Computer Vision CW1 Report

1. Introduction

The main goal of this task is to threshold an image in order to locate edges. First, the image goes through the process of filtering, and then the Sobel kernel is applied to calculate the image gradient on the X-axis and Y-axis, and then they are combined to obtain the image gradient size. Finally, by generating histograms, we find a feasible threshold to segment the image. We will compare two filtering methods, smoothing with mean filter and weighted mean filter. The subject of this experiment is the kitten in Figure 1.

Figure 1: kitty image

2. Image padding-Q1

Typically, padding is applied to an image's boundaries by adding pixels with a certain value, such as zeros or the average pixel value. In many computer vision algorithms, padding is a crucial step since it aids in maintaining the size and spatial organisation of the image during operations like convolution and filtering.

```python
def padding(image, top_pad, bottom_pad, left_pad, right_pad):
    # Get the dimensions of the input image
    image_height, image_width = image.shape

    # Get new image with padding
    padded_image = np.zeros((image_height + top_pad + bottom_pad, image_width + left_pad + right_pad), dtype=np.uint8)

    # Copy original image to padded image
    padded_image[top_pad:image_height+top_pad, left_pad:image_width+left_pad] = image

    return padded_image
```

Figure 2: padding function python code

```
(233, 208)
```

Figure 3: image padding

3. Convolution-Q2

Convolution is the term used to describe the process of applying a kernel or filter to an image's pixels, computing a weighted sum of each location's adjacent pixels, and then replacing the centre pixel's value with the sum. Convolution can be used for a wide range of tasks in image processing and computer vision, including edge detection, smoothing, more. The specific kernel or filter(average and the weighted average) used in the convolution operation will depend on the task at hand and the desired output.
In Figure 4, I chose two 3x3 kernels, one is the average smoothing kernels, the other is the weighted average smoothing kernels. Next I will compare the two kernel experiments.

```python
def convolution(image, weighted = False):
    # Get the dimensions of the input image
    image_height, image_width = image.shape

    # whether the average or the weighted average smoothing kernels
    if not weighted:
        kernel = np.ones((3,3), np.float32)/9
    else:
        # Define the size of the Gaussian kernel
        kernel_size = (3, 3)

        # Define the standard deviation for the Gaussian kernel
        sigma = 1.0

        # Get a 2D Gaussian kernel
        kernel = np.zeros(kernel_size)
        for i in range(kernel_size[0]):
            for j in range(kernel_size[1]):
                x, y = i - kernel_size[0] // 2, j - kernel_size[1] // 2
                kernel[i, j] = np.exp(-(x ** 2 + y ** 2) / (2 * sigma ** 2))
        kernel /= np.sum(kernel)

    # image padding
    padded_image = padding(image,1,1,1,1)

    # Initialize the output image
    convolution_image = np.zeros((image_height, image_width), dtype=np.uint8)

    # Loop over the pixels in the input image
    for i in range(1, image_height+1):
        for j in range(1, image_width+1):

            # Compute the sum of element-wise multiplication of kernel and image sub-region
            sub_region = padded_image[i-1:i+2, j-1:j+2]
            pixel_sum = np.sum(np.multiply(sub_region, kernel))

            # Set the output pixel value
            convolution_image[i-1, j-1] = np.uint8(pixel_sum)

    return convolution_image
```

Figure 4: convolution python code

Mean filter

A very basic smoothing kernel known as the mean filter replaces each pixel value with the average of it and all of its neighbours. The drawback of this approach is that it ignores the spatiality of the pixels, which means that a neighbouring pixel that doesn't represent the same object as the one being considered will nevertheless have an equal impact on the final pixel.
My 3x3 Mean filter kernels: [[0.11111111 0.11111111 0.11111111]
[0.11111111 0.11111111 0.11111111]
[0.11111111 0.11111111 0.11111111]]

