COMP37212-CW1-Report

Vansh Goenka

February 26, 2024

1 Introduction

The purpose of this report is to present the findings of Coursework 1, which investigates the impact of using image convolution by utilizing average and weighted average smoothing kernels for blurring. The work encompassed the implementation of a custom convolution function, the calculation of horizontal and vertical gradient images, the derivation of the edge strength image, and the application of thresholding techniques to isolate prominent image edges.

2 Design

The script is written in Python and all of the functions are written from scratch excluding some use of external libraries which include - NumPy, openCV, and matplotlib. Numpy arrays support matrix manipulations, and OpenCV manages image I/O. Matplotlib for plotting histograms. The convolution function iteratively processes pixels with the kernel, using explicit looping over the image pixels. The edges of the input image were padded with zeros to handle the edges and corners of the original image to prevent the loss of potentially valuable information. The filters used will be discussed in details.

2.1 Convolution

Convolution involves applying a small filter (kernel) to an image, modifying the pixel values based on their neighboring pixels. This technique is used for various purposes like blurring, sharpening, and edge detection. Here we are using it to blur the image by using appropriate kernel.

$$\tilde{\mathbf{I}}(\mathbf{x},\mathbf{y}) = \frac{\sum_{\mathbf{a}}\sum_{\mathbf{b}} \mathbf{g}(\mathbf{a},\mathbf{b})\mathbf{I}(\mathbf{x}+\mathbf{a},\mathbf{y}+\mathbf{b})}{\sum_{\mathbf{c}}\sum_{\mathbf{e}} \mathbf{g}(\mathbf{c},\mathbf{e})}$$

Figure 1: Formula for convolution.

2.2 Mean Filter

The mean filter is a basic smoothing technique that replaces a pixel's value with the average of its surrounding pixels. All the values in the kernel are '1'. We normalize the values with the kernel sum to preserve the image brightness. The disadvantage of the mean filter is that it treats all neighboring pixels equally.

$\frac{1}{9}$

1	1	1
1	1	1
1	1	1

Figure 2: A 3x3 Mean Filter.

2.3 Weighted Filter

The weighted-mean filter addresses the spatial bias issue by assigning specific weights to each kernel element so that closer pixels have more influence. These weights are determined using a 2D Gaussian distribution (the familiar bell curve) which is calculated using the formula given below. Here **'a'** and **'b'** represent the distance from the center of the kernel to a specific pixel in X and Y direction respectively. σ is the standard deviation of the Gaussian distribution. This results in a normalized distribution of weights that are used during the convolution process. By using weighted mean, we can make our results sensitive to pixel values as well as pixel position.

$$\left(\exp(-a^2/2\sigma^2)\right) \times \left(\exp(-b^2/2\sigma^2)\right)$$

Figure 3: Formula for calculating Gaussian kernel (Decomposed).

2.4 Gradient Calculation

The gradient function calculates the edge strength within an image. The input images are convolved with Sobel kernels to detect horizontal and vertical gradients. Then the results are combined to determine the overall edge strength magnitude using the formula given below (figure 4). Finally, gradient values are normalized for better visualization.

$$\|\nabla f\| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}$$

Figure 4: Formula for gradient calculation.

2.5 Thresholding

We perform simple binary binary thresholding on the image. The function creates a new binary image. Pixels in the original image with values greater than the set threshold are set to white (255), and all other pixels are set to black (0). This results in a high-contrast image with clear edges.

3 Experiment

This coursework examines image edge detection via thresholding. The workflow starts with smoothing the image with a blurring kernel(normal or weighted average), then gradient calculation by using Sobel kernels, and finally binary thresholding. The thresholded images obtained using the mean and weighted-mean filtering are compared. The experiment utilizes an image of a kitten (Figure 5). The experiments will include how changing the sigma value, kernel size, and threshold value impacts the image.

Figure 5: A kitty playing.

3.1 Changing the Kernel Size - Impact on Normal and Weighted Filter

Experiments are carried out using a 3x3, 5x5, and 7x7 kernel to compare the differences due to changing the kernel size. Upon experimentation, it was observed that on comparing the 3x3 mean filter and weighted filter, no difference was observed hence we move forward with a 5x5 kernel. Some differences are observed, but not very significant. On experimenting with a 7x7 kernel, we start to see some significant differences. The edges are more well defined and there is considerably less noise. See in figure 6.

