

1. Introduction

Several important applications of deep learning in the field of Computer Vision (CV) include image classification, object detection, and image segmentation. Image Segmentation is for pixel detection and classification. It can be applied to tasks such as portrait photograph, autonomous driving, biomedicine and so forth.

Edge detection is an image processing technique for finding the boundaries of objects within images. It is used for image segmentation and data extraction and works by detecting discontinuities in brightness. Common edge detection algorithms include Sobel, Prewitt, Laplacian and Canny methods.

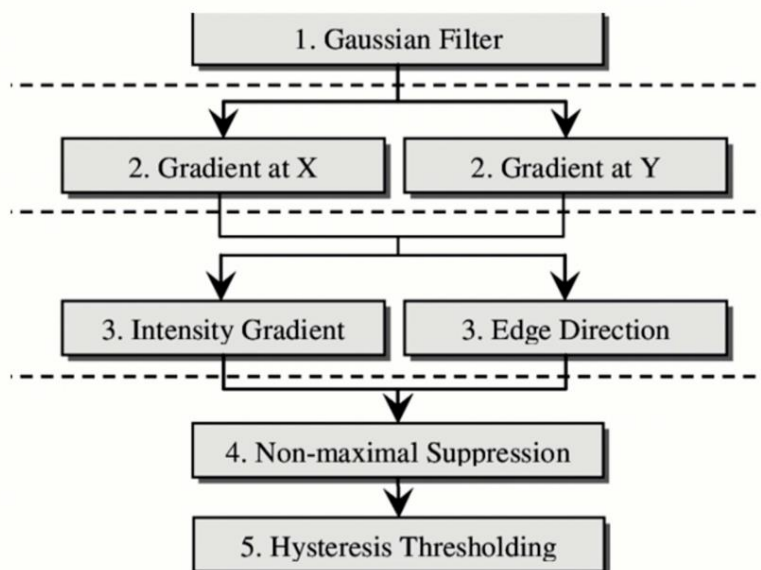
In the first assignment, I implemented the gradient technique of morphological image processing, but I didn't specifically bring it up for discussion. It is a method for boundary extraction, I think that the effect is a little similar to canny edge detector, so I want to compare their processing effect and observe the difference between my results and the canny edge detection functions provided by OpenCV.

2. Method

- Canny edge detector

Canny edge detection algorithm is a widely used algorithm that combines edge point detection and edge localization. The objectives of this image segmentation technique are low error rate, single edge point response and edge points should be well localized.

The algorithm consists of five steps:



(source:<https://medium.com/@ceng.mavuzer/canny-edge-detection-algorithm-with-python-17ac62c61d2e>)


For the edge detector, smoothing is usually a standard pre-processing step. Different degrees of smoothing affect the level of image details that we can retrieve with edge detection. The issue of varying degrees of smoothing is a concept in image process that is important beyond edge detection.

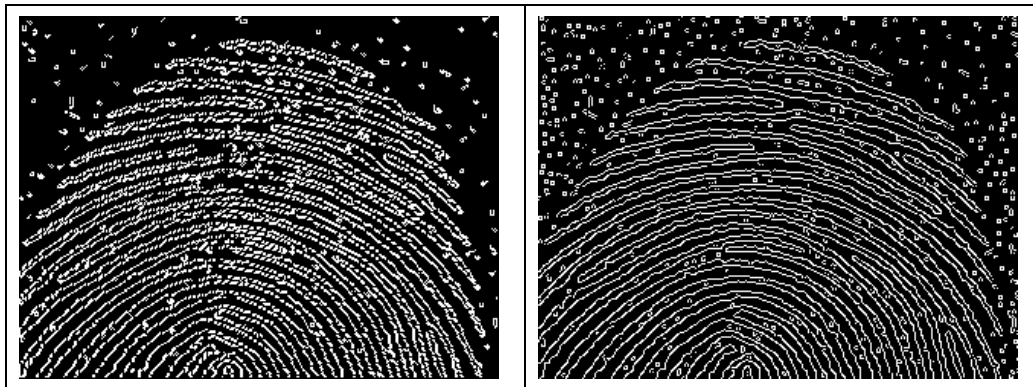
3. Experimental results

To compare with the result of gradient technique in the first assignment, one of the images I selected for canny edge detector is fingerprint. For the rest of the test images, I used two images in the textbook and one color image for experiments.