Weighted-mean filter

The weighted-mean filter accounts for the issue mentioned previously by setting a weight for each element in the kernel. The weights in the kernel can be chosen to reflect the importance or

relevance of each neighboring pixel to the center pixel. I choice Gaussian kernel as weighted-mean filter(sigma=1).
My 3x3 weight-mean filter kernels: [[0.07511361 0.1238414 0.07511361]
[0.1238414 0.20417996 0.1238414]
[0.07511361 0.1238414 0.07511361]]

4. Gradient Image-Q3

The gradient image of an image is a representation of the rate of change of pixel intensities across the image.The gradient image represents the direction and magnitude of the change in pixel intensities at each point in the image. The direction of the gradient is perpendicular to the direction of the steepest increase in pixel intensity, and the magnitude of the gradient is proportional to the steepness of the increase. with different colors or brightness levels representing different magnitudes or directions of the gradient.
In figure 5, first I select the kernel for the gradient operation that the Sobel operator applies to the image. A gradient kernel is then applied to the image using a convolution to compute the image gradient on the X and Y axes. Finally find the gradient magnitude.

```python
def gradient(image):
    # Define the Sobel operator kernels
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

    # Pad the image with zeros to deal with edges
    padded_image = padding(image,1,1,1,1)

    # Compute the horizontal gradient using convolution_image
    gradientX = np.zeros_like(image, dtype=np.float32)
    for i in range(1, image.shape[0]+1):
        for j in range(1, image.shape[1]+1):
            sub_region = padded_image[i-1:i+2, j-1:j+2]
            pixel_sum = np.sum(np.multiply(sub_region, sobel_x))
            gradientX[i-1, j-1] = pixel_sum

    # Compute the vertical gradient using convolution_image
    gradientY = np.zeros_like(image, dtype=np.float32)
    for i in range(1, image.shape[0]+1):
        for j in range(1, image.shape[1]+1):
            sub_region = padded_image[i-1:i+2, j-1:j+2]
            pixel_sum = np.sum(np.multiply(sub_region, sobel_y))
            gradientY[i-1, j-1] = pixel_sum

    # Compute the gradient magnitude
    gradientMagn = np.sqrt(gradientX**2 + gradientY**2)

    return gradientX, gradientY, gradientMagn
```

Figure 5: Gradient Image python code

5. Threshoding-Q4

Thresholding is a fundamental computer vision process that divides pixel values into two classes depending on a predetermined threshold value, transforming a grayscale or colour image into a binary image. A thresholding function is then applied to the edge strength images. By examining the histogram of the magnitude image and selecting a value where the rate of change becomes smaller than at the peak, the value for this function was determined. All pixels below the threshold value are set to 0, while all pixels above the threshold value are set to 255 during thresholding(figure 6).
In figure 7, I also use the average kernel to generate images with 3 different T (T=50, 75, 100) to compare the results. You will find that as the T value increases, the details(wood-grain, fur) of the image gradually disappear and the outline becomes clearer.

```python
def thresholding(image, threshold_value):
    # Create an output image array of the same shape as the input image
    threshold_image = np.zeros_like(image)

    # Iterate over each pixel and set the pixel value based on the threshold
    for i in range(threshold_image.shape[0]):
        for j in range(threshold_image.shape[1]):
            if image[i,j] >= threshold_value:
                threshold_image[i,j] = 255
            else:
                threshold_image[i,j] = 0

    return threshold_image
```

Figure 6: thresholding python code

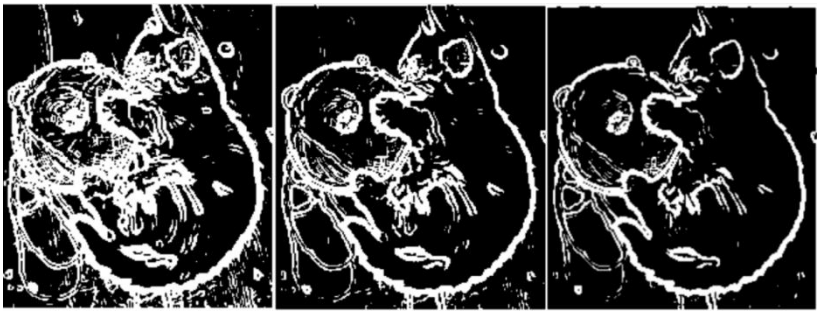

Figure 7: different T-edge images(T=50,75,100)

In Q4, The T value I chose is 100. Because I compared multiple sets of T values(Figure 7,11), I found that the image generated by T=100 has more clear outlines and no redundant fur details. Better results.

