Canny Edge Detector

Eashan Kaushik – ek3575, N19320245

Srijan Malhotra – sm9439, N18390405

Instruction to Run Code:

cv2 library was used for ready so make sure you have cv2 installed.

Or install using **pip install opencv-python** or **pip3 install opencv-python**

Method 1:

From your terminal run the file test.py. This can be done using – **python test.py**

Method 2:

Open the test.ipynb file from jupyter notebook, and run all cells

The output will be generated in the **output folder.**

Output Images

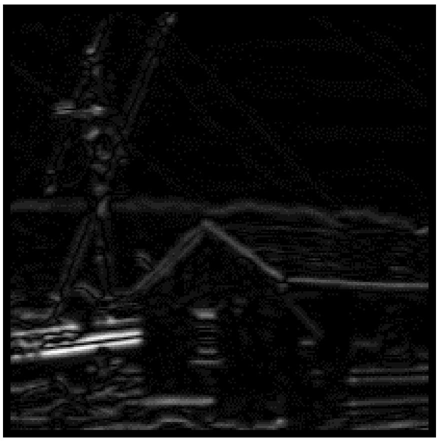
House.bmp

Gaussian Smoothing

Gradient Magnitude

Gradient x

Gradient y



Non Max

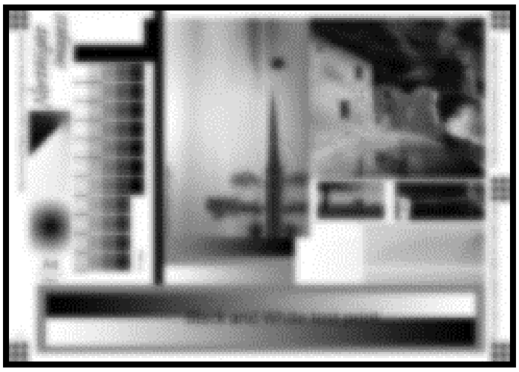
Threshold-25th Percentile

Threshold-50th Percentile

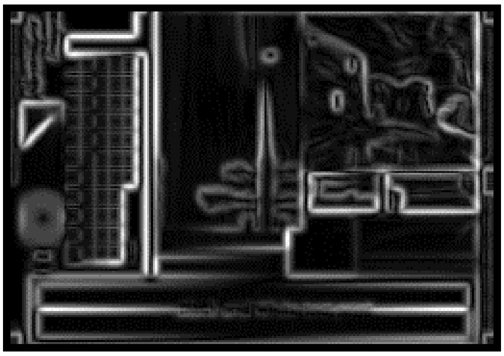
Threshold-75th Percentile

Test Patterns.bmp

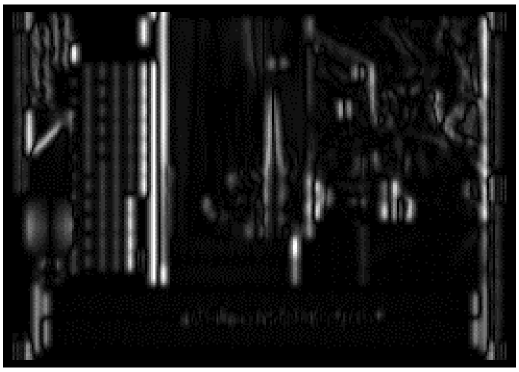
Gaussian Smoothing



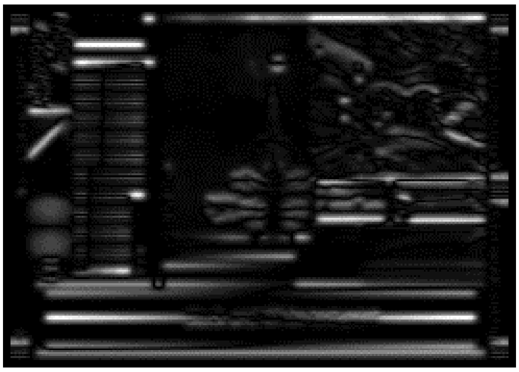
Gradient Magnitude



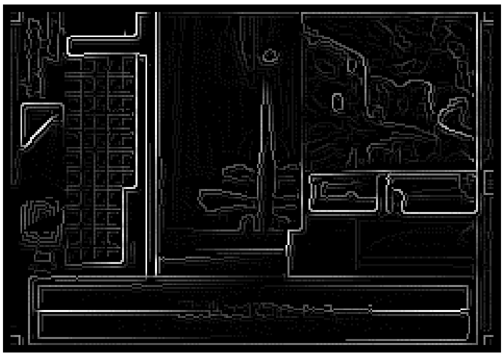
Gradient x



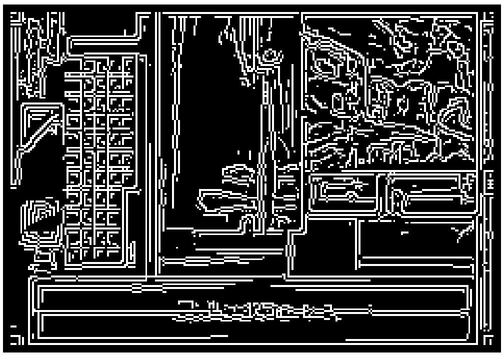
Gradient y



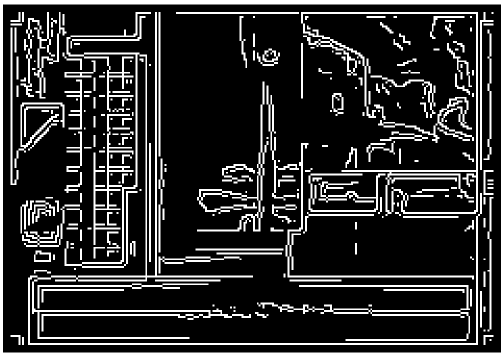
Non Max



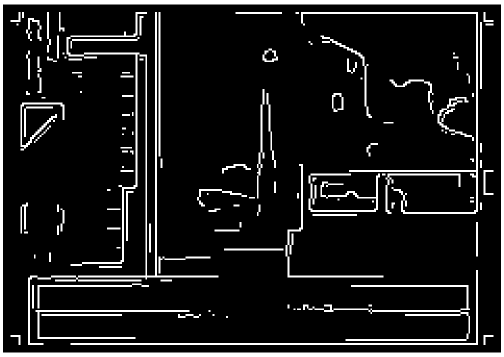
Threshold-25th Percentile



Threshold-50th Percentile



Threshold-75th Percentile



<u>Source Code:</u>

<u>Canny.py</u>

```
####################################################
## Developed by: Eashan Kaushik & Srijan Malhotra ##
## Project Start: 8th October 2021                ##
####################################################
import os
# cv2 is just used for reading images
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt
from PIL import Image as im
import datetime
import math
import copy
import shutil

# The Convolution module is developed by us
from convolution import SeConvolve

# image name to save image after gray scale conversion
GRSC_PATH = 'grsc.PNG'
```

```
class CannyEdgeDetector:

    def __init__(self, image_path):
        # Path of the image, on which canny detector will be applied
        self.image_path = image_path
        # Read Image
        self.image_read()
        # Gaussian Kernel used for smoothing
        self._gaussian_kernel =
np.array([[1,1,2,2,2,1,1],[1,2,2,4,2,2,1],[2,2,4,8,4,2,2],[2,4,8,16,8,4,2],[2,
2,4,8,4,2,2],[1,2,2,4,2,2,1],[1,1,2,2,2,1,1]])
        # Gradient x and y operations Prewitt's Operators
        self._convolution_matrix_gx = np.array([[-1, 0, 1], [-1, 0, 1], [-1,
0, 1]])
        self._convolution_matrix_gy = np.array([[1, 1, 1], [0, 0, 0], [-1, -1,
-1]])
        # Output of step 1
        self._smoothed_image = None
        # Output of step 2
        self._gradient_x = None
        self._gradient_y = None
        self._gradient_x_norm = None
        self._gradient_y_norm = None
        # Output of step 3
        self._magnitude = None
        self._magnitude_norm = None
        # Angle Output
        self._angle = None
        self._edge_angle = None
        # Output of step 4
        self._non_max_output = None
        # Output of step 5
        self._threshold_output_25 = None
        self._threshold_output_50 = None
        self._threshold_output_75 = None

    #######################
    ## Getter and Setter ##
    #######################

    @property
    def gaussian_kernel(self):
        return self._gaussian_kernel

    @property
    def convolution_matrix_gx(self):
        return self._convolution_matrix_gx
```

```python
    @property
    def convolution_matrix_gy(self):
