Labsheet 11
Edge and Line Detection

Ragja Palakkadavath
(SC18M003)

Sub: Image and Video Processing Lab **Department**: Mathematics(MLC)
Date of Lab sheet: October 25, 2018 **Date of Submission**: December 2, 2018

1. Using finite difference kernels

Find the edges of the images using the following kernels :

1. Sobel Operator
2. Prewitts Operator
3. Roberts Operator Show all the intermediate gradient images.

Show all the intermediate gradient images. Comment on the results obtained. How does the kernels behave in presence of noise? Explain the disadvantages of these kernels.

Aim

To perform line detection using finite difference kernels

Theory/Discussion

Gradient of the given image $g(m, n)$ can be obtained using any of the gradient operators (Roberts, Sobel, Prewitt, etc) to get the magnitude the orientation:

$$M(n, n) = \sqrt{g_m^2(m, n) + g_n^2(m, n)}$$

and

$$\theta(m, n) = tan^{-1}[g_n(m, n)/g_m(m, n)]$$

Algorithm

1. Read an Image
2. Form the Sobel, Prewitt, Robert Kernels - discrete form
3. Convolve the image with the edge detection kernels and find out the gradients
4. Repeat the same for noised images.
5. Save the images

Code

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 30 22:36:27 2018

@author: IIST
"""

import numpy as np
from PIL import Image
from scipy import signal
import matplotlib.pyplot as plt

img=Image.open('gaussian_noise.png').convert('L')
img=np.asarray(img)

#vertical kernel
fx=[[-1,0,1],[-2,0,2],[-1,0,1]]

#horizontal kernel
fy=[[-1,-2,-1],[0,0,0],[1,2,1]]

vertical=signal.convolve2d(img,fy)
horizontal=signal.convolve2d(img,fx)

plt.imshow(Image.fromarray(vertical),cmap='gray')
plt.figure()
plt.imshow(Image.fromarray(horizontal),cmap='gray')
```

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 30 23:02:17 2018

@author: IIST
"""

import numpy as np
from PIL import Image
from scipy import signal
import matplotlib.pyplot as plt

img=Image.open('gaussian_noise.png').convert('L')
img=np.asarray(img)

#vertical kernel
fx=[[-1,0,1],[-1,0,1],[-1,0,1]]

#horizontal kernel
fy=[[-1,-1,-1],[0,0,0],[1,1,1]]

vertical=signal.convolve2d(img,fy)
horizontal=signal.convolve2d(img,fx)

#vertimg=Image.fromarray(vertedge)
#horimg=Image.fromarray(horedg)

plt.imshow(Image.fromarray(horizontal),cmap='gray')
```

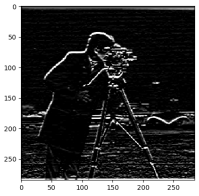
```
plt.figure()
plt.imshow(Image.fromarray(vertical),cmap='gray')

# -*- coding: utf-8 -*-
"""
Created on Fri Nov 30 23:02:29 2018

@author: IIST
"""

import numpy as np
from PIL import Image
from scipy import signal
import matplotlib.pyplot as plt

img=Image.open('gaussian_noise.png').convert('L')
img=np.asarray(img)

fx=[[1,0],[0,-1]]
fy=[[0,1,],[-1,0]]
horizontal=signal.convolve2d(img,fx)
vertical=signal.convolve2d(img,fy)

plt.imshow(Image.fromarray(horizontal),cmap='gray')
plt.figure()
plt.imshow(Image.fromarray(vertical),cmap='gray')
```