6. Experiment (Average kernels vs Weighted average kernels)-Q5

During the experiment, I found that 3X3 size average kernel and weighted average kernel is used for smoothing, also follow-up images, there is not much difference in the image. Maybe for small kernels such as 3x3, the difference between a uniform average and a weighted average may be minimal. This is because the kernel only covers a small neighborhood around each pixel, and the difference in weights between nearby pixels may not be significant enough to cause a noticeable difference in the smoothed image.
So in order to better compare the results of the two kernel, I increased the size of the kernel to 7x7.

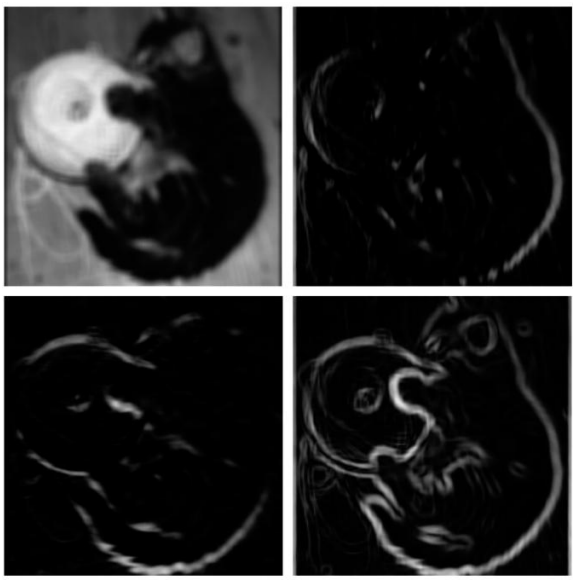

Figure 8: Image smoothed using a 7x7 average kernel; top-left: convolution image, top-right: X gradient, bottom-left: Y gradient, bottom-right: image magnitude

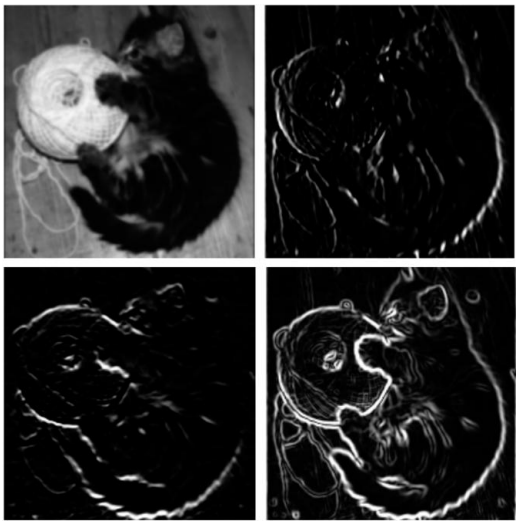

Figure 9: Image smoothed using a 7x7 weight mean kernel; top-left: convolution image, top-right: X gradient, bottom-left: Y gradient, bottom-right: image magnitude