        return self._convolution_matrix_gy

    @property
    def image_matrix(self):
        return self._image_matrix

    @property
    def smoothed_image(self):
        return self._smoothed_image

    @property
    def gradient_x(self):
        return self._gradient_x

    @property
    def gradient_y(self):
        return self._gradient_y

    @property
    def magnitude(self):
        return self._magnitude

    @property
    def angle(self):
        return self._angle

    @property
    def edge_angle(self):
        return self._edge_angle

    @property
    def non_max_output(self):
        return self._non_max_output

    @property
    def threshold_output_25(self):
        return self._threshold_output_25

    @property
    def threshold_output_50(self):
        return self._threshold_output_50

    @property
    def threshold_output_75(self):
        return self._threshold_output_75
```

```
    #######################
    #######################

    # This function reads the path provided to it and calls the
convert_to_matrix
    def image_read(self):

        # self._image_matrix = cv2.imread(self.image_path,
cv2.IMREAD_GRAYSCALE)

        src = cv2.imread(self.image_path)
        self.img = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

        now_time = datetime.datetime.now().strftime("%d%m%Y%H%M%S")

        plt.imsave('gray-op/' + now_time + '-' + GRSC_PATH, self.img,
cmap='gray')

        self.covert_to_matrix('gray-op/' + now_time + '-' + GRSC_PATH)

    # This function converts the image to numpy matrix
    def covert_to_matrix(self, path):

        gsrc = cv2.imread(path, 0)

        matrix = list()

        for row_index in range(0, gsrc.shape[0]):

            row = []

            for column_index in range(0, gsrc.shape[1]):

                pixel = gsrc.item(row_index, column_index)

                row.append(pixel)

            matrix.append(row)

        self._image_matrix = np.array(matrix)

    # Main procedure - calls different functions to compute edge detection
    def canny_detector(self):

        # Gaussian Smoothing - Step1
        self.gaussian_smoothing()
        # Gradient Operation Prewitt - Step 2 and 3
        self.gradient_operation()
```

```
        # Non-Max Supression - Step4
        self.non_max_suppression()
        # Thresholding - Step5
        self.thresholding()
        # Generating output
        self.generate_output()

    # Step 1: Gaussian Smoothing
    def gaussian_smoothing(self):

        # Convolution done on the image_matrix
        smoothing = SeConvolve(self._image_matrix, self._gaussian_kernel)
        _, self._smoothed_image = smoothing.convolution()

    # Step 2: Gradient Operation
    def gradient_operation(self):

        # Convolution done on the image_matrix w.r.t gradient x
        gradient_x = SeConvolve(self._smoothed_image,
self._convolution_matrix_gx, mode='gradient')
        self._gradient_x, self._gradient_x_norm = gradient_x.convolution()

        # Convolution done on the image_matrix w.r.t gradient y
        gradient_y = SeConvolve(self._smoothed_image,
self._convolution_matrix_gy, mode='gradient')
        self._gradient_y, self._gradient_y_norm = gradient_y.convolution()

        # We compute gradient magnitude, gradient angle and edge angle
        self._magnitude, self._magnitude_norm =
self.calcuate_magnitude(self._gradient_x, self._gradient_y)