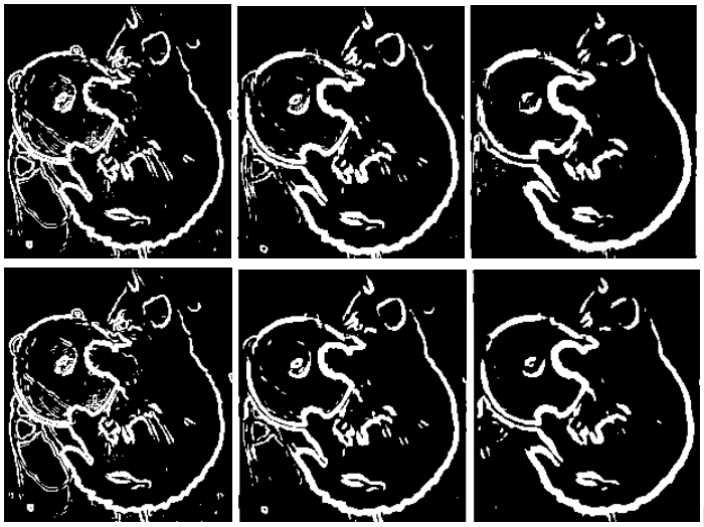

Figure 6: (Top: Mean, Bottom: Weighted) Left :3x3 Middle :5x5 Right :7x7

3.2 Changing the Threshold value - Impact on Normal and Weighted Filter

To get the best value for thresholding, I experimented with different values of thresholds - 20, 25, 30 and 35 keeping the sigma value constant at 2.5 on a 7x7 weighted kernel. The results are shown in figure 7. When the threshold values were less than 30, the image had a lot of small scale structures and noise, and when the threshold was greater than 30, we lot some of the edges. So as a results the best values of threshold was selected as 30.

Figure 7: Threshold = 20, Threshold = 25, Threshold = 30, Threshold = 35

3.3 Changing the sigma value - Impact on Normal and Weighted Filter

The sigma value determines the amount of blur in the image. To estimate the best value of sigma, we try different values from 1, 2.5, and 5. Based on the above experiments, we can say for sure that a kernel size of 7x7 and a threshold value will probably give the best edge detection. So we keep these parameters fixed and experiment with sigma values. When the sigma value was 1, there were too much details (less blurring). When sigma was 2.5 the result was an image with less details and not much noise. And when the sigma value is 5, there is too much blurring and hence the edges are lost.

Figure 8: Left : sigma = 1 Middle : sigma = 2.5 Right : sigma = 5

4 Conclusion

After all the experimentation it was observed that given a fixed kernel of size 7x7, sigma = 2.5 and threshold as 30, a weighted mean filter was better and more accurate than a normal averaging filter. This is due to the fact that a normal average kernel does not give any importance to the neighbor pixels and treats them equally. While a weighted mean filter gives more weights to the neighbors depending on the sigma values. The results from the weighted kernel had significantly less noise and minor irrelevant small-scale structures. See figure 9 and 10 to see the difference between a 7x7 normal kernel and a weighted kernel of size 7x7. The histogram in figure 11 shows the difference in the gradient values of mean and weighted filter.