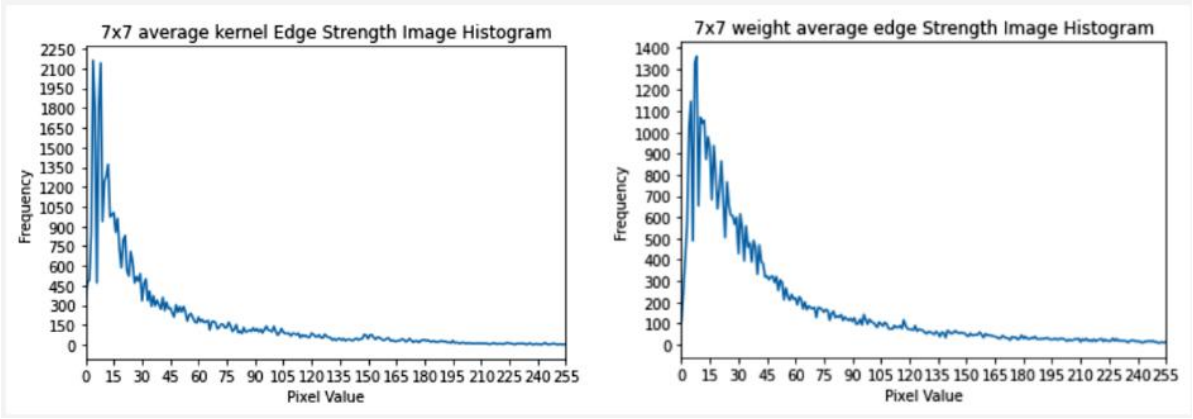

Figure 10: Histograms; left: average kernel, right: weighted-average kernel

Figure 11: Edges image; left: average kernel(T=75), right: weighted-average kernel(T=100)

Result

In figure 11, we found that the boundary between the kitten and the ball of string in the image on the left is thicker, but the fur pattern and wood grain are gone. The kitten on the right still retains this clear boundary, but the fur pattern and wood grain don't have perfect places to go. And the choice of threshold is also different.(figure 10)

Analysis and conclusion

In some cases, a uniform smoothing may be sufficient for removing noise or smoothing the image, while in other cases a more selective smoothing may be necessary to preserve important features or structures in the image.

Because the weighted average kernel takes into account the spatial location of each pixel in the kernel, it can preserve edges better than the average kernel. In particular, the weighted average kernel is less likely to blur edges or boundaries in the image.

7. Extra Experiment (Bigger size of kernel)-Q6

In the previous experiments, we found that the results of the 3X3-sized kernel did not have a significant contrast effect. So in this experiment, I plan to explore the kernels of different sizes, compare the resulting images, and draw conclusions.(average kernel: 3,5,7,9). Perform thresholding of the edge strength image, and hence display the major edges of the image.
Kernel: kernel = np.ones((i,i), np.float32)/(i*i)

```python
def convolution_diff_kernel(image, kernel):
    # Get the dimensions of the input image and kernel
    image_height, image_width = image.shape

    N = kernel.shape[0]-1
    P = N//2

    padded_image = padding(image,P,P,P,P)

    # Initialize the output image
    output_image = np.zeros((image_height, image_width), dtype=np.uint8)

    # Loop over the pixels in the input image
    for i in range(1, image_height+1):
        for j in range(1, image_width+1):

            # Compute the sum of element-wise multiplication of kernel and image sub-region
            sub_region = padded_image[i-1:i+N, j-1:j+N]
            # print(sub_region)
            pixel_sum = np.sum(np.multiply(sub_region, kernel))

            # Set the output pixel value
            output_image[i-1, j-1] = np.uint8(pixel_sum)

    return output_image
```

Figure 12: Convolution plus function python code

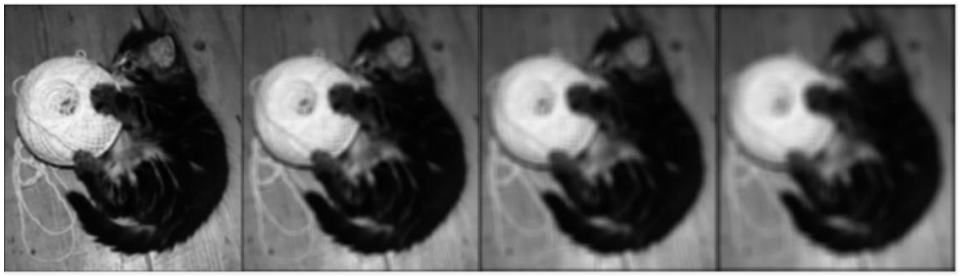

Figure 13: Convolution image(size 3,5,7,9)