        self._angle, self._edge_angle = self.calculate_angle(self._gradient_x,
self._gradient_y)

    # Step 3: Magnitude computation
    # This function calculates the gradient magnitude
    def calcuate_magnitude(self, gradient_x, gradient_y):
        height, width = gradient_x.shape

        # After gaussing smoothing and gradient computation we have lost a
total of 8 rows and 8 columns
        magnitude = np.zeros((height - 8, width - 8))

        # looping over the desired matrix
        for i in range(4,height - 4):
            for j in range(4,width - 4):
                # gradient calculated using root(gx**2 + gy**2)
                temp = (gradient_x[i, j] ** 2) + (gradient_y[i, j] ** 2)
```

```
                magnitude[i - 4, j - 4] = math.sqrt(temp)

        # Nomralization of Magnitude
        magnitude_norm = magnitude / 360.624
        # same size as original image
        magnitude = np.pad(magnitude, 4, mode='constant')
        magnitude_norm = np.pad(magnitude_norm, 4, mode='constant')

        return magnitude, magnitude_norm

    def calculate_angle(self, gradient_x, gradient_y):

        height, width = gradient_x.shape

        # After gaussing smoothing and gradient computation we have lost a
total of 8 rows and 8 columns
        angle = np.zeros((height - 8, width - 8))
        edge_angle = np.zeros((height - 8, width - 8))

        # looping over the desired matrix
        for i in range(4,height - 4):
            for j in range(4,width - 4):
                if gradient_x[i, j]  != 0:
                    # gradient angle computed using tan-1(gy/gx)
                    angle[i - 4, j - 4] =
math.degrees(math.atan((gradient_y[i, j] / gradient_x[i, j])))
                    edge_angle[i - 4, j - 4] = angle[i - 4, j - 4] + 90

        # same size as original image
        angle = np.pad(angle, 4, mode='constant')
        edge_angle = np.pad(angle, 4, mode='constant')

        return angle, edge_angle

    # Step 4: Non-Maxima Suppression
    def non_max_suppression(self):
        angle = self._angle
        magnitude = copy.deepcopy(self._magnitude)

        height, width = magnitude.shape

        # looping over the desired matrix
        for i in range(4,height - 4):
            for j in range(4,width - 4):
                # this code calculates the sector the pixel belongs to
according to gradient angle
                if angle[i, j] < 0:
                    current_sector = self.sector(angle[i, j] + 360)
```

```
                else:
                    current_sector = self.sector(angle[i, j])

                # this code returns which pixel we should compare with
according to sector
                check_one, check_two = self.check(current_sector, i, j)
                check_one_x, check_one_y = check_one
                check_two_x, check_two_y = check_two

                # non max suppression
                if not(magnitude[i, j] > magnitude[check_one_x, check_one_y]
and magnitude[i, j] > magnitude[check_two_x, check_two_y]):
                    magnitude[i, j] = 0

        self._non_max_output = magnitude

    # this function returns the sector value according to angle
    def sector(self, angle):
        # for sector 0
        if((0 <= angle <= 22.5) or (337.5 < angle <= 360) or (157.5 < angle <=
202.5)):
            return '0'
        # for sector 1
        elif((67.5 >= angle > 22.5) or (247.5 >= angle > 202.5)):
            return '1'
