Uneven Lighting Conditions**: Useful when the image has varying illumination.
- **Better Detail Preservation**: Captures local details that global thresholding might miss.

2.5 Bilateral Filtering

```
roi = cv2.bilateralFilter(roi, 9, 75, 75)
```

What It Does

- **Purpose**: Smooths the image while preserving edges, reducing noise without blurring sharp edges.
- **Process**:
  - **Spatial Filtering**: Considers pixels within a certain spatial distance (neighborhood).
  - **Intensity Filtering**: Considers pixels with similar intensity values.
- **Result**: A denoised image that retains important edge information.

Adjusting Parameters

1. **Diameter (9)**
   - **Represents**: The diameter of each pixel neighborhood used during filtering.
   - **Adjustment**:
     - *Smaller Diameter*: Less smoothing; faster computation.
     - *Larger Diameter*: More smoothing; may preserve edges better but increases computation time.
2. **SigmaColor (75)**
   - **Represents**: Filter sigma in the color (intensity) space.
   - **Adjustment**:
     - *Smaller SigmaColor*: Only pixels with similar intensity are considered; less blurring.
     - *Larger SigmaColor*: More colors are mixed together; more blurring.
3. **SigmaSpace (75)**
   - **Represents**: Filter sigma in the coordinate space (distance).
   - **Adjustment**:
     - *Smaller SigmaSpace*: Only nearby pixels influence each other; edges are better preserved.
     - *Larger SigmaSpace*: Distant pixels can influence each other if their colors are similar; more global smoothing.

Why Use Bilateral Filtering?

- **Edge Preservation**: Unlike other smoothing filters (like Gaussian blur), the bilateral filter preserves edges while reducing noise.
- **Preprocessing Step**: Useful before applying edge detection or thresholding to improve results.

Chapter 3

Effective Parameter Adjustment

3.1 How to Adjust Parameters Effectively

1. **Understand Your Image**
    - Analyze the histogram of the image to understand the distribution of pixel intensities.
    - Identify areas with varying lighting or important features.
2. **Experiment Iteratively**
    - Start with default or recommended values.
    - Adjust one parameter at a time to see its effect.
3. **Visual Inspection**
    - After each adjustment, visually inspect the output image.
    - Check if important features are preserved and noise is minimized.
4. **Automate Parameter Selection**
    - For large datasets, consider algorithms that automatically adjust parameters based on image statistics.
5. **Combine Techniques**
    - Sometimes, applying multiple filters in sequence yields better results (e.g., use bilateral filtering before thresholding).
    - Denoising before thresholding can improve edge detection (e.g., edge detection after smoothing).

Chapter 4

Practical Tips

4.1 Preprocessing

- **Noise Reduction**:
  - Apply filters like `GaussianBlur` to reduce noise before thresholding.
  - Apply the bilateral filter to reduce noise while preserving edges before thresholding or edge detection.

4.2 Post-processing

- **Morphological Operations**: Use erosion or dilation to remove small noise or fill gaps.
- **Bilateral Filter Sensitivity**: Sensitive to `sigmaColor` and `sigmaSpace`; small changes can have significant effects.
- **Adaptive Thresholding**: The choice of block size and constant can greatly affect the result.

4.3 Visualization

- Use plotting libraries like `matplotlib` to display images inline if working in a Jupyter notebook.
- Compare the original and processed images side by side.

Chapter 5

Conclusion

By understanding the purpose and effect of each parameter, you can fine-tune these filters to enhance your images effectively. Remember that the optimal settings often depend on the specific characteristics of your images, so don't hesitate to experiment!