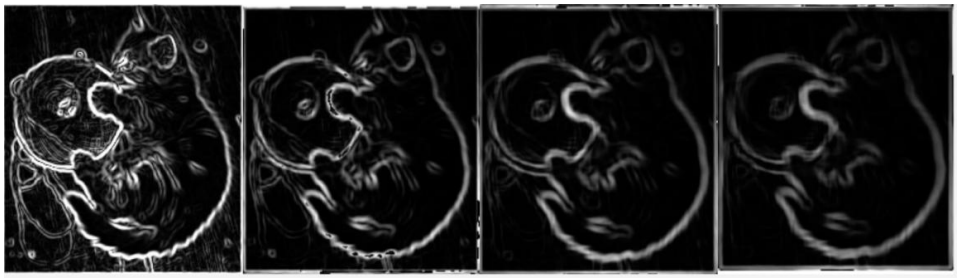

Figure 14: image magnitude(size 3,5,7,9)

Figure 15: Edges image(same T=100)(size 3,5,7,9)

Figure 16: similar Edges image(different T=100,70,60,50)(size 3,5,7,9)

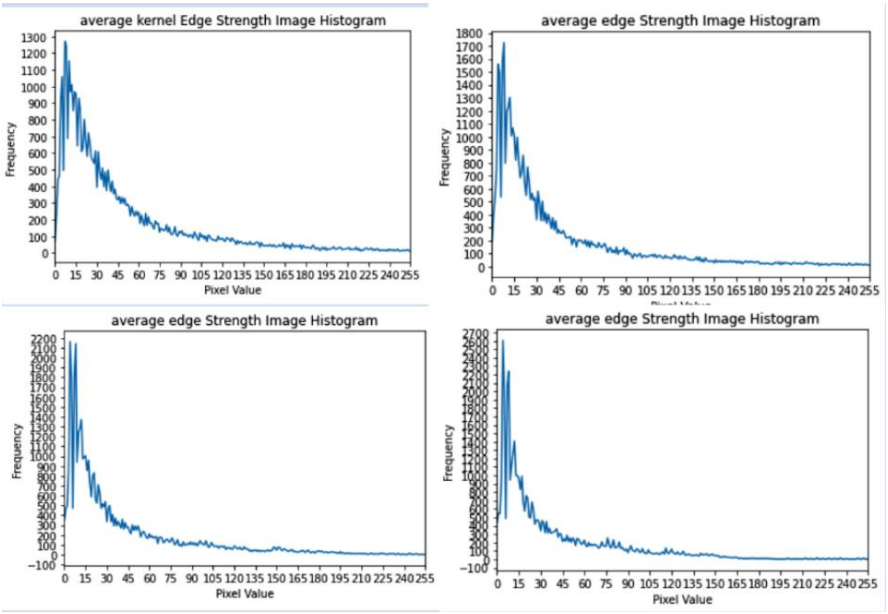

Figure 17: Histograms(size 3,5,7,9)

Result

In figure 13 and 14, We found that the image was getting blurry. This helps the effect of the edge image that follows. Because of the blurring, some details of the pattern are weakened(Kitten pattern and wood grain), making the edges more prominent.

In figure 15, 16, 17, our aim is to filter out as much woodgrain and pattern on the fur as possible. But at some point the edges of the kitten start to lose their shape and Kitten pattern and wood grain disappear. For different kernel sizes, we find that the required threshold is significantly lower.But the resulting image has thicker edges.

Analysis and conclusion

A smaller average kernel size, such as 3x3 or 5x5, will produce a more selective smoothing of the image and preserve more fine details and edges. However, smaller kernels are more sensitive to noise and may not be effective at removing larger scale features or structures in the image.

On the other hand, an image will be smoother and have more noise and bigger scale characteristics removed when the average kernel size is larger, such as 7x7 or 9x9. Larger kernels, however, can potentially distort critical edges and details in the image.