        # for sector 2
        elif((112.5 >= angle > 67.5) or (292.5 >= angle > 247.5)):
            return '2'
        # for sector 3
        elif((157.5 >= angle > 112.5) or (337.5>= angle > 292.5)):
            return '3'
        else:
            print('wtf')

        # return '0'

    # this function returns which pixel we should compare with according to
sector
    def check(self, current_sector, current_i, current_j):
        if(current_sector == '0'):
            return ((current_i,current_j-1), (current_i,current_j+1))
        elif(current_sector == '1'):
            return ((current_i-1,current_j+1), (current_i+1,current_j-1))
        elif(current_sector == '2'):
            return ((current_i-1,current_j), (current_i+1,current_j))
        elif(current_sector == '3'):
            return ((current_i-1,current_j-1), (current_i+1,current_j+1))
        else:
```

```
            print('wtf')

    # Step 5 Thresholding
    def thresholding(self):

        temp_magnitude = copy.deepcopy(self._non_max_output)
        temp_magnitude = temp_magnitude[4: temp_magnitude.shape[0] - 4, 4:
temp_magnitude.shape[1] - 4].flatten()
        temp_magnitude = temp_magnitude[temp_magnitude != 0]

        # output according to 25th Percentile
        magnitude = copy.deepcopy(self._non_max_output)
        percentile = np.percentile(temp_magnitude, 25)
        self._threshold_output_25 = self.threshold(magnitude, percentile)

        # output according to 50th Percentile
        magnitude = copy.deepcopy(self._non_max_output)
        percentile = np.percentile(temp_magnitude, 50)
        self._threshold_output_50 = self.threshold(magnitude, percentile)

        # output according to 75th Percentile
        magnitude = copy.deepcopy(self._non_max_output)
        percentile = np.percentile(temp_magnitude, 75)
        self._threshold_output_75 = self.threshold(magnitude, percentile)

    # this function performs thresholding
    def threshold(self, magnitude, T):

        height, width = magnitude.shape

        # looping over the desired matrix
        for i in range(4,height - 4):
            for j in range(4,width - 4):

                if magnitude[i,j] < T:
                    magnitude[i,j] = 0
                else:
                    magnitude[i, j] = 1

        return magnitude

    # this function is used to save .PNG images of results
    def generate_output(self):

        name = self.image_path.split('.')[0].split('/')[-1]
        name = 'Output/' + name

        if os.path.isdir(name):
```

```
            shutil.rmtree(name)

        os.mkdir(name)

        # Output of step 1
        plt.imsave(name + '/' + 'GaussianSmoothing.bmp', self._smoothed_image, 
cmap='gray')

        # Output of step 2
        plt.imsave(name + '/' + 'GradientX.bmp', self._gradient_x_norm, 
cmap='gray')
        plt.imsave(name + '/' + 'GradientY.bmp', self._gradient_y_norm, 
cmap='gray')

        # Output of step 3
        plt.imsave(name + '/' + 'GradientMagnitude.bmp', self._magnitude_norm, 
cmap='gray')

        # Output of Step 4
        plt.imsave(name + '/' + 'Non-Max.bmp', self._non_max_output, 
cmap='gray')

        # Output of step 5
        plt.imsave(name + '/' + 'T25.bmp', self._threshold_output_25, 
cmap='gray')
        plt.imsave(name + '/' + 'T50.bmp', self._threshold_output_50, 
cmap='gray')
        plt.imsave(name + '/' + 'T75.bmp', self._threshold_output_75, 
cmap='gray')
```