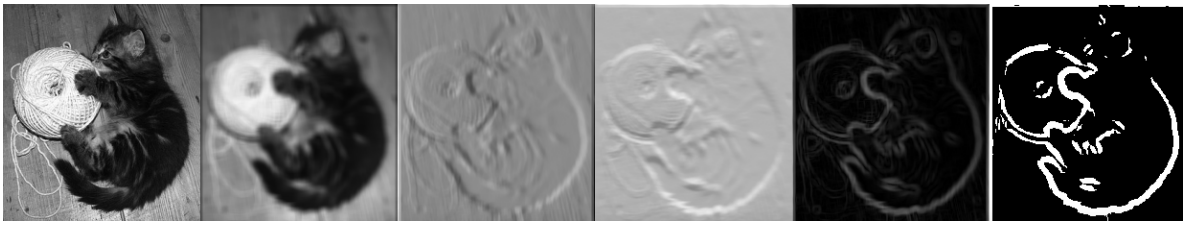

Figure 9: Mean Filter : 7x7

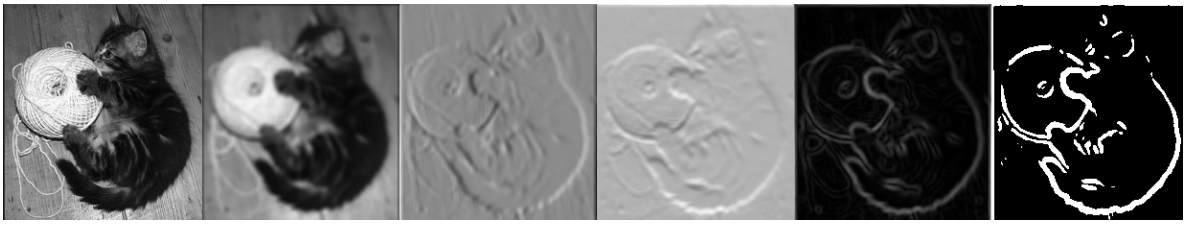

Figure 10: Weighted Filter : 7x7

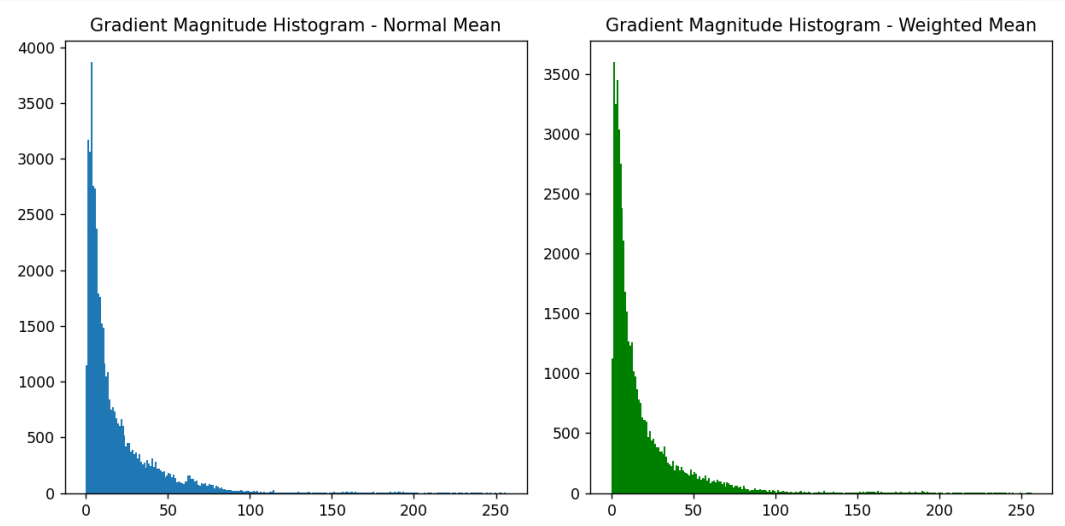

Figure 11: Mean vs Weighted image gradients

5 Appendix

```python
def convolve_image(image, kernel):
    # Getting the  image and kernel dimensions
    image_height = image.shape[0]
    image_width = image.shape[1]
    kernel_height = kernel.shape[0]
    kernel_width = kernel.shape[1]

    print(kernel_size, kernel_height)

    # Calcultaing padding dimensions for handling edges
    pad_height = (kernel_height - 1) // 2
    pad_width = (kernel_width - 1) // 2

    # Creating padded image
    padded_image = pad_image(image, pad_height, pad_width)

    # Initializing the output image with zeros
    output_image = np.zeros_like(image)

    # looping reversely is more efficient
    for y in range(image_height):
        for x in range(image_width):
            # Extract the region of interest around the current pixel
            region_of_interest = padded_image[y:y + kernel_height, x:x + kernel_width]
            # Element-wise multiplication and summation
            output_image[y, x] = np.sum(region_of_interest * kernel)

    return output_image
```

Figure 12: Function for convolution

```python
def compute_sobel_gradients(smoothed_image):

    # Converting to float32 for calculations and to avoid overflow while using the normalization function
    smoothed_image = smoothed_image.astype('float32')

    # Defining Sobel kernels
    sobel_x_kernel = np.array([[-1, 0, 1],
                               [-2, 0, 2],
                               [-1, 0, 1]])


    sobel_y_kernel = np.array([[-1, -2, -1],
                               [ 0,  0,  0],
                               [ 1,  2,  1]])

    # calculating the gradient_x and gradient_y using the convolution function
    gradient_x = convolve_image(smoothed_image, sobel_x_kernel)
    gradient_y = convolve_image(smoothed_image, sobel_y_kernel)


    # Calculating gradient magnitude
    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

    # Normalizing
    gradient_x = normalize(gradient_x)
    gradient_y = normalize(gradient_y)
    gradient_magnitude = normalize(gradient_magnitude)

    return gradient_x, gradient_y, gradient_magnitude
```

Figure 13: Function for Calculating gradient magnitude

```python
def gaussian_kernel(sigma, kernel_size):
    if kernel_size % 2 == 0:
        raise ValueError("Kernel size must be an odd number.")

    center = (kernel_size - 1) // 2
    x, y = np.mgrid[-center:center+1, -center:center+1]

    # decomposable gaussian
    g = np.exp((-x**2)/(2*sigma**2)) * np.exp((-y**2)/(2*sigma**2))
    return g / g.sum()  # Shorter normalization
```

Figure 14: Function for calculating Gaussian kernel

```python
def threshold_with_one(image, initial_threshold):
    # initializing the image with zeros first
    thresholded_image = np.zeros_like(image)
    # looping through all the pixel locations
    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            if image[y, x] > initial_threshold:
                thresholded_image[y, x] = 255
            else :
                thresholded_image[y,x] = 0

    return thresholded_image
```

Figure 15: Function for Thresholding

```python
# Normalization
def normalize(array):
    min = array.min()
    max = array.max()
    range = max - min
    return (255 * (array - min) / range).astype('uint8')  # Scaling to 0-255, converting to 8-bit
```

Figure 16: Function for normalizing the image gradients

```python
def pad_image(image, pad_height, pad_width, pad_value=0):
    padded_height = image.shape[0] + 2 * pad_height
    padded_width = image.shape[1] + 2 * pad_width
    padded_image = np.full((padded_height, padded_width), pad_value, dtype=image.dtype)
    padded_image[pad_height:-pad_height, pad_width:-pad_width] = image
    return padded_image
```

Figure 17: Function for Padding the image