Generally, the specific qualities of the image being processed and the desired result will determine the kernel size. A larger kernel may be preferable for images with more consistent textures or noise, while a smaller kernel may be ideal for images with many little details or edges. Finding the best kernel size for a specific image or application frequently requires testing out a variety of kernel sizes.

Appendix image

```
# Load input image
input_image = cv2.imread("./kitty.bmp", cv2.IMREAD_GRAYSCALE)

# Perform convolution
convolution_image = convolution(input_image, False)

# Perform convolution
grad_x, grad_y, grad_mag = gradient(convolution_image)

# Set the threshold value
threshold_value = 100
threshold_image = thresholding(grad_mag, threshold_value)

# Show the output image
cv2_imshow(input_image)
cv2_imshow(convolution_image)
cv2_imshow(grad_x)
cv2_imshow(grad_y)
cv2_imshow(grad_mag)
cv2_imshow(threshold_image)
# cv2.imwrite('avg_convolution_image.jpg', convolution_image)
# cv2.imwrite('avg_grad_x.jpg', grad_x)
# cv2.imwrite('avg_grad_y.jpg', grad_y)
# cv2.imwrite('avg_grad_mag.jpg', grad_mag)
# cv2.imwrite('threshold_image.jpg', threshold_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Question 1,2,3,4 - average kernel

```
# Plot the histogram
histogram, bin_edges2 = np.histogram(grad_mag, bins=256, range=[0, 256])
plt.plot(histogram)
plt.xlim([0, 256])
plt.xlabel('Pixel Value')
plt.ylabel('Frequency')
plt.title('average kernel Edge Strength Image Histogram')
x_major_locator=MultipleLocator(15)
y_major_locator=MultipleLocator(100)
ax=plt.gca()
ax.xaxis.set_major_locator(x_major_locator)
ax.yaxis.set_major_locator(y_major_locator)
plt.xlim(0,255)
plt.show()
```

Histogram

```python
# Load input image
input_image2 = cv2.imread("./kitty.bmp", cv2.IMREAD_GRAYSCALE)

# Perform convolution
convolution_image2 = convolution(input_image, True)

# Perform convolution
grad_x2, grad_y2, grad_mag2 = gradient(convolution_image2)

# Set the threshold value
threshold_value2 = 100
threshold_image2 = thresholding(grad_mag2, threshold_value2)

# Show the output image
cv2_imshow(input_image2)
cv2_imshow(convolution_image2)
cv2_imshow(grad_x2)
cv2_imshow(grad_y2)
cv2_imshow(grad_mag2)
cv2_imshow(threshold_image2)
cv2.imwrite('weight_avg_convolution_image.jpg', convolution_image2)
cv2.imwrite('weight_avg_grad_x.jpg', grad_x2)
cv2.imwrite('weight_avg_grad_y.jpg', grad_y2)
cv2.imwrite('weight_avg_grad_mag.jpg', grad_mag2)
cv2.imwrite('weight_avg_threshold_image.jpg', threshold_image2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Question5 - weight-average kernel

```python
arr = [3,5,7,9]

# Load input image
input_image3 = cv2.imread("./kitty.bmp", cv2.IMREAD_GRAYSCALE)

for i in arr:
    # kernel
    kernel = np.ones((i,i), np.float32)/(i*i)

    # Perform convolution
    convolution_image3 = convolution_diff_kernel(input_image3, kernel)

    # Perform convolution
    grad_x3, grad_y3, grad_mag3 = gradient(convolution_image3)

    # Set the threshold value
    threshold_value3 = 40

    threshold_image3 = thresholding(grad_mag3, threshold_value3)

    # Show the output image
    cv2_imshow(input_image3)
    cv2_imshow(convolution_image3)
    cv2_imshow(grad_x3)
    cv2_imshow(grad_y3)
    cv2_imshow(grad_mag3.astype(np.uint8))
    cv2_imshow(threshold_image3)

    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Extra experiment