My experiment flow is implemented according to the steps mentioned in the above method section. First, the result of applying Gaussian Filter on the image is blurring and reducing the noise from the image. Then, gradient calculation allows us to find contours and outlines of objects in images. Especially, x and y derivatives are using for the calculation of gradient magnitude and gradient direction. The next part is very powerful concept canny edge detector actually deployed in this algorithm for edge detection. We check every pixel on the gradient magnitude image and choose two neighbors of the pixel according to gradient direction. If center pixel is larger than the both neighbors then keep it, otherwise set the pixel to 0. The final step is the hysteresis thresholding, it can connect a lot of broken edges.

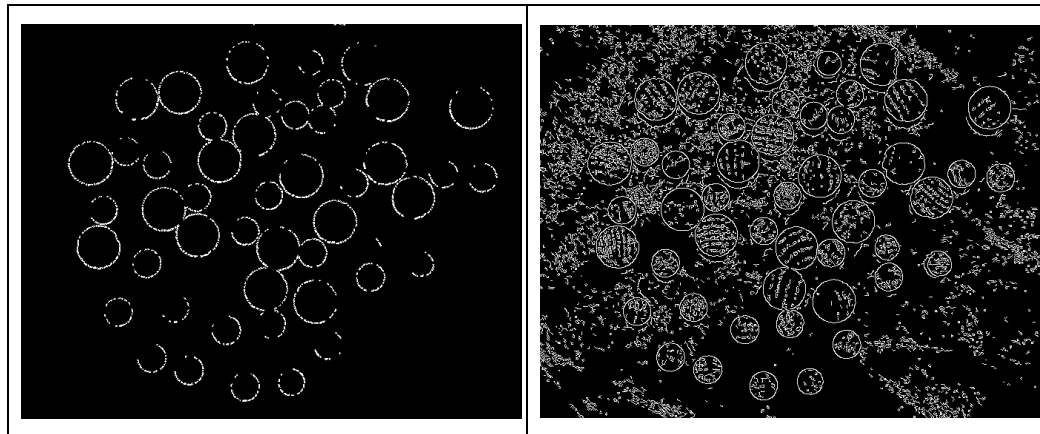
The following table is my experimental result:




1. Fingerprint	
Original image	
My code	OpenCV





2. Text	
Original image	<p>ponents or broken connection paths. There is no point in going past the level of detail required to identify those components.</p> <p>Segmentation of nontrivial images is one of the most difficult tasks in image processing. Segmentation accuracy determines the effectiveness of computerized analysis procedures. For this reason, considerable effort must be taken to improve the probability of rugged segmentation. In applications such as industrial inspection applications, at least some level of ruggedness in the environment is possible at times. The experienced image processing designer invariably pays considerable attention to such factors.</p>
My code	OpenCV
<p>ponents or broken connection paths. There is no point in going past the level of detail required to identify those components.</p> <p>Segmentation of nontrivial images is one of the most difficult tasks in image processing. Segmentation accuracy determines the effectiveness of computerized analysis procedures. For this reason, considerable effort must be taken to improve the probability of rugged segmentation. In applications such as industrial inspection applications, at least some level of ruggedness in the environment is possible at times. The experienced image processing designer invariably pays considerable attention to such factors.</p>	<p>ponents or broken connection paths. There is no point in going past the level of detail required to identify those components.</p> <p>Segmentation of nontrivial images is one of the most difficult tasks in image processing. Segmentation accuracy determines the effectiveness of computerized analysis procedures. For this reason, considerable effort must be taken to improve the probability of rugged segmentation. In applications such as industrial inspection applications, at least some level of ruggedness in the environment is possible at times. The experienced image processing designer invariably pays considerable attention to such factors.</p>

3. Wood dowels	
Original image	
My code	OpenCV



4. Color image		
Original image		
My code	OpenCV	
		

In addition, I have an idea to improve the performance of edge detection, so I tried to implement my thought, that is, calculating the absolute difference between a mask and the dilated mask by morphologic edge processing. Similarly, I didn't use the functions provided by OpenCV itself, I reused the dilation function I have implemented for the first assignment and selected the color image for testing. The following table is the result of my experiment:

My thought for improving	Canny edge detector using my code
	

This result seems to be better than the result of canny edge detection using my code, but it is still slightly worse than the effect of OpenCV function.