Result

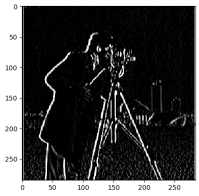
(a) Original Cameraman

(b) Cameraman with Noise

Figure 1: Original Figures

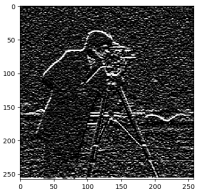


(a) Horizontal Edge Operator

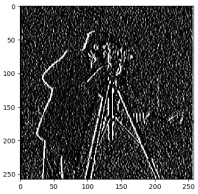


(b) Vertical Edge Operator

Figure 2: Sobel Operator

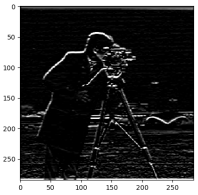


(a) Horizontal Edge Operator

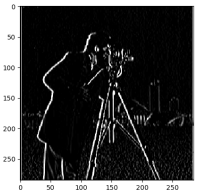


(b) Vertical Edge Operator

Figure 3: Sobel Operator on Noise

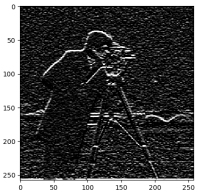


(a) Horizontal Edge Operator

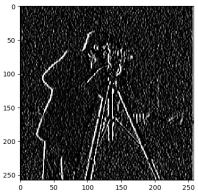


(b) Vertical Edge Operator

Figure 4: Prewitt Operator

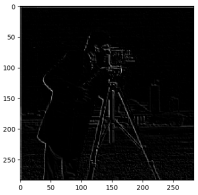


(a) Horizontal Edge Operator

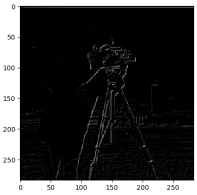


(b) Vertical Edge Operator

Figure 5: Prewitt Operator on Noise

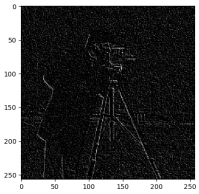


(a) Horizontal Edge Operator

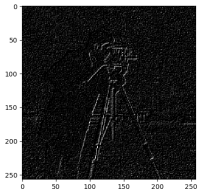


(b) Vertical Edge Operator

Figure 6: Robert Operator



(a) Horizontal Edge Operator



(b) Vertical Edge Operator

Figure 7: Robert Operator on Noise

Inference

Sobel, Prewitt and Robert (finite difference) operators were used to detect edges in cameraman image. However, it was found that when the images were noised, the kernels did not perform that well due to the interference of noise.

2. Using 2nd order derivative kernel

Find the edges of the images using the Laplacian Operator. Compare the results with those obtained in qn1.

Aim

To perform line detection using second order derivative kernel

Theory/Discussion

Laplacian edge detector uses only one kernel. It calculates second order derivatives in a single pass. Laplacian uses highlight gray level discontinuities in an image and try to deemphasize regions with slowly varying gray levels. This operation in result produces such images which have grayish edge lines and other discontinuities on a dark background. This produces inward and outward edges in an image.

Algorithm

1. Read the Image
2. Form the discrete laplacian kernel
3. Perform 2D convolution with the image and kernel
4. Save the resulting image - which consists of the detected edges

Code

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 30 23:37:31 2018

@author: IIST
"""

import numpy as np

from PIL import Image
import matplotlib.pyplot as plt
from  scipy import signal
```