Convolution.py

```
####################################################
## Developed by: Eashan Kaushik & Srijan Malhotra ##
## Project Start: 29th October 2021               ##
####################################################

import numpy as np

class SeConvolve:

    def __init__(self, image_matrix, kernel, mode='smoothing'):
        # input image matrix
        self.image_matrix = image_matrix
        # inpur kernel
        self.kernel = kernel
        # output after convolution with the kerne
        self._output = None
```

```
        # normalized output
        self._output_norm = None
        # mode smoothing or gradient
        self.mode = mode

    #######################
    ## Getter and Setter ##
    #######################

    @property
    def output(self):
        return self._output

    @property
    def output_norm(self):
        return self._output_norm

    #######################
    #######################

    def convolution(self):

      height, width = self.image_matrix.shape
      # if mode == smoothing
      if self.mode == 'smoothing':
        self._output = np.zeros((height - 6, width - 6))
        # code to perform gaussian smoothing
        # looping over the desired matrix
        for i in range(3,height-3):
          for j in range(3,width-3):
            # martix multiplication for convolution
            # 7 x 7 gaussian smoothing leads to loss of 3 rows and 3 columns
and thats why we start from i-3 and j-3
            self._output[i - 3,j - 3] = (self.kernel[0, 0] *
self.image_matrix[i - 3, j - 3]) + (self.kernel[0, 1] * self.image_matrix[i -
3, j - 2]) + (self.kernel[0, 2] * self.image_matrix[i - 3, j - 1]) + \
            (self.kernel[0, 3] * self.image_matrix[i - 3, j]) +
(self.kernel[0, 4] * self.image_matrix[i - 3, j + 1]) + (self.kernel[0, 5] *
self.image_matrix[i - 3, j + 2]) + \
            (self.kernel[0, 6] * self.image_matrix[i - 3, j + 3]) +
(self.kernel[1, 0] * self.image_matrix[i - 2, j - 3]) + (self.kernel[1, 1] *
self.image_matrix[i - 2, j - 2]) + \
            (self.kernel[1, 2] * self.image_matrix[i - 2, j - 1]) +
(self.kernel[1, 3] * self.image_matrix[i - 2, j]) + (self.kernel[1, 4] *
self.image_matrix[i - 2, j + 1]) + \
            (self.kernel[1, 5] * self.image_matrix[i - 2, j + 2]) +
(self.kernel[1, 6] * self.image_matrix[i - 2, j + 3]) + (self.kernel[2, 0] *
self.image_matrix[i - 1, j - 3]) + \
```

```
            (self.kernel[2, 1] * self.image_matrix[i - 1, j - 2]) +
(self.kernel[2, 2] * self.image_matrix[i - 1, j - 1]) + (self.kernel[2, 3] *
self.image_matrix[i - 1, j]) + \
            (self.kernel[2, 4] * self.image_matrix[i - 1, j + 1]) +
(self.kernel[2, 5] * self.image_matrix[i - 1, j + 2]) + (self.kernel[2, 6] *
self.image_matrix[i - 1, j + 3]) + \
            (self.kernel[3, 0] * self.image_matrix[i, j - 3]) +
(self.kernel[3, 1] * self.image_matrix[i, j - 2]) + (self.kernel[3, 2] *
self.image_matrix[i, j - 1]) + \
            (self.kernel[3, 3] * self.image_matrix[i, j]) + (self.kernel[3, 4]