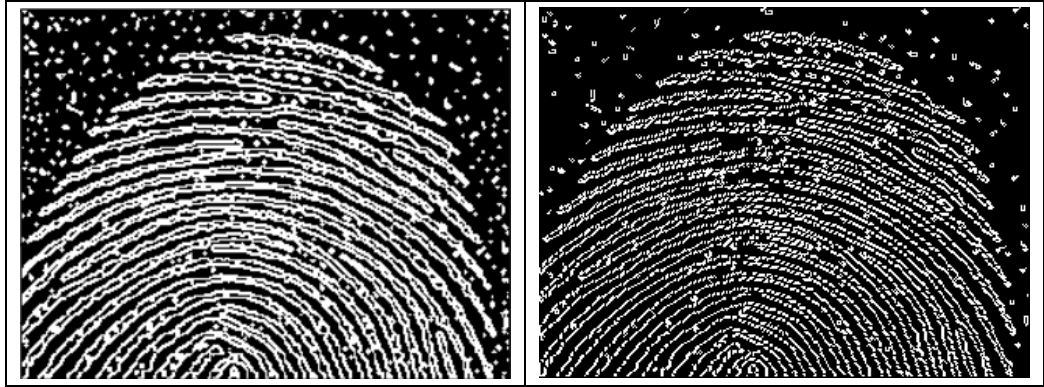
4. Discussion

- Observations / Interpretations

From the results above, it can be seen that the results of OpenCV function seem to have better results in fingerprint, text image and color image. Especially for the edge detection of people in color image, the effect is very poor, only part of the edge is framed. If there is no comparison with the original image, it may not be obvious that it is a human outline. In addition, the performance on the wood dowels image, we can observe that there are more noise in the result image of OpenCV function, the edge in the result image of my code performed looks more like the contour of wood dowels, but it has some gaps, the edge isn't connected as expected. The only explanation I can think of is that maybe my code is wrong in the last step, that is, the part of hysteresis thresholding, because the objective of this step is to connect the broken edges, weak edges that are connected to strong edges will be real edges, I would go back and check my code. As for why there is always more noise in the results of OpenCV function, I am still thinking about the reasons for this. Maybe the degree of blurring in the previous image smoothing step is lower, so the noise and smaller objects are enlarged together, since this is a trade-off. Stronger smoothing leads to less noise, reduced details and less accurate edge localization. Conversely, weaker smoothing causes more noise, more details kept and more accurate edge localization.

The other important issue I want to discuss is comparison between the gradient technique of morphological image processing and canny edge detector.

Comparison between gradient and canny edge detector	
Gradient	Canny edge detector



From the above table, because we applied Gaussian Filter on the image first, the noise has been reduced, so that the small white dots around the fingerprint are kept less. The edges detected by our canny edge detector is thinner and more precise than the edge extracted by gradient technique, but there are still some gaps.

- Remaining questions
 - Why is the performance of my code performed always worse than OpenCV functions, especially color images for portrait photograph?
I will double check my code and try my best to improve the performance, maybe professor or assistants can give me some suggestions?
 - Why there is always more noise in the results of OpenCV function, or we can say that OpenCV function would enlarge the smaller features?
The first thing I should check is how the Gaussian filter of OpenCV canny function works.

```

import cv2
import numpy as np
import matplotlib.pyplot as plt
import math

def sHalf(T, sigma):
    temp = -np.log(T) * 2 * (sigma ** 2)
    return np.round(np.sqrt(temp))

def calculate_filter_size(T, sigma):
    return 2*sHalf(T, sigma) + 1

def MaskGeneration(T, sigma):
    N = calculate_filter_size(T, sigma)
    shalf = sHalf(T, sigma)
    y, x = np.meshgrid(range(-int(shalf), int(shalf) + 1), range(-int(shalf), int(shalf) +
1))
    return x, y

def Gaussian(x,y, sigma):
    temp = ((x ** 2) + (y ** 2)) / (2 * (sigma ** 2))
    return (np.exp(-temp))

def calculate_gradient_X(x,y, sigma):
    temp = (x ** 2 + y ** 2) / (2 * sigma ** 2)
    return -((x * np.exp(-temp)) / sigma ** 2)

def calculate_gradient_Y(x,y, sigma):
    temp = (x ** 2 + y ** 2) / (2 * sigma ** 2)
    return -((y * np.exp(-temp)) / sigma ** 2)

def pad(img, kernel):
    r, c = img.shape
    kr, kc = kernel.shape
    padded = np.zeros((r + kr,c + kc), dtype=img.dtype)
    insert = np.uint((kr)/2)
    padded[insert: insert + r, insert: insert + c] = img
    return padded

```

```

def smooth(img, kernel=None):
    if kernel is None:
        mask = np.array([[1,1,1],[1,1,1],[1,1,1]])
    else:
        mask = kernel
    i, j = mask.shape
    output = np.zeros((img.shape[0], img.shape[1]))
    image_padded = pad(img, mask)
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            output[x, y] = (mask * image_padded[x:x+i, y:y+j]).sum() / mask.sum()
    return output