```
import scipy.misc
def get_image():
    #img1 = Image.open('cameraman.png').convert('L')
    #img1.load()
    img2 = Image.open('gaussian_noise.png').convert('L')
    img2.load()
    data1 = np.asarray( img2, dtype="uint8" )
    laplacian=[[0,1,0],[1,-4,1],[0,1,0]]
    #log=[[-12,-24,-12],[-24,-40,-24],[-12,-24,-12]]
    ans1=signal.convolve2d(data1,laplacian)
    plt.imshow(Image.fromarray(ans1),cmap='gray')
get_image()
```

Result

(a) Laplacian with Cameraman Image

(b) Laplacian with Noised Cameraman

Figure 8: Laplacian Operator

Inference

Laplacian operator was used for edge detection. Edges detected were still susceptible to the noise.

3. Using 2D Edge Detection Filters

Find the edges of the images using the Difference of Gaussian(DoG) and Laplacian of Gaussian(LoG) operators. Compare the results with those obtained in qn1 and qn2.

Aim

To convert a gray image to a binary image by finding the threshold value using Otsu's global thresholding algorithm

Theory/Discussion

A second derivative filter based on the Laplacian of the Gaussian is called a **LOG filter**. A LOG filter can be approximated nicely by taking the difference of two Gaussians $g''(x) \approx c_1 e^{-\frac{x^2}{2{\sigma_1}^2}} - c_2 e^{-\frac{x^2}{2{\sigma_2}^2}}$, which is often called a **DOG** filter (for **D**ifference **O**f **G**aussians). For a positive center lobe, we must have $\sigma_1 < \sigma_2$; also, σ_2 must be carefully related to σ_1 to obtain the correct location of zero crossings and so that the total negative weight balances the total positive weight.
The LOG filter responds well to intensity differences of two kinds – small blobs coinciding with the center lobe, and large step edges very close to the center lobe.

Algorithm

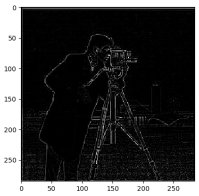
1. Read an Image
2. Form the Laplacian of Gaussian and Difference of Gaussian operators
3. Convolve the image with the edge detection kernels and find out the resulting edge image.
4. Repeat the same for noised images.
5. Save the images

Code

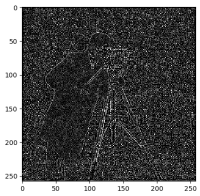
```
# -*- coding: utf-8 -*-
"""
Created on Sat Dec  1 09:58:31 2018

@author: IIST
"""
from PIL import Image
import matplotlib.pyplot as plt
from  scipy import signal
import scipy.misc
import math
import numpy as np
o=1.4
pi=math.pi
log=np.ones((9,9))
k=0
for i in range(-5,4):
    l=0
    for j in range(-5,4):
        log[k,l]=(-1)/(pi*o**4)*(1-(i**2+j**2)/(2*o**2))*math.exp(-(i**2+j**2)/(2*o**2))
        log[k,l]=log[k,l]*10**3
        #print(log[k,l])
        l=l+1
    k=k+1
log=log.astype(int)
print(log)

img2 = Image.open('gaussian_noise.png').convert('L')
img2.load()
data1 = np.asarray( img2, dtype="uint8" )
ans1=signal.convolve2d(data1,log)
plt.imshow(Image.fromarray(ans1),cmap='gray')

# -*- coding: utf-8 -*-
"""
Created on Sat Dec  1 11:53:45 2018

@author: IIST
"""
from PIL import Image
import matplotlib.pyplot as plt
from  scipy import signal
import scipy.misc
import math
```

```
import numpy as np
from scipy.ndimage import filters

N = 9 #filter size
ff = np.zeros((N,N))
ff[int((N-1)/2)][int((N-1)/2)] = 1

g1=filters.gaussian_filter(ff, 0.5)
g2=filters.gaussian_filter(ff, 3)

gauss = g2 - g1

img2 = Image.open('gaussian_noise.png').convert('L')
img2.load()
data1 = np.asarray( img2, dtype="uint8" )
ans1=signal.convolve2d(data1,gauss)
plt.imshow(Image.fromarray(ans1),cmap='gray')
```

Result

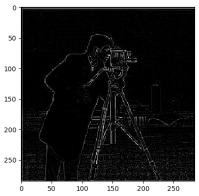


(a) LOG with Cameraman Image

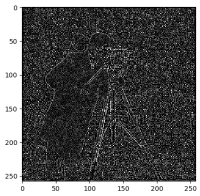


(b) LOG with Noised Cameraman

Figure 9: Laplacian of Gaussian Operator



(a) DOG with Cameraman Image



(b) DOG with Noised Cameraman

Figure 10: Difference of Gaussian Operator

Inference

Edge detection was done based on Laplacian of Gaussian and Difference of Gaussian operators. The same was repeated for noise images too. These operators responded better to noised images.

4. Canny Edge Detection

Apply the Canny Edge Detection algorithm to get the edges of the images. Note the advantages and disadvantages of the algorithm. Give some suggestions to improve on the disadvantages.

Aim

To apply Canny Edge Detector on cameraman and noised cameraman and to observe the results.

Theory/Discussion

Canny edge detection is a multi-step algorithm that can detect edges with noise supressed at the same time. The general criteria for edge detection include:

1. Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible
2. Locality - The edge point detected from the operator should accurately localize on the center of the edge.
3. Reduce the false alarms - a given edge in the image should only be marked once, and where possible, image noise should not create false edges.

Algorithm

---

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

Code