* self.image_matrix[i, j + 1]) + (self.kernel[3, 5] * self.image_matrix[i, j +
2]) + \
            (self.kernel[3, 6] * self.image_matrix[i, j + 3]) +
(self.kernel[4, 0] * self.image_matrix[i + 1, j - 3]) + (self.kernel[4, 1] *
self.image_matrix[i + 1, j - 2]) + \
            (self.kernel[4, 2] * self.image_matrix[i + 1, j - 1]) +
(self.kernel[4, 3] * self.image_matrix[i + 1, j]) + (self.kernel[4, 4] *
self.image_matrix[i + 1, j + 1]) + \
            (self.kernel[4, 5] * self.image_matrix[i + 1, j + 2]) +
(self.kernel[4, 6] * self.image_matrix[i + 1, j + 3]) + (self.kernel[5, 0] *
self.image_matrix[i + 2, j - 3]) + \
            (self.kernel[5, 1] * self.image_matrix[i + 2, j - 2]) +
(self.kernel[5, 2] * self.image_matrix[i + 2, j - 1]) + (self.kernel[5, 3] *
self.image_matrix[i + 2, j]) + \
            (self.kernel[5, 4] * self.image_matrix[i + 2, j + 1]) +
(self.kernel[5, 5] * self.image_matrix[i + 2, j + 2]) + (self.kernel[5, 6] *
self.image_matrix[i + 2, j + 3]) + \
            (self.kernel[6, 0] * self.image_matrix[i + 3, j - 3]) +
(self.kernel[6, 1] * self.image_matrix[i + 3, j - 2]) + (self.kernel[6, 2] *
self.image_matrix[i + 3, j - 1]) + \
            (self.kernel[6, 3] * self.image_matrix[i + 3, j]) +
(self.kernel[6, 4] * self.image_matrix[i + 3, j + 1]) + (self.kernel[6, 5] *
self.image_matrix[i + 3, j + 2]) + \
            (self.kernel[6, 6] * self.image_matrix[i + 3, j + 3])
      # if mode == gradient
      elif self.mode == 'gradient':

        # code to find the gradients
        self._output = np.zeros((height - 8, width - 8))

        # looping over the desired matrix
        for i in range(4,height - 4):
          for j in range(4,width - 4):
            # martix multiplication for gradient computation
            # prewit convolution after gaussian smoothing leads to loss of 4
rows and 4 columns thats why we start from i-4 and j-4
```

```
            self._output[i - 4,j - 4] = (self.kernel[0, 0] *
self.image_matrix[i - 1, j - 1]) + (self.kernel[0, 1] * self.image_matrix[i -
1, j]) + (self.kernel[0, 2] * self.image_matrix[i - 1, j + 1]) + \
                      (self.kernel[1, 0] * self.image_matrix[i, j - 1]) +
(self.kernel[1, 1] * self.image_matrix[i, j]) + (self.kernel[1, 2] *
self.image_matrix[i, j + 1]) + (self.kernel[2, 0] * self.image_matrix[i + 1, j
- 1]) + \
                      (self.kernel[2, 1] * self.image_matrix[i + 1, j]) +
(self.kernel[2, 2] * self.image_matrix[i + 1, j + 1])

      # we call the normalize function to normalize the output
      self.normalize()

      return self._output, self._output_norm

    # normalize
    def normalize(self):
      if self.mode == 'smoothing':
        # normalize using sum of all values
        self._output_norm = self._output / 140
        self._output_norm = np.pad(self._output_norm, 3, mode='constant')

      elif self.mode == 'gradient':
        # normalize using sum of absolute values
        temp_output = np.absolute(self._output)
        self._output_norm = temp_output / 3
        self._output_norm = np.pad(self._output_norm, 4, mode='constant')
```