```

```

def Create_Gx(fx, fy):
    gx = calculate_gradient_X(fx, fy, sigma)
    gx = (gx * 255)
    return np.around(gx)

```

```

def Create_Gy(fx, fy):
    gy = calculate_gradient_Y(fx, fy, sigma)
    gy = (gy * 255)
    return np.around(gy)

```

```

def ApplyMask(image, kernel):
    i, j = kernel.shape
    kernel = np.flipud(np.fliplr(kernel))
    output = np.zeros_like(image)
    image_padded = pad(image, kernel)
    for x in range(image.shape[0]):
        for y in range(image.shape[1]):
            output[x, y] = (kernel * image_padded[x:x+i, y:y+j]).sum()
    return output

```

```

def Gradient_Magnitude(fx, fy):
    mag = np.zeros((fx.shape[0], fx.shape[1]))
    mag = np.sqrt((fx ** 2) + (fy ** 2))
    mag = mag * 100 / mag.max()

```



```

return np.around(mag)

def Gradient_Direction(fx, fy):
    g_dir = np.zeros((fx.shape[0], fx.shape[1]))
    g_dir = np.rad2deg(np.arctan2(fy, fx)) + 180
    return g_dir

def Digitize_angle(Angle):
    quantized = np.zeros((Angle.shape[0], Angle.shape[1]))
    for i in range(Angle.shape[0]):
        for j in range(Angle.shape[1]):
            if 0 <= Angle[i, j] <= 22.5 or 157.5 <= Angle[i, j] <= 202.5 or 337.5 <
Angle[i, j] < 360:
                quantized[i, j] = 0
            elif 22.5 <= Angle[i, j] <= 67.5 or 202.5 <= Angle[i, j] <= 247.5:
                quantized[i, j] = 1
            elif 67.5 <= Angle[i, j] <= 122.5 or 247.5 <= Angle[i, j] <= 292.5:
                quantized[i, j] = 2
            elif 112.5 <= Angle[i, j] <= 157.5 or 292.5 <= Angle[i, j] <= 337.5:
                quantized[i, j] = 3
    return quantized

def Non_Max_Supp(qn, magni, D):
    M = np.zeros(qn.shape)
    a, b = np.shape(qn)
    for i in range(a-1):
        for j in range(b-1):
            if qn[i,j] == 0:
                if magni[i,j-1] < magni[i,j] or magni[i,j] > magni[i,j+1]:
                    M[i,j] = D[i,j]
                else:
                    M[i,j] = 0
            if qn[i,j] == 1:
                if magni[i-1,j+1] <= magni[i,j] or magni[i,j] >= magni[i+1,j-1]:
                    M[i,j] = D[i,j]
                else:
                    M[i,j] = 0
            if qn[i,j] == 2:

```

```

        if magni[i-1,j]<= magni[i,j] or magni[i,j] >= magni[i+1,j]:
            M[i,j] = D[i,j]
        else:
            M[i,j] = 0
    if qn[i,j] == 3:
        if magni[i-1,j-1]<= magni[i,j] or magni[i,j] >= magni[i+1,j+1]:
            M[i,j] = D[i,j]
        else:
            M[i,j] = 0

return M

```

```

def color(quant, mag):
    color = np.zeros((mag.shape[0], mag.shape[1], 3), np.uint8)
    a, b = np.shape(mag)
    for i in range(a-1):
        for j in range(b-1):
            if quant[i,j] == 0:
                if mag[i,j] != 0:
                    color[i,j,0] = 255
            else:
                color[i,j,0] = 0
            if quant[i,j] == 1:
                if mag[i,j] != 0:
                    color[i,j,1] = 255
            else:
                color[i,j,1] = 0
            if quant[i,j] == 2:
                if mag[i,j] != 0:
                    color[i,j,2] = 255
            else:
                color[i,j,2] = 0
            if quant[i,j] == 3:
                if mag[i,j] != 0:
                    color[i,j,0] = 255
                    color[i,j,1] = 255
            else:
                color[i,j,0] = 0