```
# -*- coding: utf-8 -*-
"""
Created on Sat Dec  1 22:51:59 2018

@author: IIST
"""

from scipy import misc
from scipy import ndimage
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from PIL import Image

def Sobel(img, direction):
    if(direction == 'x'):
        fx=[[-1,0,1],[-2,0,2],[-1,0,1]]
        edge=signal.convolve2d(img,fx)
    if(direction == 'y'):
        fy=[[-1,-2,-1],[0,0,0],[1,2,1]]
        edge=signal.convolve2d(img,fy)
    return edge


def normalize(img):
    img = img/np.max(img)
    return img

def nonmaxima_supression(Gmag, Grad):
    NMS = np.zeros(Gmag.shape)
```

```
    for i in range(1, int(Gmag.shape[0]) - 1):
        for j in range(1, int(Gmag.shape[1]) - 1):
            if((Grad[i,j] >= -22.5 and Grad[i,j] <= 22.5) or (Grad[i,j] <= -157.5 and G
                if((Gmag[i,j] > Gmag[i,j+1]) and (Gmag[i,j] > Gmag[i,j-1])):
                    NMS[i,j] = Gmag[i,j]
                else:
                    NMS[i,j] = 0
            if((Grad[i,j] >= 22.5 and Grad[i,j] <= 67.5) or (Grad[i,j] <= -112.5 and Gra
                if((Gmag[i,j] > Gmag[i+1,j+1]) and (Gmag[i,j] > Gmag[i-1,j-1])):
                    NMS[i,j] = Gmag[i,j]
                else:
                    NMS[i,j] = 0
            if((Grad[i,j] >= 67.5 and Grad[i,j] <= 112.5) or (Grad[i,j] <= -67.5 and Gra
                if((Gmag[i,j] > Gmag[i+1,j]) and (Gmag[i,j] > Gmag[i-1,j])):
                    NMS[i,j] = Gmag[i,j]
                else:
                    NMS[i,j] = 0
            if((Grad[i,j] >= 112.5 and Grad[i,j] <= 157.5) or (Grad[i,j] <= -22.5 and G
                if((Gmag[i,j] > Gmag[i+1,j-1]) and (Gmag[i,j] > Gmag[i-1,j+1])):
                    NMS[i,j] = Gmag[i,j]
                else:
                    NMS[i,j] = 0

    return NMS

def hysterisis_threshold(img):
    highThresholdRatio = 0.2
    lowThresholdRatio = 0.15
    GSup = np.copy(img)
    h = int(GSup.shape[0])
    w = int(GSup.shape[1])
    highThreshold = np.max(GSup) * highThresholdRatio
    lowThreshold = highThreshold * lowThresholdRatio
    x = 0.1
    oldx=0
    while(oldx != x):
        oldx = x
        for i in range(1,h-1):
            for j in range(1,w-1):
                if(GSup[i,j] > highThreshold):
                    GSup[i,j] = 1
                elif(GSup[i,j] < lowThreshold):
                    GSup[i,j] = 0
                else:
                    if((GSup[i-1,j-1] > highThreshold) or
                        (GSup[i-1,j] > highThreshold) or
                        (GSup[i-1,j+1] > highThreshold) or
                        (GSup[i,j-1] > highThreshold) or
                        (GSup[i,j+1] > highThreshold) or
                        (GSup[i+1,j-1] > highThreshold) or
                        (GSup[i+1,j] > highThreshold) or
                        (GSup[i+1,j+1] > highThreshold)):
                        GSup[i,j] = 1
        x = np.sum(GSup == 1)