```

```

        color[i,j,1] = 0

    return color

def _double_thresholding(g_suppressed, low_threshold, high_threshold):
    g_thresholded = np.zeros(g_suppressed.shape)
    for i in range(0, g_suppressed.shape[0]): # loop over pixels
        for j in range(0, g_suppressed.shape[1]):
            if g_suppressed[i,j] < low_threshold: # lower than low threshold
                g_thresholded[i,j] = 0
            elif g_suppressed[i,j] >= low_threshold and g_suppressed[i,j] <
high_threshold: # between thresholds
                g_thresholded[i,j] = 128
            else: # higher than high threshold
                g_thresholded[i,j] = 255
    return g_thresholded

def _hysteresis(g_thresholded):
    g_strong = np.zeros(g_thresholded.shape)
    for i in range(0, g_thresholded.shape[0]): # loop over pixels
        for j in range(0, g_thresholded.shape[1]):
            val = g_thresholded[i,j]
            if val == 128: # check if weak edge connected to strong
                if g_thresholded[i-1,j] == 255 or g_thresholded[i+1,j] == 255 or
g_thresholded[i-1,j-1] == 255 or g_thresholded[i+1,j-1] == 255 or g_thresholded[i-
1,j+1] == 255 or g_thresholded[i+1,j+1] == 255 or g_thresholded[i,j-1] == 255 or
g_thresholded[i,j+1] == 255:
                    g_strong[i,j] = 255 # replace weak edge as strong
            elif val == 255:
                g_strong[i,j] = 255 # strong edge remains as strong edge
    return g_strong

# Specify sigma and T value Also calculate Gradient masks
sigma = 0.5
T = 0.3
x, y = MaskGeneration(T, sigma)
gauss = Gaussian(x, y, sigma)
gx = -Create_Gx(x, y)
gy = -Create_Gy(x, y)

```

```

def dilation(img1):

    p, q = img1.shape

    # Define new image to store the pixels of dilated image
    imgDilate = np.zeros((p,q), dtype=np.uint8)

    # Define the structuring element
    SED = np.array([[0,1,0], [1,1,1], [0,1,0]])
    constant1 = 1

    # Dilation
    for i in range(constant1, p-constant1):
        for j in range(constant1, q-constant1):
            temp = img1[i-constant1:i+constant1+1, j-constant1:j+constant1+1]
            product = temp*SED
            imgDilate[i,j] = np.max(product)

    plt.imshow(imgDilate, cmap = "gray")
    cv2.imwrite("./images/dilated_finger.png", imgDilate)

    return imgDilate

def thresholdimg(img, n):
    img_shape = img.shape
    height = img_shape[0]
    width = img_shape[1]
    for row in range(width):
        for column in range(height):
            if img[column, row] > n:
                img[column, row] = 0
            else:
                img[column, row] = 255
    return img

## my code
for filename in os.listdir(image_path):

```

```

image = cv2.imread(image_path + '/' + filename)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

smooth_img = smooth(gray, gauss)
fx = ApplyMask(smooth_img, gx)
fy = ApplyMask(smooth_img, gy)

mag = Gradient_Magnitude(fx, fy)
mag = mag.astype(int)

Angle = Gradient_Direction(fx, fy)

quantized = Digitize_angle(Angle)
nms = Non_Max_Supp(quantized, Angle, mag)

threshold = _double_thresholding(nms, 30, 60)

hys = _hysteresis(threshold)
cv2.imwrite(save_path + '/' + filename.split('.')[0] + "_canny_result.png", hys)
plt.figure(figsize = (10,10))
plt.imshow(hys, cmap='gray')

## opencv
for filename in os.listdir(image_path):

    image = cv2.imread(image_path + '/' + filename)
    edges = cv2.Canny(image, 100, 200)

    plt.subplot(121)
    plt.imshow(image, cmap = 'gray')
    plt.title('Original Image')
    plt.xticks([])
    plt.yticks([])

    plt.subplot(122)
    plt.imshow(edges, cmap = 'gray')
    plt.title('Edge Image')

```

```

plt.xticks([])
plt.yticks([])
plt.show()

cv2.imwrite(save_path + '/' + filename.split('.')[0] + "_canny_result_withcv.png",
edges)

## Test my idea for improving the performance of edge detection
# read image
img = cv2.imread("./images2/Tai_tzu_ying.jpg")

# convert to gray
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# threshold
threshold = thresholding(gray, 127)

# morphology dilate
dilate = dilation(threshold)

# get absolute difference between dilate and thresh
diff = dilate - threshold

# invert
edges = 255 - diff

# write result to disk
cv2.imwrite("./images2/results/Tai_threshold.jpg", threshold)
cv2.imwrite("./images2/results/Tai_dilate.jpg", dilate)
cv2.imwrite("./images2/results/Tai_diff.jpg", diff)
cv2.imwrite("./images2/results/Tai_edges.jpg", edges)

# display it
cv2.imshow("thresh", threshold)
cv2.imshow("dilate", dilate)
cv2.imshow("diff", diff)
cv2.imshow("edges", edges)
cv2.waitKey(0)

```