    GSup = (GSup == 1) * GSup

    return GSup
```

```
img = misc.imread("gaussian_noise.png")
img = ndimage.gaussian_filter(img, sigma=1.2)


gx = Sobel(img, 'x')
gx = normalize(gx)

gy = Sobel(img, 'y')
gy = normalize(gy)

dx = ndimage.sobel(img, axis=1)
dy = ndimage.sobel(img, axis=0)


G = np.hypot(gx,gy)
G = normalize(G)

mag = np.hypot(dx,dy)
mag = normalize(mag)

Gradient = np.degrees(np.arctan2(gy,gx))

gradient = np.degrees(np.arctan2(dy,dx))

NMS = nonmaxima_supression(G, Gradient)
NMS = normalize(NMS)

image = hysterisis_threshold(NMS)
plt.imshow(image, cmap = plt.get_cmap('gray'))
plt.show()
```

Result

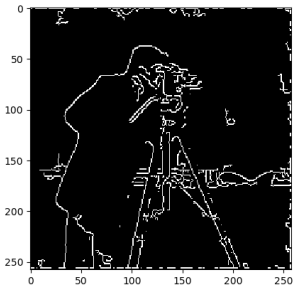


Figure 11: Canny Edge Detection on Image with Gaussian Noise

Inference

Canny Edge Detection was performed on the cameraman image.
Advantages:
Smoothing effect to remove noise.
Good localization and response.
Enhances signal to noise ratio.
Immune to noisy environment.
Disadvantages:
Difficult to implement to reach real time response.

Time consuming.

5. Line Detection

Use Hough Trasform to find the lines present in the image. Use the line.jpg image.

Aim

To detect line in an image using hough transform

Theory/Discussion

Each point (d,T) in Hough space corresponds to a line at angle T and distance d from the origin in the original data space. This is a parametrization. The value of a function in Hough space gives the point density along a line in the data space. The Hough Transform utilises this by the following method.

For each point in the original space consider all the lines which go through that point at a particular discrete set of angles, chosen a priori. For each angle T, calculate the distance to the line through the point at that angle and discretise that distance using an a priori chosen discretisation, giving value d.

Make a corresponding discretisation of the Hough space - this will result in a set of boxes in Hough space. These boxes are called the Hough accumulators. For each line we consider above, we increment a count (initialised at zero) in the Hough accumulator at point (d,T). After considering all the lines through all the points, a Hough accumulator with a high value will probably correspond to a line of points.

Algorithm

---

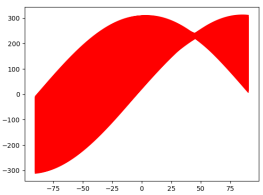
1. Lines in co-ordinate space are converted to points in mc(hough) space
2. Any line passing through
$$(X_a, Y_a) Y_a := m * X_a + c$$
3. Rearranging:
$$c = -X_{am} + Y_a$$
4. The above is the equation of a line in the mc space.

Code

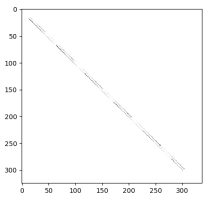
```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec  2 00:47:08 2018

@author: IIST
"""

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2
from mpl_toolkits.mplot3d import Axes3D

A = Image.open('line.jpg').convert('L')
B = np.copy(A)
A = np.array(A)

count = 0
```

```
index = 0
theta = np.linspace(-90, 90, 180)
r, c = np.shape(A)

for i in range(0, r):
    for j in range(0, c):
        if A[i, j] != 255:
            count = count + 1


y = list([] for j in range(0,count))


for i in range(0, r):
    for j in range(0, c):
        if A[i, j] != 255:
            y[index] = (r-i)*np.cos(np.deg2rad(theta)) + j*np.sin(np.deg2rad(theta))
            index = index + 1

for i in range(0,count):
    plt.plot(theta, y[i], color='red')

detect = cv2.line(B,(0,3),(324,321),(255,0,0),2)
dimg = Image.fromarray(detect)
plt.figure()
plt.imshow(dimg)
```

Result



(a) Hough Space



(b) Line Detected via Hough Transform

Figure 12: Hough Transform

Inference

The line detection was performed using the hough transform.